geo-distributed storage in data centers

marcos k. aguilera

microsoft research silicon valley

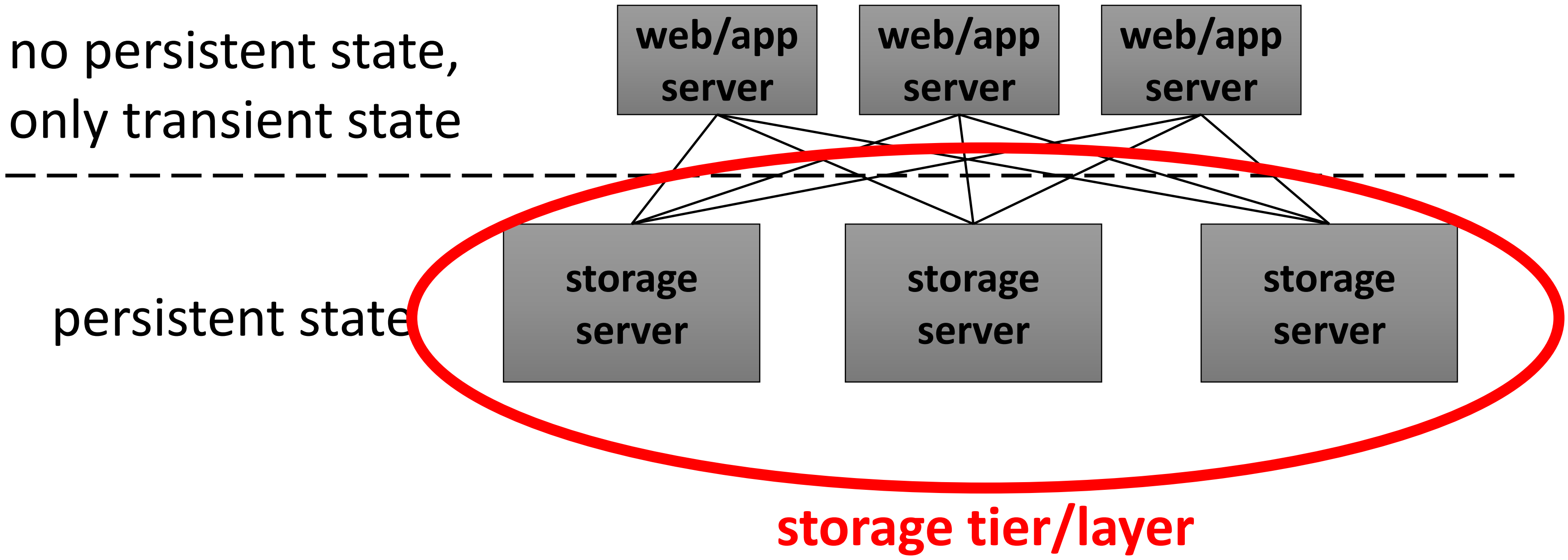
context: large web applications

- examples

  microsoft: *bing, hotmail, skype;* **google**: *search, gmail;*
  yahoo!: *search, email, portal;* **amazon:** *store;*
  facebook, twitter: *social network;* **ebay:** *auctions*
- hosted in large data centers around the world
- tons of users and data

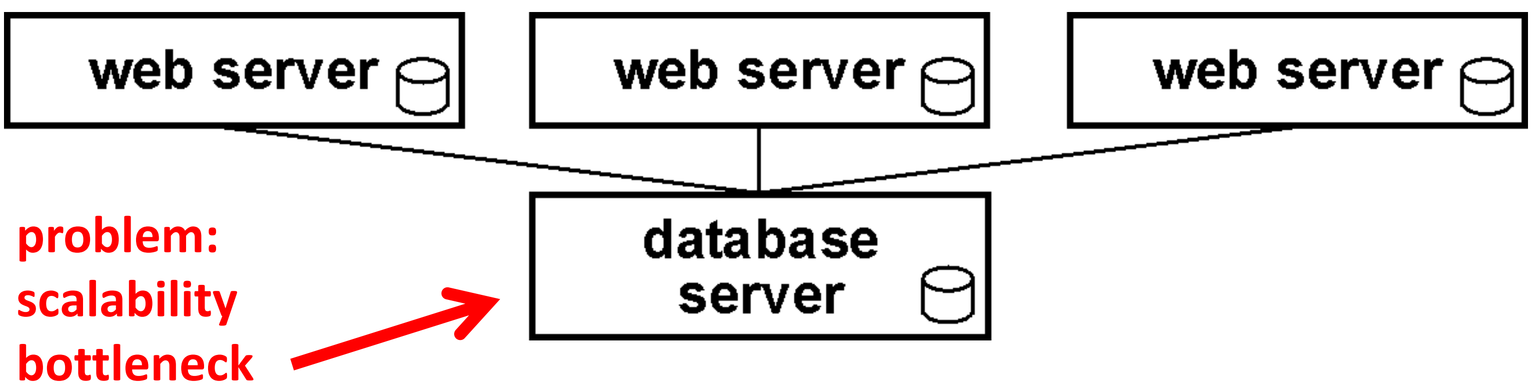
service	**millions of users** (source: comscore, 5/2012)	**max space/user**
hotmail,outlook.com	325	unlimited
yahoo mail	298	1024 GB
gmail	289	15 GB

