Recap

**Linearizability:**
- Any history of read/write operations has some equivalent sequential ordering.
- This ordering preserves completion-to-issue order in the original history.

**How to implement linearizability?**
- Partition state space among servers.
- Example: Figure 1: S_a serializes all operations w.r.t. object a, S_b serializes operations w.r.t. object b.

**Pros and Cons:**
- **Pros:**
  - Strongest yet still practical semantics: requires lots of coordination among nodes, resulting in long latencies if replicas are far away.
  - Unavailability if network is partitioned.

- **Cons:**
  - Many relaxed consistency models: Goal: better performance, less coordination.
  - Tradeoff: less intuitive semantics (more anomalies).

**Today:** Eventual consistency.

**Same name, several meanings:**
- For lower latency and better availability:
  - Accept a write before being able to serialize it
  - Reads can return a (possibly) stale value.
  - Anomalies!

**Example:**
- Write a and b to album, then b and a:
  - Write-writer conflict: S_a can accept writes, and S_b can accept writes.
  - S_a and S_b can both accept a write before being able to serialize it.

**Idea:** Ordered updates at each node.
- Ordering updates at each node: B < b if b < b.
- Ordering updates across nodes: A < B if A < B.
- Example:
  - A at t = 1, B at t = 2, C at t = 3, D at t = 4.
  - Write a at t = 2, then b at t = 3, then c at t = 4.
  - Write a at t = 2, then b at t = 3, then c at t = 4.

**Challenge:**
- How to implement eventual state convergence?

**Goal:**
- Eventual state convergence.
- Consistency model is partition-tolerant, resulting in eventual availability.

**Desired properties:**
- Complete convergence of state.
- Eventual output.
- Limited by causality.
- Liveness: eventually output.

**Operations:**
- Add album and photo to album, delete photo etc.

**Example:**
- Alex writes a photo album to the cloud.
- Alex updates photos in the album.
- Sarah writes a comment to the cloud.
- Sarah updates comments in the album.

**Example:**
- Alex writes a photo album to the cloud.
- Alex updates photos in the album.
- Sarah writes a comment to the cloud.
- Sarah updates comments in the album.

**Conclusion:**
- All devices should be able to get and use partial consistency in the network and other nodes.
- All devices should cache partially consistent data from the cloud.
- All devices should be able to use partial consistency in the network and other nodes.
- All devices should cache partially consistent data from the cloud.
Will update order be consistent with causality?
What if A adds a photo pid1, then B sees it, then B deletes pid1
Perhaps:
\(<10,A>\) add
\(<9,B>\) delete -- B’s clock is slow
Now delete will be ordered before add!
Unlikely to work:
Diffs from wall-clock time case b/c system *knew* B had seen the add

Lamport logical clocks
Want to timestamp events s.t.
node observes E1, then generates E2, TS(E2) > TS(E1)
Thus other nodes will order E1 and E2 the same way.
Each node keeps a clock T
in increments T as real time passes, one second per second
\(T = \max(T’, T’+1)\) if sees T’ from another node
Note properties:
E1 then E2 on same node \(\Rightarrow\) TS(E1) < TS(E2)
But it’s a partial order
TS(E1) < TS(E2) does not imply E1 came before E2

Logical clock solves add/delete causality example
When B sees \(<10,A>\), B will set its clock to 11, so
B will generate \(<11,B>\) for its delete

Irritating that there could always be a long-delayed update with lower TS
That can cause the results of my update to change
Would be nice if updates were eventually "stable"
\(\Rightarrow\) no changes in update order up to that point
\(\Rightarrow\) results can never again change -- e.g. you know for sure pid1 is at position
3. \(\Rightarrow\) no need to re-run update function

How about a fully decentralized "commit" scheme?
You want to know if update \(<10,A>\) is stable
Have sync always send in log order -- "prefix property"
If you have seen updates w/ TS > 10 from *every* node
Then you’ll never again see one \(<10,A>\)
So \(<10,A>\) is stable
Why doesn’t Bayou do something like this? (Bayou commits updates through designated primary replica)

How to sync?
A sending to B
Need a quick way for B to tell A what to send
A has:
\(<,10,X>\), \(<,20,Y>\), \(<,30,X>\), \(<,40,X>\)
B has:
\(<,10,X>\), \(<,20,Y>\), \(<,30,X>\)
At start of sync, B tells A *X 30, Y 20*
Sync prefix property means B has all X updates before 30, all Y before 20
A sends all X’s updates after \(<,30,X>\), all Y’s updates after \(<,20,X>\), etc
This is a version vector -- it summarizes log content
It’s the "P" vector in Figure 4
A’s F: \([X:40,Y:20]\)
B’s F: \([X:30,Y:20]\)

How did all this work out?
Replicas: write any copy, and sync are good ideas
Now used by both user apps *and* multi-site storage systems
Requirement for p2p interaction is debatable

Bayou introduced some very influential design ideas
Bayou made good use of existing ideas
Logical clock

COPS [SOSP’11]
System setup:
Multiple data centers, separated by long distance links
Each data center has many nodes, storage state is fully replicated at each data center
Desired performance:
Writes finish w/o waiting for remote sites (async. replication)
Reads contact local site only

What’s causal consistency?
Systems that obey the following set of partial orders
1. if op1 and op2 are in the single thread of execution and op1 is issued before op2, then op1 \(\Rightarrow\) op2.
   - (On client1, op1: creates pid1, op2: adds pid1 to album aid. All nodes see the effect of op2 after op1)
2. If op2 reads the result written by op1, then op1 \(\Rightarrow\) op2.
   - (On client1, op2: adds pid1 to album aid on client 2, op2: reads pid1 in album)
3. if op1 \(\Rightarrow\) op2, op2 \(\Rightarrow\) op3, then op1 \(\Rightarrow\) op3
   - (On client 1, op1: adds pid1 to album aid on client 2, op2: reads pid1 in album, op3: deletes pid1. All nodes see op1 \(\Rightarrow\) op2 \(\Rightarrow\) op3, i.e. pid1 is deleted)

Does Bayou provide causal consistency? Is it scalable?
COPS’ approach
partition key-space among nodes
explicitly keep track explicit dependencies (partial orders) for each write
Site A performs a write, replicates it together with the dependencies to another site B.
Site B waits until the write’s dependencies are satisfied in B before committing the write.

Client library
put(key, value, context); //put’s dependencies are set by context, new dependency includes the new put version.
value = get(key, context); //add dependencies of get to context

Anomalies under causal consistency
-- write-write conflict
-- do not capture causality caused by external communication. I posted a picture, called my friend to check it out.

Parts of the notes is due to Robert Morris