ECE454 Final Exam Practice

Eliot Chan

1) P1: $W_x(a)$ $R(x)b$
P2: $W(x)b$ $R(x)a$
P3: $R(x)a$

a) Is this sequentially consistent?

Remember something is sequentially consistent if
- each read in the overall execution order returns the value written by the most recent write
- some operation PX.01 precedes PX.02 for some PX, 01 must precede 02 in the overall execution order as well

We'll begin with precedent constraints:

P1.W(x)a → P2.R(x)a, P3.R(x)a } read out the correct write
P2.W(x)b → P1.R(x)b

P1.W(x)a → P1.R(x)b } retain process order
P2.W(x)b → P2.R(x)a

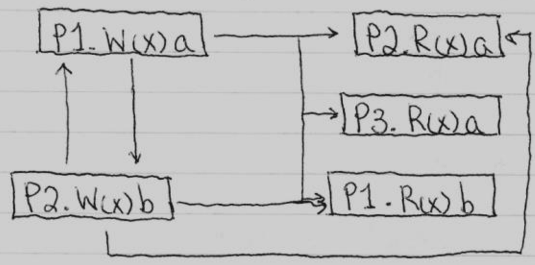
P2 writes b to x, then reads out a from x. This implies that a write occurred between these two operations, so

P2.W(x)b → P1.W(x)a

It's the same in P1, where we write a but read out b, which implies

P1.W(x)a → P2.W(x)b

Obviously, there's something wrong here. We can draw out the whole dependency graph to get a look at the big picture.



There's a cycle between P1.W(x)a and P2.W(x)b. As such we can conclude it is NOT sequentially consistent.

b) Is this causally consistent?

An execution is causally consistent if:

- each read in the ~~overall~~ execution of a specific process reads out the value of the most recent write
- if PX.O1 causally precedes PX.O2* ~~for some PX~~, PX.O1 must causally precede PX.O2 in the overall execution order as well.

where "causally precedes" means either

- O1 precedes O2 in the same process
- O1 reads a value written by O2

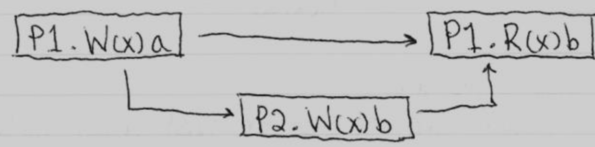
Essentially, there has to be an ordering that makes sense to each process, but doesn't necessarily need to make sense on the whole.

We need to draw a graph for each process, which has its operations AND any writes whose value we read out.

* not necessarily in the same process

P1: We need P1.W(x)a, P1.R(x)b, and P2.W(x)b.

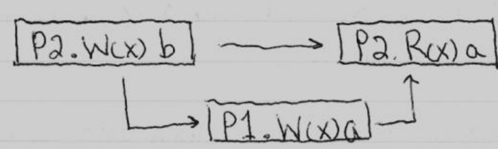
P1.W(x)a → P1.R(x)b (execution order)
~~P2.R(x)b~~
P1.W(x)a → P2.W(x)b (for the read to make sense)
P2.W(x)b → P1.R(x)b (read out correct value)



No cycles here. Let's move on.

P2: We need P2.W(x)b, P2.R(x)a, and P1.W(x)a.

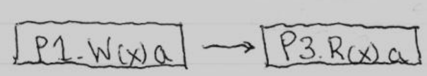
P2.W(x)b → P2.R(x)a (execution order)
P1.W(x)a → P2.R(x)a (read out correct value)
P2.W(x)b → P1.W(x)a (for read to make sense)



No cycles either.

P3: We need P3.R(x)a and P1.W(x)a.

P1.W(x)a → P3.R(x)a (for the read to make sense)



No cycles. Therefore this execution is causally consistent.

c) Is this execution linearizable?

The diagram given is a bit funky-looking so I'll redraw it.

P1: o—W(x)a—o o—R(x)b—o
P2: o—W(x)b—o o—R(x)a—o
P3: o—R(x)a—o

An execution is linearizable if:

- each read returns the value of the most recent write
- if O1 <u>FINISHES</u> before O2, O1 must precede O2 in the overall execution order

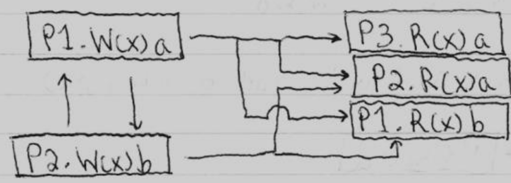
Again, let's look at constraints.