general architecture of web app



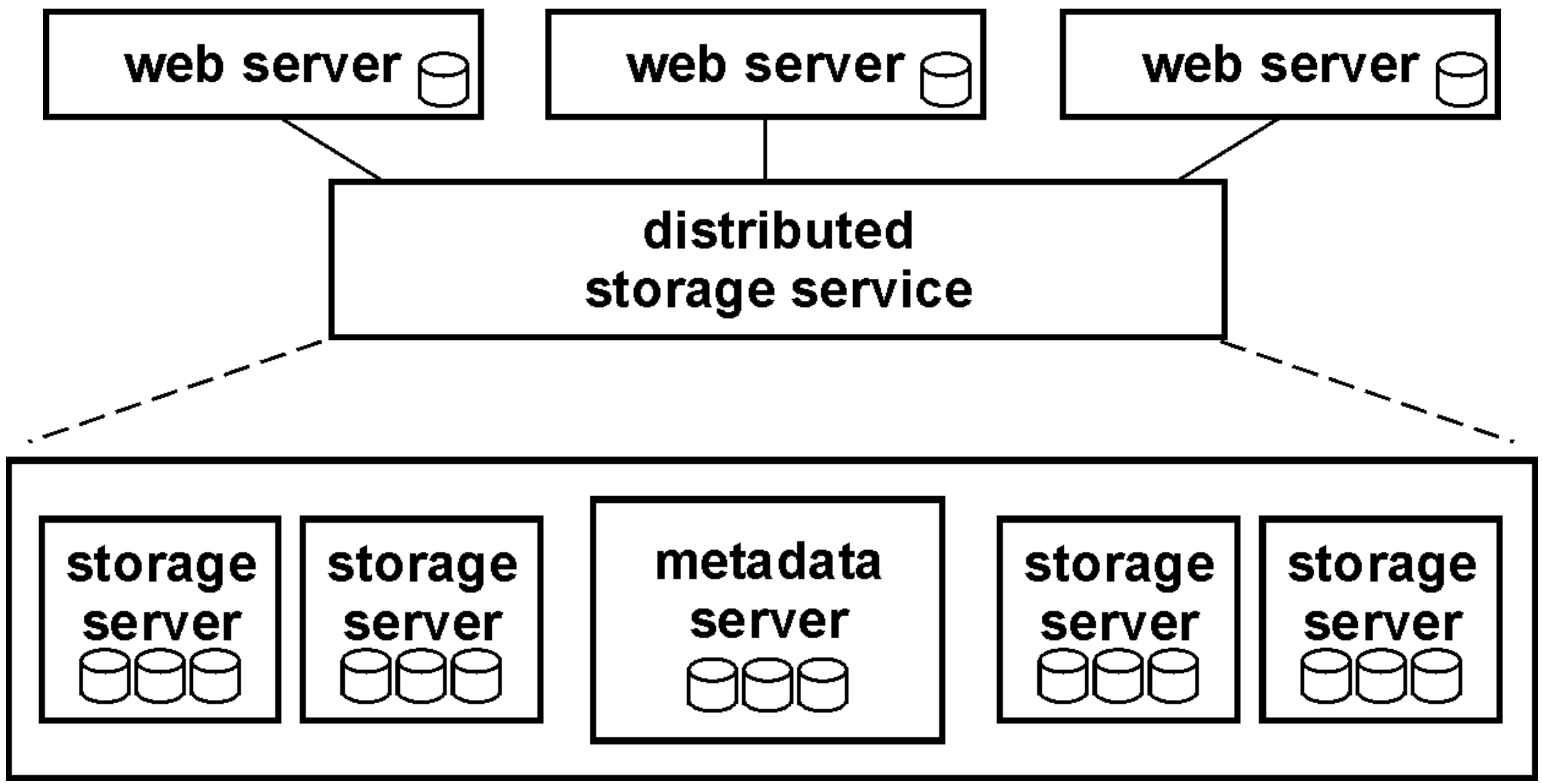
history of storage of web apps

- first generation (90s): local file system
  - static html pages
  - scalability from content distribution networks
- second generation (mid-late 90s): multi-tier systems
  - mix of file system (static content)
    and database system (dynamic content)
  - php, perl, python, database server
  - java (and later .net), database server



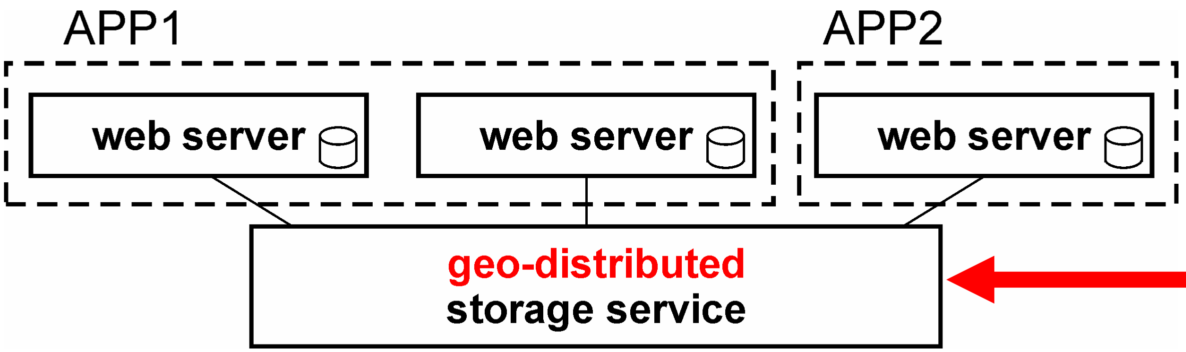
history of storage of web apps (cont'd)

- third generation (2000s): highly scalable systems
  - google file system, big table, dynamo, etc
  - start of nosql movement



history of storage of web apps (cont'd)

- fourth generation (2010-): cloud
  - infrastructure shared among apps
  - geo-distribution of storage



geo-distributed: distributed over many geographic locations

geo-distributed storage system

what

a storage system that spans many datacenters
at many geographic locations

why

scale: one datacenter too small
disaster tolerance: protect data against catastrophes
availability: keep working after intermittent problems
access locality: serve users from a nearby datacenter

