```
Recap
Linearizability:
        * any history of read/write ops has some equivalent sequential ordering
        * equivalent sequential order preserves completion−to−issue order in the
 original history.


        how to implement linearizability?
        partition state space among servers
        [figure−1]

        Sa    Sb  (S_a serializes all ops w.r.t. object a, S_b serializes object
 b)




        C0    C1 (possibly w/ cache)

        Pros and Cons:
        + strongest yet still practical semantics
        − require lots of coordination among nodes, resulting in
          − long latency operation if replicas are far away
                  − unavailability if network is partitioned

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

Today: Eventual consistency
        Same name has several meanings

For lower latency & better availability:
        * accept a write before being able to serialize it.
        * reads can return a (possibly) stale value than blocking for the latest
 value.

Anomalies! (draw a diagram next to figure−1 where both Sa and Sb can accept writ
es)
  * write−write conflict
   (Sa writes a=1, Sb writes a=2, the corresponding replicated writes cause a=2
at Sa, and a=1 at Sb)
  * stale read
   (C1 writes b=3 at Sb, C0 reads b=* at Sa)
  * loss of causality
   (C1 writes b=3, then b’=2 at Sb, C0 reads b’=2 but b=0 at Sa)

Desired properties:
  * Eventual convergence of state
  * Causality perserving

Let’s build a shared photo gallery accessibly by mobile devices:
 gallery consists of many albums, each identified by an aid
 Each album (aid) contains an *ordered* list of photos each identified by a pid.

 Operations: add album, add photo to album, delete photo etc.

Design #1:
All devices send reads/writes to some storage service in the cloud.
   Storage is linearizable using partitioned primary/backup approach
App is available only when connected to the cloud.

Design #2:
Each device can act as a storage server. A device caches photos/albums and can m
odify its local copy. Reads read from local copy.
App is available despite periodic connectivity to net and other nodes.

Straw man 1: merge storage.
  What if there’s a write−write conflict? Add two photos to the same album "at t
```

```
he same time"
   But we want automatic conflict resolution.

Idea: update functions
  Have update be a function, not a new value.
        e.g. add pid to album w/ aid.
  Read current state of storage, decide best change.
  Function must be deterministic
    Otherwise nodes will get different answers

Challenge:
  A: add pid1 to album w/ aid
        B: add pid2 to album w/ aid
  X syncs w/ A, then B
  Y syncs w/ B, then A
  Will X show pid1 in front of pid2, and Y show pid2 in front of pid1?
  If update function is commutative (e.g. album is an *unordered* set of photos)
, then it dos not matter.

Goal: eventual state convergence

Idea: ordered update log
  Ordered list of updates at each node.
  Storage state is result of applying updates in order.
  Syncing == ensure all nodes have same updates in log.

How can nodes agree on update order?
  Update ID: <time T, node ID>
  Assigned by node that creates the update.
  Ordering updates a and b:
    a < b if a.T < b.T or (a.T = b.T and a.ID < b.ID)

Example:
 <10,A>: add pid1 to album aid
 <20,B>: add pid2 to album aid
 What’s the final ordered list in aid?
   the result of executing update functions in timestamp order
   [..pid1, pid2] (not [..pid2, pid1])

What’s the status before any syncs?
  I.e. content of each node’s storage state
  A: pid1 at 3rd position for album aid
  B: pid2 at 3rd position for album aid
  This is what A/B user will see before syncing.

Now A and B sync with each other
  Both now know the full set of updates
  Can each just run the new update function against its storage state?
    A: pid1 at 3rd position, pid2 at 4th position
    B: pid2 at 3rd position, pid1 at 3rd position
  That’s not the right answer!

Roll back and replay
  Naive way: Re−run all update functions, starting from empty storage state
  Since A and B have same ordered set of updates
    they will arrive at same final state
  We will optimize this in a bit

Displayed photo positions are "tentative"
  B’s user saw a photo pid2 at 3rd position, then it’s changed to 4th position
  You never know if there’s some other photo from nodes you haven’t yet synced
    That will change the pid1’s position yet again

Will update order be consistent with wall−clock time?
  Maybe A went first (in wall−clock time) with timestamp <10,A>
  Node clocks are not perfectly synchronized
  So B could then generates <9,B>
  B’s meeting gets priority, even though A asked first
```

```
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
    Differs 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
    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 => 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"
    => no changes in update order up to that point
    => results can never again change -- e.g. you know for sure pid1 is at posit
ion 3.
    => 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 desig
nated 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>, &c
  This is a version vector -- it summarize log content
    It’s the "F" 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
```

```
    clients (phones, ipads) can just (sporadically) contact the servers
  Bayou introduced some very influential design ideas
    Update functions
    Ordered update log
    Allowed general purpose conflict resolution
  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 ea
ch 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 issue
d before op2, then op1 --> op2.
           (On client1, op1: creates pid1, op2: adds pid1 to album aid. All node
s see the effect of op2 after op1) 2. If op2 reads the result written by op1, th
en op1--> op2
           (On client1, op1: adds pid1 to album aid On client 2, op2: reads pid1
 in album)
        3. if op1-->op2, op2-->op3, then op1-->op3
           (On client 1, op1: adds pid1 to album aid On client 2, op2: reads pid
1 in album, op3: deletes pid1. All nodes see op1--->op2--->op3, i.e. pid1 is del
eted)

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 co
mmitting the write.


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

                                        dependencies
        Client 1:  put(x,1) ---> put(y,2)
                   ctx1:x1       store (x1) with y2, ctx=x1,y2
        Client 2:                get(y)=2 --> put(x,4)
                                 ctx2:x1,y=2   store (y2), ctx2:x=4,y=2
        Client 3:                             get(x)=4) --> put(z,5)
                                              ctx3:x=4,y=2  store (x=4,y=2), ctx
3:x=4,y=2,z=5

        Site A replicate y2 with dependency (x1) to site B.
        Site B performs a dependency check locally to wait for x1 to commit befo
re committing y2.

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

Parts of the notes is due to Robert Morris
```