P1.W(x) a → P2.R(x)a, P3.R(x) a } write before read
P2. W(x) b → P1. R(x) b.

P1. W(x) a → P1. R(x) b, P2.R(x) a, P3.R(x) a } finishes before
P2. W(x)b → P2. R(x) a, P3. R(x) a

P1. W(x) a → P2. W(x) b because P1 writes a then reads b
P2. W(x) b → P1. W(x) a because P2 writes b then reads a

We can see a cycle already.



As such, this execution is NOT linearizable.

2) Consider a Dynamo-style quorum-replicated storage system such as Apache Cassandra. Each data object is replicated in EACH of DC1, DC2, and DC3. One-way network delays are shown below.

	DC1	DC2	DC3
DC1	0ms	15ms	20ms
DC2	15ms	0ms	25ms
DC3	20ms	25ms	0ms

Assume storage servers have accurate estimates of these latencies, and ignore processing delays.

A workload is 80% GET and 20% PUT. Assume all gets are issued by clients in DC1, and all puts by DC3.

What client-side consistency settings (ONE, QUORUM, ALL) should be used so that $N_R + N_W > N$ is satisfied and average latency is minimized? State one consistency level for gets and one consistency level for puts. Show latency calculations.

Our options for (N_R, N_W) are as follows:

$(1, 3) \rightarrow$ (ONE, ALL)
$(2, 2) \rightarrow$ (QUORUM, QUORUM)
$(3, 1) \rightarrow$ (ALL, ONE)

We don't want more than 4 because ~~[illegible]~~ that's more than required to satisfy the $N_R + N_W > N$ requirement. So let's do calculations on all 3.

(1, 3): read: 80% × (0ms) = 0ms
write: 20% × (0ms + 20ms + 25ms)* = 9ms
total: 9ms

* should be max of these latencies, not the sum

(2,2): read: 80% × (0ms + 15ms) = 12ms
choose lower one for better latency
put: 20% × (0ms + 20ms) = 4ms
total: 16ms

*3 should be max as well

(3,1): read: 80% × (0ms + 15 ms + 20ms)[*3] = 28 ms
put: 20% × (0 ms) = 0ms
total: 28 ms

As such, (ONE, ALL) is the setting that satisfies $N_R + N_W > N$ and provides the lowest latency, at 9 ms.**

3)

a) Under what combination of client-side consistency settings does Cassandra behave like an AP system in the context of Brewer's CAP principle?

Remember that CAP is Consistent - Available - Partition Tolerant, and Brewer states that distributed systems can only be two out of three of these things.*

So when is Cassandra available and partition-tolerant but not consistent? Consistency is when $N_R + N_W > N$, so we must be violating this constraint.

This happens when we use the write setting ANY, and read setting is anything except ALL, as ANY means $N_W = 1$. Using write ANY allows Cassandra to use hinted handoff, which allows any given node to accept an update for a key until a replica becomes available.

** should be doubled as I forgot the trip back

* technically: in the event of a partition, system must choose A or C.

b) Anti-entropy (reversing disorder) is used by Cassandra to ensure eventual consistency. Cassandra does this by periodically exchanging Merkle (hash) trees between nodes.

c) Compare and contrast NFSv4 and HDFS.

NFSv4 - supports session semantics (changes on close)
- supports byte range file locking

HDFS - files are immutable (no changes possible)
- has an append function for log-esque data

d) What are the two principle advantages of weak consistency models versus strong consistency models?

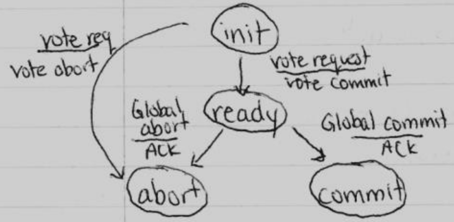
1) Latency
Since we don't need to spend time worrying about if other nodes properly replicated our changes, weakly consistent systems have lower latency. We can OK a request once the node that received the request updates its data without needing to wait on other nodes.

2) Availability
Since strongly consistent systems work on quorums and majority, we need a majority of the system to be up for it to work. In weakly consistent systems, we're more available because we don't need as much of the system to be up to function properly.

4) a) Define "recovery line" in the context of distributed checkpoints.

A recovery line is the most recent distributed snapshot - a time where no send events are awaiting receive events, and the system is stable and "cycles endlessly and uselessly", as Lamport and Chandy say.

b) Draw the state diagram for the participant in the two-phase commit protocol.



Let's go over the steps a participant does.