geo-distribution challenges

high latency tens to thousands of milliseconds

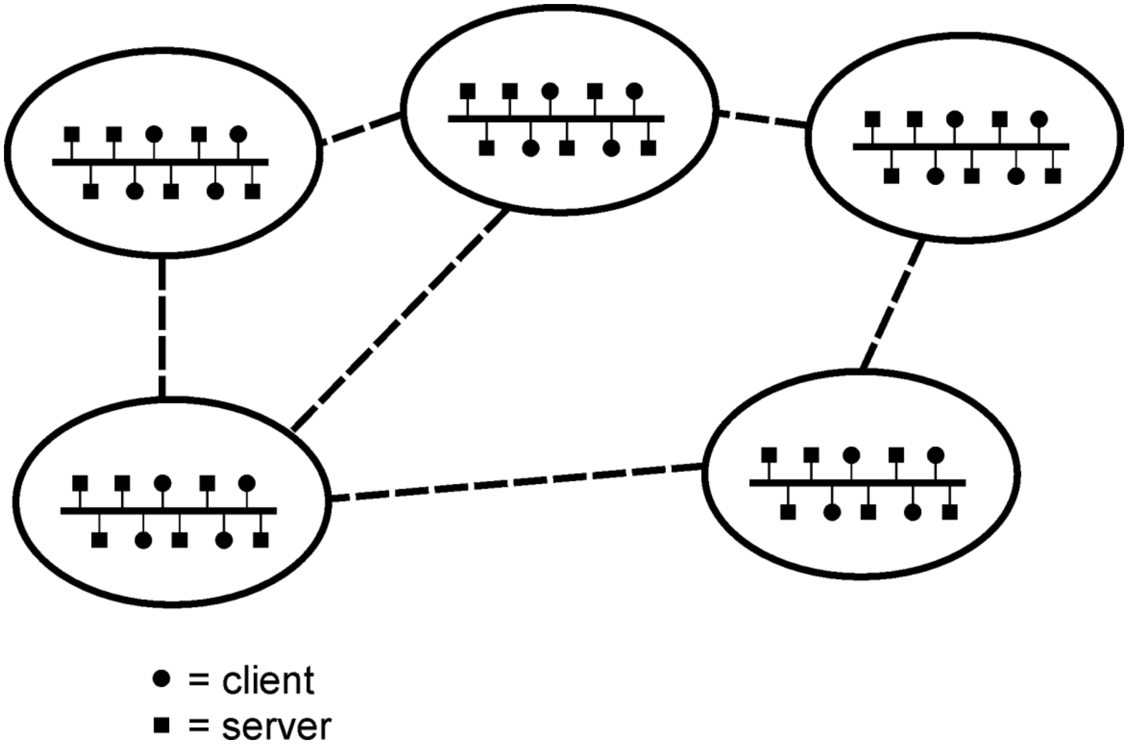
low bandwidth maybe as low as 100Mbps

congestion overprovisioning prohibitive

network partitions depending on geo-topology

legacy apps designed for one datacenter

model of geo-distributed storage



- network of networks
- two types of processes:
  - clients: wish to execute operations, run some protocol
  - servers: help implement operations (e.g., store data)

what are the client operations?

- two possibilities: read-write, state machine

1. read-write
   - two operations

     read(key) $\rightarrow$ value

     write(key,value) $\rightarrow$ ok

2. state machine
   - general operation *op*: state $\rightarrow$ state $\times$ output
   - read and write are special cases
   - more powerful operations

     atomic increment: op(x)=(x+1,ok)

     atomic swap(v): op(x)=(x,v)

topics of interest

- replicating data
- dealing with congestion
- providing transactions
- migrating data

across datacenters

geo-replication

replicating data across datacenters

goal

- maintain data copies across datacenters
  - eg, when client writes, data goes to many datacenters
- for disaster tolerance, data locality, etc

- full vs partial replication
  - full: replicas at every datacenter
  - partial: replicas at some subset of datacenters

- this talk: full replication, for simplicity

two types of replication

- synchronous replication
  - updates some/all remote replicas before completing op

- asynchronous replication
  - may update all remote replicas after completing op (“in the background”)

replication schemes

	read-write	state machine
synchronous replication	ABD	Paxos

replication schemes

	read-write	state machine
synchronous replication	ABD priority msgs	Paxos priority msgs
asynchronous replication	?	?

async read-write: last timestamp wins

- write obtains a timestamp
  - eg, from clock at local replica
  - clock need not be synchronized across replicas
    (but good if they are approximately synchronized)
- when data is copied to remote replicas:
  writes with higher timestamps obliterate
  writes with lower timestamps

async state machine: exploit commutativity

- for async replication, state machine operations
- allow only commutative update operations
- so replicas can apply them in any order and still converge

- requires commutative data types [letia,preguiça,shapiro 2009]

counting set:
example of commutative data type

- set operations are generally non-commutative
- counting set = map from id to counter
  - empty set: maps all ids to zero
- add to set: increment counter;
  remove from set: decrement counter
- negative counters ok:
  removing from empty set → set with "anti-element"
- increment and decrement always commute

- many more sophisticated data types exist

other techniques for async replication

- history merge: used in dynamo [sosp 2007]
- vector timestamps
- dependency tracking: used in cops [sosp 2011]
- history reorder

open question

what are the best abstractions for async replication?

- easy and intuitive semantics
- efficient to implement (little bookkeeping)

geo-congestion

dealing with congestion across datacenters

motivation

- bandwidth across datacenter is expensive
- storage usage subject to spikes

- how do we provision the bandwidth?
  - using peak workload: low utilization, high cost
  - using average workload: better utilization, lower cost, but congestion

- how can we live with congestion?

ways to live with congestion

- weak consistency
  - asynchronous replication
  - good performance
  - semantics undesirable in some cases
- prioritize messages

prioritized message

- bypasses congestion
- should be small

strong consistency under congestion

- new replication protocol with small prioritized msgs
- the vivace key-value storage system
  - strong consistency: linearizability
  - resilient to congestion

vivace algorithms

two algorithms:

1. read-write algorithm
   - very simple, based on ABD
2. state machine algorithm
   - more complex, based on Paxos

THIS TALK

basic idea of vivace read-write protocol

- separate data and timestamp
- replicate data locally first
- replicate timestamp remotely with prioritized msg
- replicate data remotely in background

details

- to write
  - obtain timestamp ts
  - write data,ts to f+1 temporary local replicas (big msg)
  - write only ts to f+1 real replicas (small msg, cross datacenter)
  - in background, send data to real replicas (big msg)
- to read
  - read ts from f+1 replicas (small msg)
  - read data associated with ts from 1 replica (big msg, often local)
  - write only ts to f+1 real replicas (small msg, cross datacenter)

stepping back

- separate data and metadata, prioritize metadata
- when possible,
  - use local replicas in foreground
  - use remote replicas in background

- same ideas can be applied to paxos

more general open question

- what can we solve efficiently with this combination of message types?

	local	remote
small	fast	fast
large	fast	slow

geo-transactions

providing transactions across data centers

why transactions in storage system?

- help dealing with hard problems arising from concurrency+failures

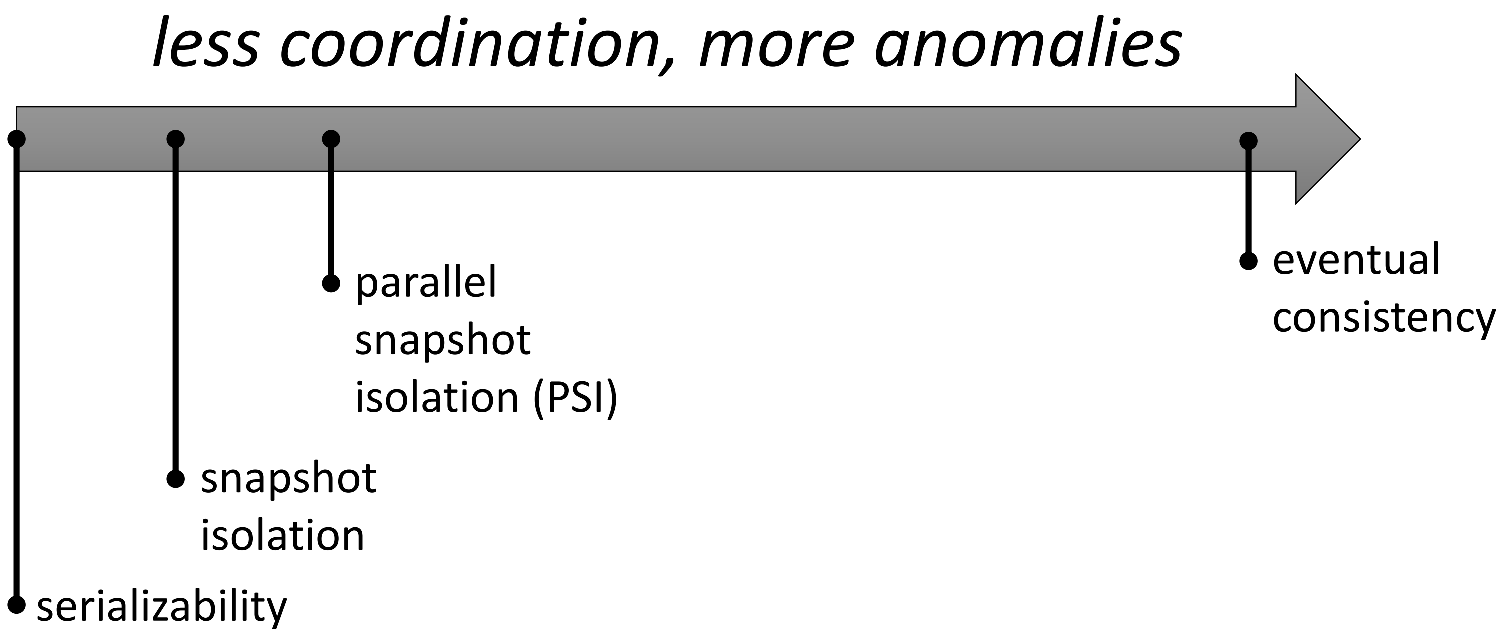
- transfer hard problems from application to storage infrastructure
  - fewer storage systems than applications
  - infrastructure developers have better expertise

why transactions, part two: life without transactions

- issue of integrity of the storage state
  - dangling references
  - orphan objects
  - unexpected reference cycles
  - garbage
- resulting in
  - code complexity
  - lack of productivity
  - loss of software quality

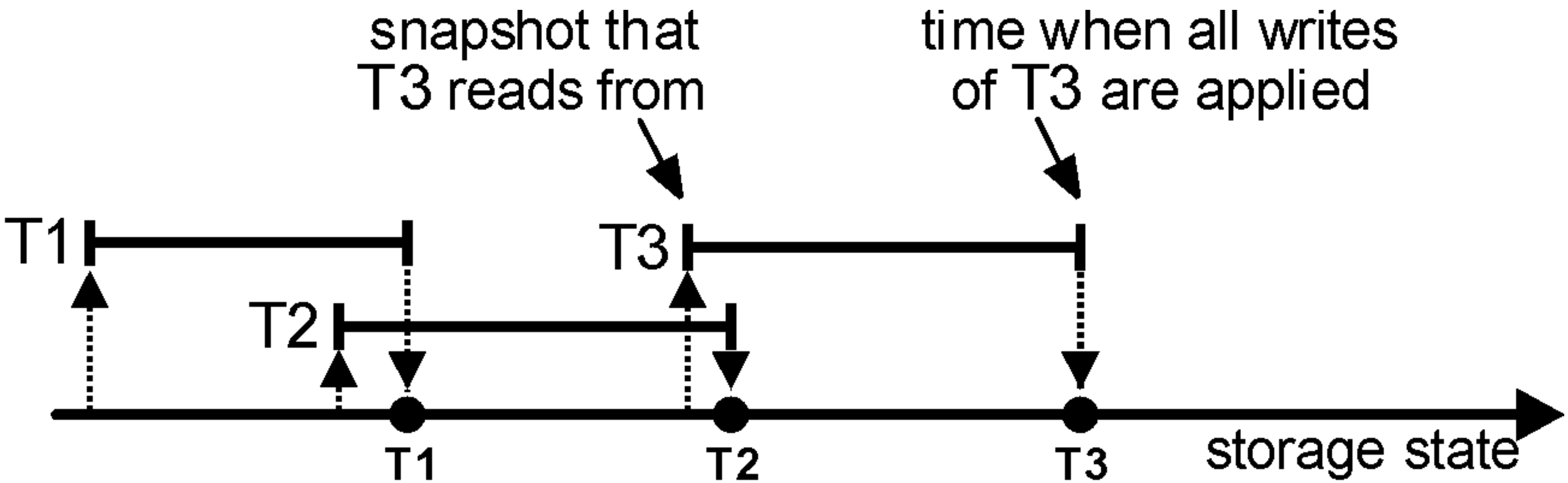
- our goal in providing transactions:
  facilitate job of developer
  without sacrificing performance