1) Write init to log
2) Wait for vote request
3) If it times out, write vote_abort
4) On receiving vote request, we can vote to commit

4a) write vote-commit to log
4b) ~~write~~ Send vote-commit to coordinator
5) wait for decision
5a) if it times out, wait for decision from other participants after multicasting decision_request
5b) write decision to log
6) write decision to log

c) Explain rigorously the difference between the distributed commitment problem and the consensus problem. Refer to safety properties.

Remember that safety properties say that "nothing bad should happen". So what bad things do we want to prevent in each case?

The distributed commit problem is worried about whether an update completes or not: the atomicity of an update. We want to prevent partial updates - they should update on all replicas, or none of them.

In the consensus problem, we're worried about what order we should perform operations. Specifically, which operation proposed by a process should we decide on? So we want to ensure a) we only decide on one thing, b) we can only decide on something that was proposed, and c) if some value is proposed, a learner will eventually learn of some value (though it might not be the one proposed).

a and b are safety properties, while c is a liveness property, as it has to do with the need for the protocol to make progress.

d) In 2PC, is it possible for one participant to be in the init stage while another is in the commit stage?

No. All participants will enter the ready state once the coordinator sends out vote-request. If a participant never receives, a vote-request, it will enter the abort stage.

The process is unable to continue past ready unless all participants are ready.

e) State the safety properties of 2PC rigorously.

The safety properties for 2PC are the safety properties for transactions - ACID.

A: atomicity. All updates take effect, or none. The transaction should never be left partially complete.

C: consistency. Referential integrity should be upheld.

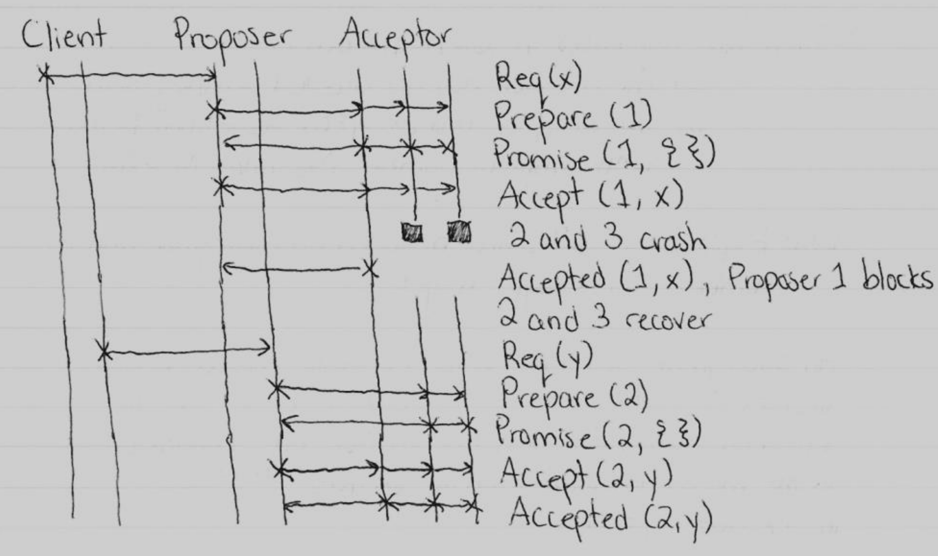
I: isolation. Operations within a transaction should be serializable.

D: durability. Updates that have been committed should not be lost, even in the event of failures.

5) Assume there are 3 Paxos acceptors, and a quorum is two out of three.

a) Is it possible for the same acceptor to accept two distinct values at different times, with distinct proposal numbers?

Yes, it is.



Acceptor 1 has accepted x with PN=1, then y with PN=2.

If you need a more in-depth explanation of this, check out the last example in my Paxos notes.

b) What does it mean for Paxos to converge?

Paxos converges when a majority of acceptors accept the same value.

c) What is the minimum number of learners required for Paxos to converge?

Zero. Learners are only to notify clients or to do external-facing work based on the result of an accepted value. They don't have anything to do with the consensus part itself.

d) Suppose an acceptor receives an accept request with proposal number N and value V. Under what assumptions does the acceptor accept V and reply with Accepted?

Only when the acceptor has not promised an N higher than the one given.

* There are two cases in which this occurs.
  - a) The acceptor has not accepted anything before
  - b) The acceptor has accepted something before, with proposal number N_0, and $N > N_0$

6) a) Explain the difference between an aggregator and a combiner in Google Pregel.

An aggregator performs some operation based on ALL vertices a particular server is responsible for, and the values are aggregated in a tree structure and the root value is handed off to the master to share with all vertices in the next superstep.