transaction coordination and anomalies



snapshot isolation

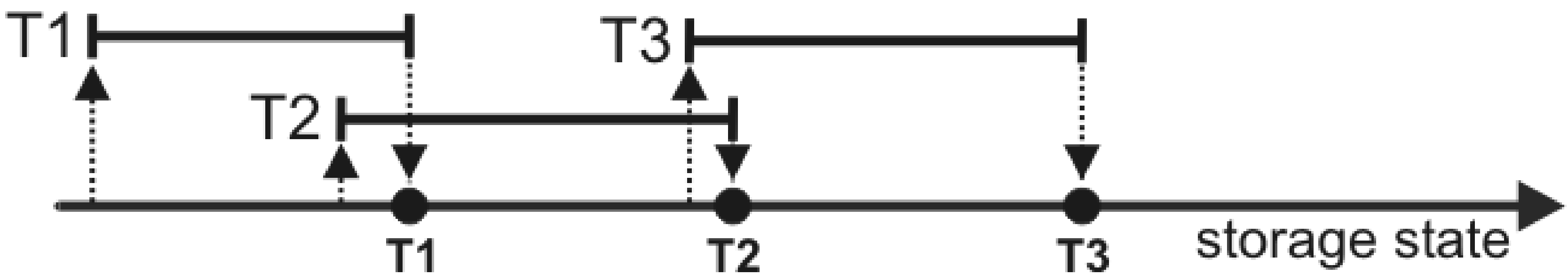
- supported by commercial DB systems



- properties
  - reads performed on a consistent snapshot
  - writes concentrated at commit time
  - no write-write conflicts

issue with snapshot isolation

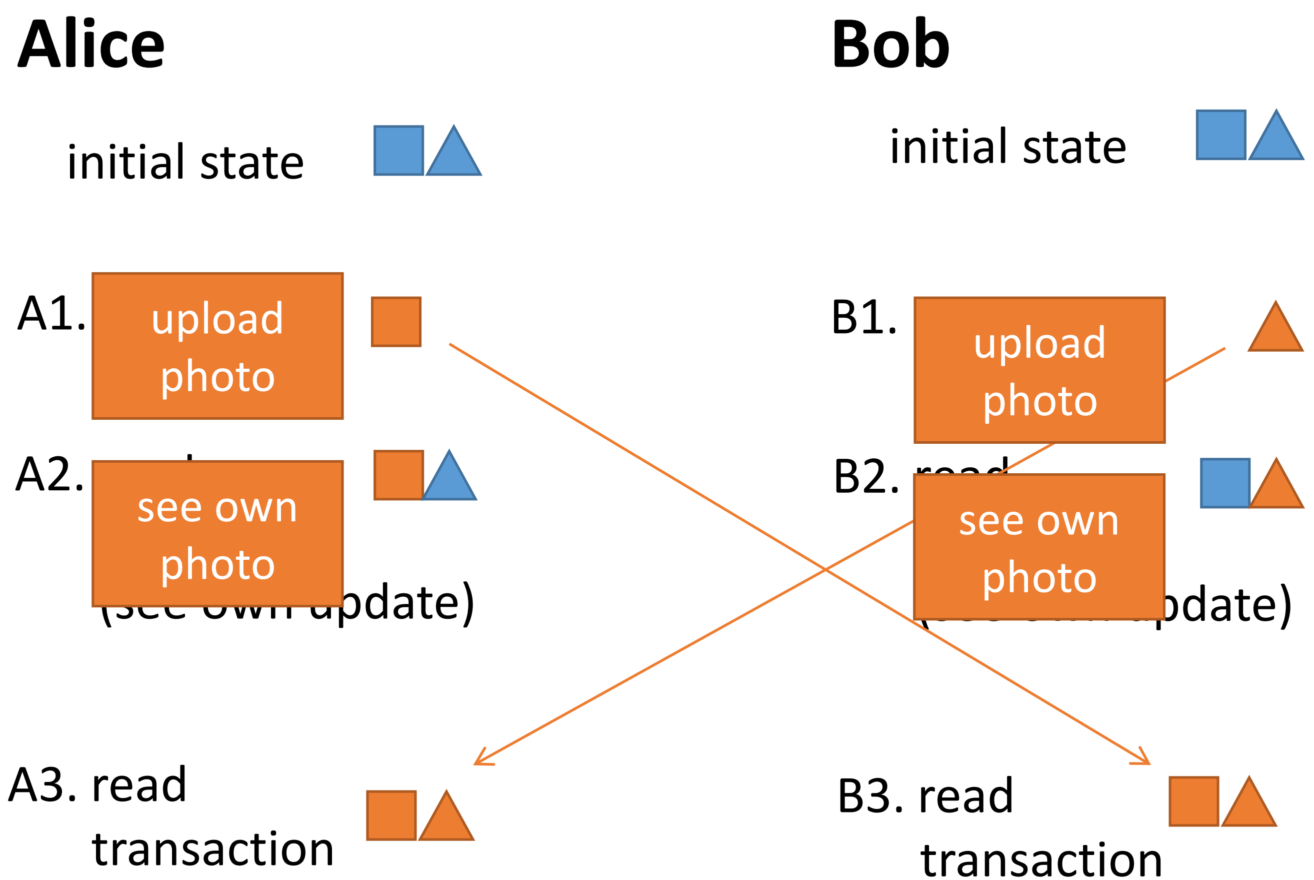
- it orders the commit time of all transactions
  even those that do not conflict with each other



- forbids some scenarios we want to allow for efficiency

issue with snapshot isolation (cont’d)

- scenario forbidden by snapshot isolation
  (it requires total ordering of update transactions)



parallel snapshot isolation (PSI)

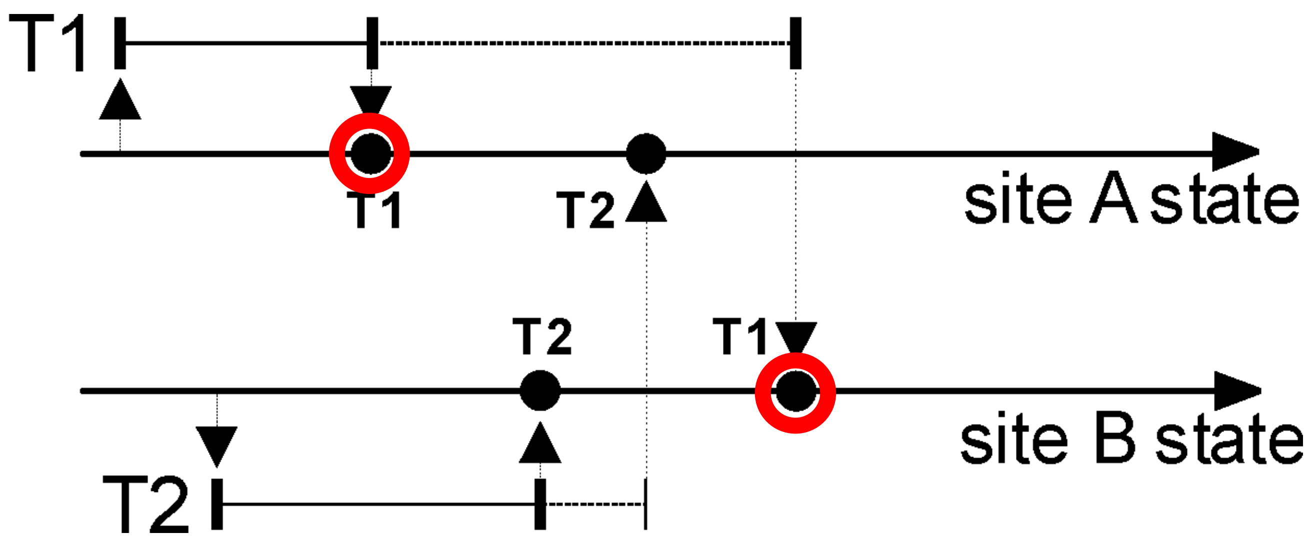
snapshot isolation

- one commit time
- one timeline
- read from snapshot
- no write-write conflicts

PSI

- a commit time per site
- a timeline per site
- read from snapshot at site
- ditto
- causality property

parallel snapshot isolation (PSI)



features

(1) commit time per site, (2) timeline per site,
(3) read from snapshot at site, (4) no write-write conflicts
(5) causality:
if *T1* commits at site *S* before *T2* starts at site *S* then *T2* does not commit before *T1* at any site

implementing PSI efficiently

- PSI prohibits write-write conflicts (even across sites)
- issue: how to enforce it efficiently?
  (without coordination across sites)
- two ideas
  - preferred sites: optimize updates at certain sites
  - commutative ops: eliminate conflicts

idea #1: preferred sites

- each object assigned a preferred site

- at that site, object can be written efficiently

  transactions that modify objects at their preferred site
  can use *fast* commit (without cross-site communication)

- inspired by notion of a primary site

  but less restrictive, because objects modifiable at any site

- example of usage: web application

  preferred site of objects of a user =
  site close to where user usually logs in from

potential performance issue

- what if many sites often modify an object?
- no good way to assign a preferred site to object
- bad idea: keep changing preferred sites of object
  - requires cross site communication
  - defeats efficiency goal

idea #2: commutative ops

- goal: eliminate conflicts

- when clients update the same object concurrently, most likely they do not want to overwrite it
  - otherwise we have a race condition

- cast concurrent updates as commutative ops
  - example: users B and C adds themselves to A’s friends list

putting both ideas together

- walter system [sosp 2011]
- certain transactions: fast execution and commit

  = without cross site communication

  - if preferred site of written objects is local *or*
  - updates done on counting sets objects

- other transactions: slow commit

  = with cross site communication

open questions

- what is strongest isolation level that allows local commits?
- are there fundamental trade-offs between isolation and performance?

efficient serializable transactions

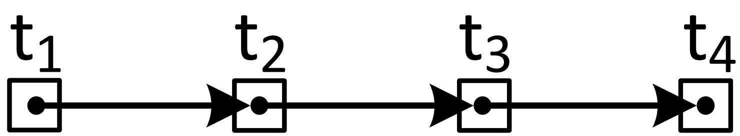
- PSI and snapshot isolation have anomalies
- can we get serializability efficiently?