Combiners work similarly to Hadoop combiners: they perform user-defined, commutative and associative operations at EACH vertex (for example, finding the sum), and allow you to send less over the network by performing the calculations before.

b) Explain the purpose of the damping factor in the iterative PageRank algorithm. Use relevant concepts from linear algebra in your answers.

The damping factor is required to create a column stochastic matrix with strictly positive values. By the Perron-Frobenius theorem, this guarantees that there will be a single eigenvector with positive components of multiplicity 1 for the eigenvalue $\lambda = 1$.

This ensures that the power method can converge upon the single eigenvector representing the page rank of each given page.

c) ~~The formula for M is as~~ In the iterative PageRank algorithm, explain concisely what happens to the output PageRank vector if the damping ratio is set to 1.

The matrix M is given by:

$$M = (1-\delta)\begin{bmatrix} 1/n & \cdots \\ \vdots & \end{bmatrix} + \delta A$$

$$\delta = 1$$
$$\Rightarrow M = A$$

So it is possible that the output vector will converge upon a solution with zero as a rank, or not converge to a solution at all.

d) Explain the difference between an ephemeral node and a sequential node in Zookeeper.

An ephemeral node gets destroyed when its creator terminates.

A sequential node is given a name that increases monotonically for each other sequential node ~~is~~ created.

e) Write down a Zookeeper recipe (in pseudo-code format) for solving consensus. Explain how each process proposes a value, and how it discovers the decision.

```
Propose ():
    Create Sequential Node (proposed Value)

Decide (): Children
nodes = get All Nodes (). sort()  // sort by ascending node number
    getValueFrom (nodes.get(0))
```

This consensus protocol is based on a first-one-wins basis. The decided value is associated with the lowest zNode number.

f) Explain the "herd effect" in a Zookeeper recipe, and how it is avoided in the particular recipe given in the lecture for implementing exclusive locks.

The "herd effect" is anything that would create a burst of network traffic at some point in time, like when we use polling or timers. This is undesirable because it can limit our scalability in that we shouldn't need to handle big bursts when we could have a consistent flow instead.

The lecture example is transcribed below

1) id = create ("../locks/x-", SEQUENCE|EPHEMERAL);
2) getChildren ("../locks", false)
3) if id is the first child, exit
4) exists (name of child before id, true)
5) if it does not exist, goto 2)
6) wait for event
7) goto 2)

The key in this question is in step 4 - but first let's walk through it a little more comprehensively.

1) We create a sequential and ephemeral node, and get its id.
2) We get a list of all children under the locks node. We do NOT set a watcher on the list.
3) If our id is the first in the list, that means we can access the critical section, do what we need to do, and then exit
4) If we're not the first, we check for the existence of the person immediately before us in the queue, and put a watcher on him.

   This is the kicker. By only watching the immediately preceding node, we avoid a mass of messages that would arise by watching the entire list of children.
5) If it doesn't exist, that means we can check again if we're at the front of the line.
6) We can wait for our watcher to fire before checking again for if we're first in line.

7) The following code snippet solves the word counting problem. Find a concurrency bug and fix it by re-writing parts of the code. Keep your modifications simple.

```
1.  public class WordCountParallel implements Runnable {
2.          private final String buffer;
3.          private final ConcurrentMap<String, Integer> counts;
              ⋮
4.          private void updateCount (String q) {
5.              Integer oldVal, newVal;
6.              Integer cnt = counts.get(q);
7.              if(cnt == null){
8.                  oldVal = counts.put(q, 1);
9.                  if (oldVal == null) return;
10.             }
11.             do {
12.                 oldVal = counts.get(q);
13.                 newVal = (oldVal == null) ? 1 : (oldVal +1);
14.             } while (!counts.replace(q, oldVal, newVal));
15.         }
              ⋮
16. }
```

So there's three things I can identify as possible concurrency issues here.

7-10: This check is not atomic. We could conceivably have two threads discover that the entry in the map doesn't exist, and both put, overwriting each other and miscounting the number of words.

We can replace this by using

```
counts.putIfAbsent(q, 1)
```

which is an atomic operation.

All: We can also use read-write locks, on the get and put operations. If we want to make this simpler, we could use regular locks.

```
private ReadWriteLock lck = new ReentrantReadWriteLock();
lck.readLock().lock();
try {
    cnt = counts.get(q);
} finally {
    lck.readLock().unlock();
}
```

All: Alternatively, we could use synchronize blocks on the pieces of code that access or modify the map.

```
synchronized (this) {
    cnt = counts.get(q);
}
```