- answer: sometimes, using *transaction chains [sosp 2013]*

transaction chains

definition

- sequence of *local* transactions (*hops*), each hop executing at one server



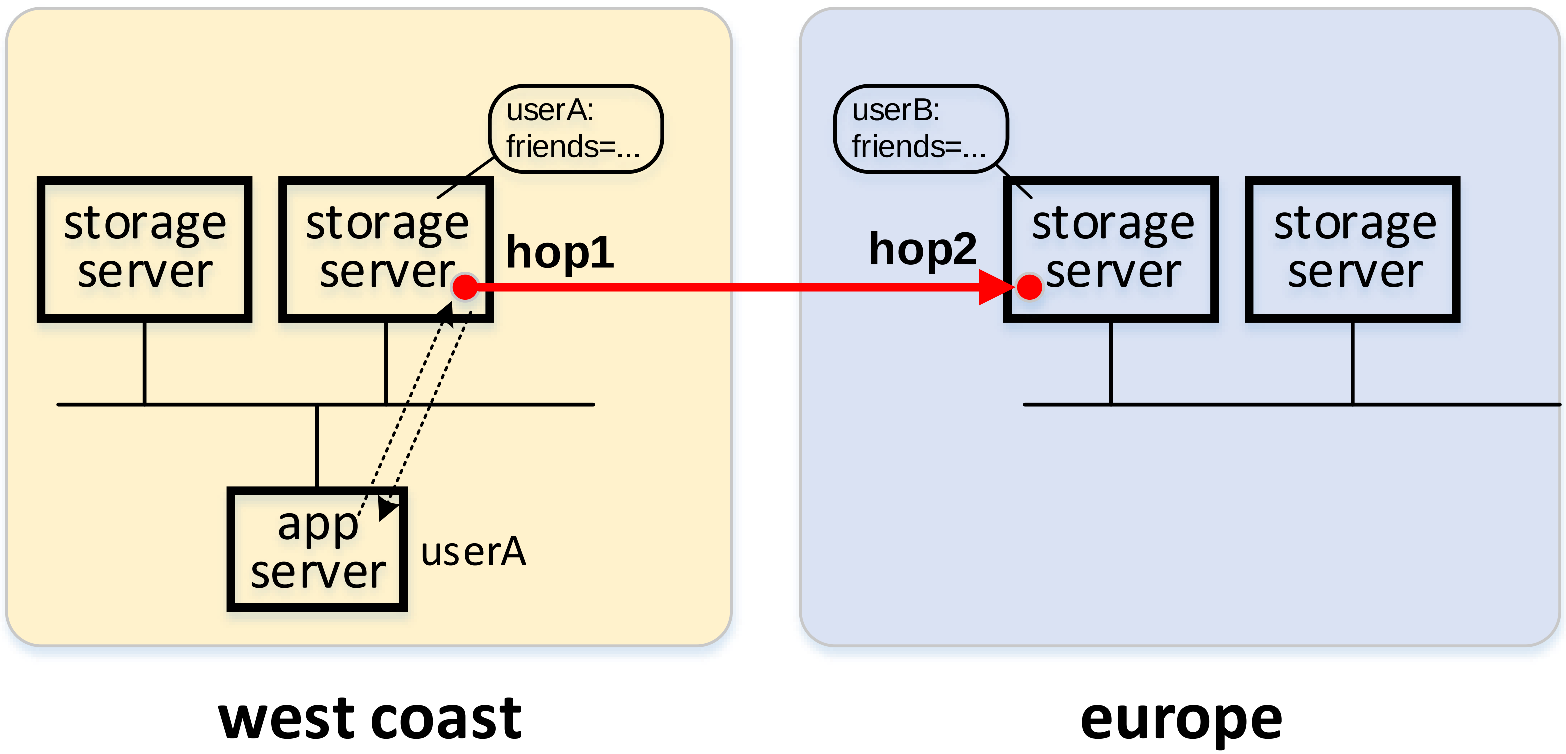
• = local transaction

□ = server

- hops must be known at beginning
- a hop can take as input data from earlier hops

a simple chain

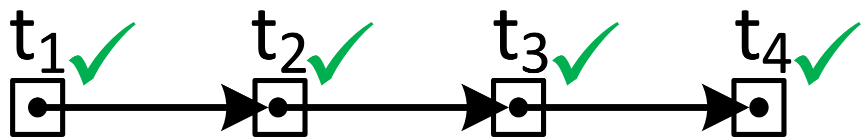
- to implement the “befriend” operation in a social network



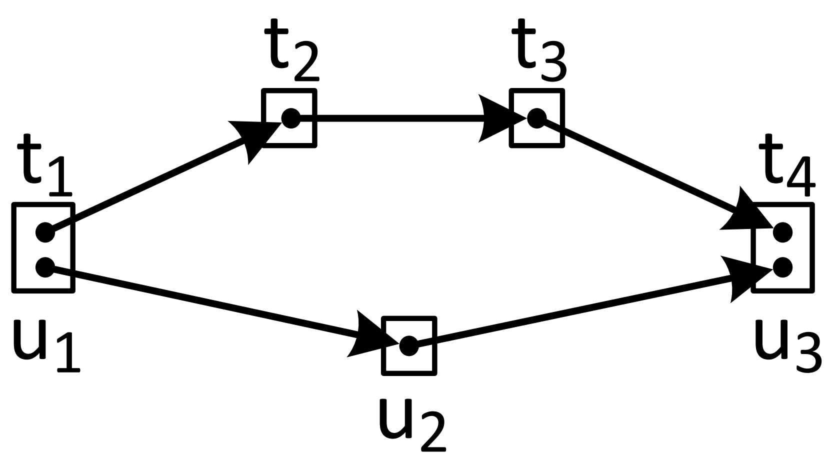
- hop1: add B as friend of A, hop2: add A as friend of B
- control returns to app *quickly* after hop1;
  hop2 runs in the background

properties of chains

(atomicity) all hops commit or none of them do



(origin ordering) if two chains start at same server
then they execute in the same order
at every server they intersect



properties of chains

(serializability) chains are serializable as transactions

execution order: t_1 u_1 t_2 u_2

must be equivalent to t_1 t_2 u_1 u_2

or u_1 u_2 t_1 t_2

chains can execute in two ways

- piecewise
  - one hop at a time, in series
  - the intended way

- all at once
  - all hops in one distributed transaction
    - standard techniques: two-phase locking, two-phase commit
  - only if piecewise execution might violate serializability

can chains execute piecewise?

theory of transaction chopping [shasha et al'95]

- when can tx be chopped into pieces?
- assume transactions known a priori
- construct sc-graph
- no sc-cycle $\Rightarrow$ chopping ok

the sc-graph

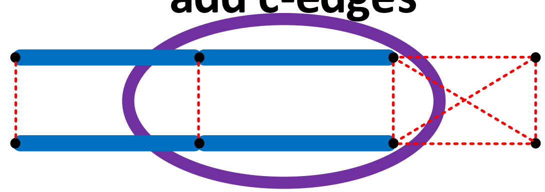
- undirected graph
- two types of edges: s-edges and c-edges
- chains included as vertices connected by **s-edges**
- there are two copies of each chain
- **c-edges** between conflicting hops of different chains
  - both hops modify the same table and one hop is an update
- sc-cycle = cycle with an s-edge and a c-edge

example of an sc-graph

chain 1 **chain 2**

two copies of chains

add c-edges



an sc-cycle

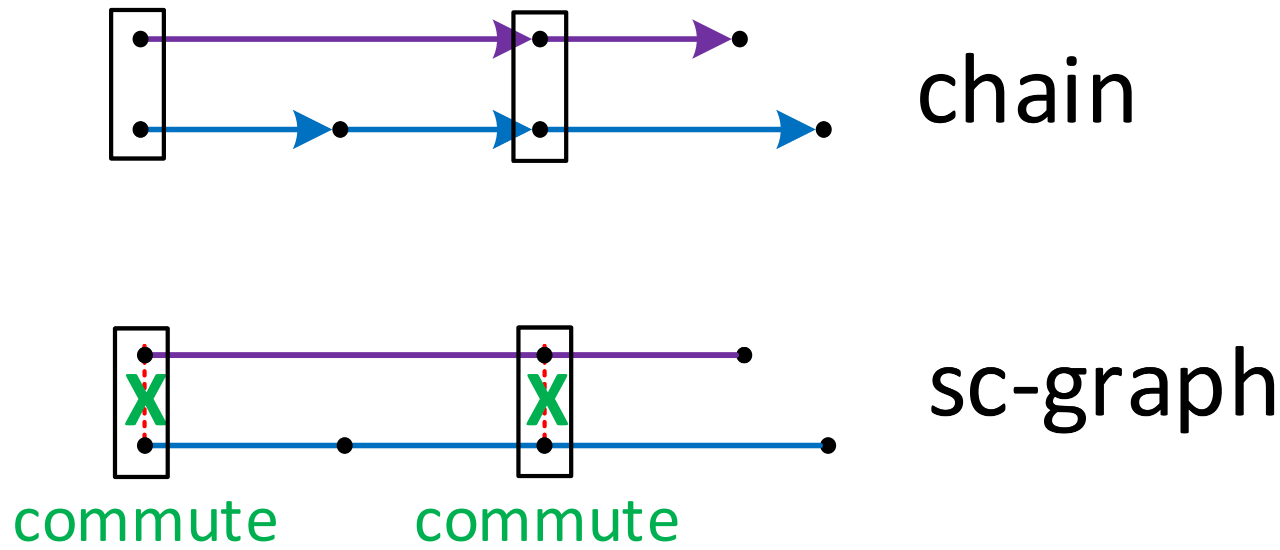
problem: sc-cycles everywhere!

eg, many edges between two copies of same chain

solution: eliminate edges from sc-graph

1. annotations of commutative pairs of hops
2. derived chains: rely on origin ordering
3. break chain into *linked chains*

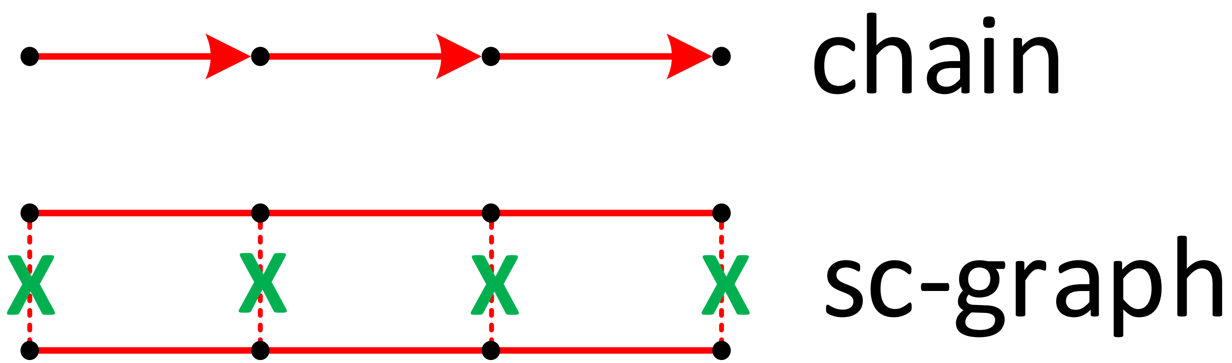
1. annotations of commutative pairs of hops



remove C edges between two hops
if user annotation says pair commutes

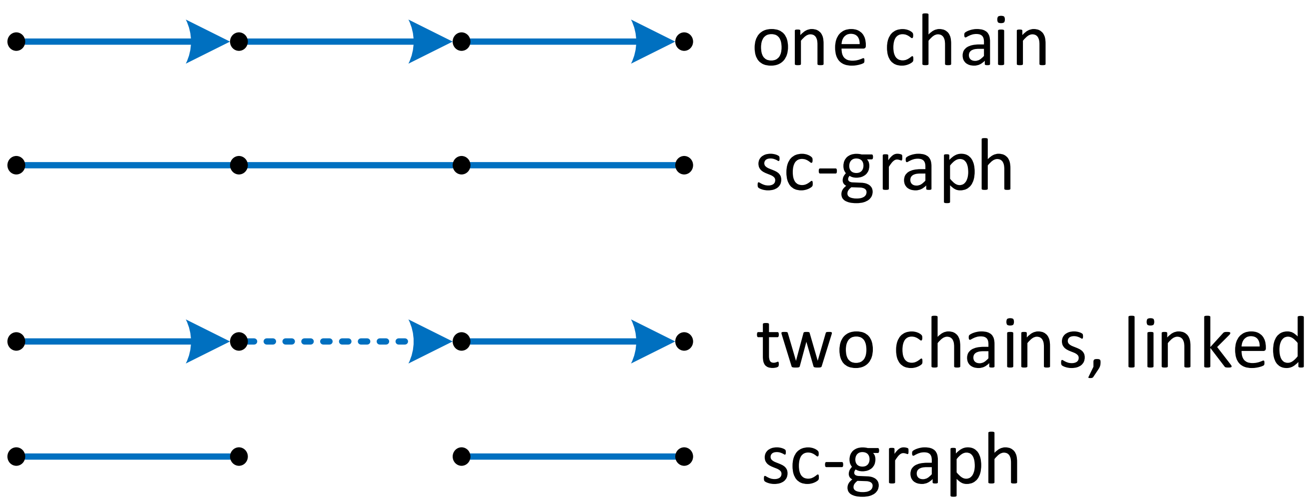
often avoids sc-cycles over two copies of same user chain

2. derived chains: rely on origin ordering



- origin ordering serializes instances of same derived chain
- can collapse derived chain to a node in SC graph

3. break chain into *linked chains*



- linked chains: chain of chains
- atomicity but not serializability across chains

implementing chains: the lynx system

1. client library dispatches to first hop
2. first hop acts as coordinator
3. coordinator assigns cid (chain-id) and sequencers, and logs chain before execution
4. for each hop
    - dispatch to appropriate server
    - server waits for appropriate sequencer
    - server executes a local transaction that
        - checks cid not in history table
        - if not then execute the hop
            - add cid to history table
    - coordinator logs hop as executed
    - if first hop, inform client of first-hop completion
5. inform client of chain completion

how lynx ensures chain properties

- atomicity
  - log chain in a replicated storage system
  - avoid repeating hops on recovery
    - at each server, history table tracks which chains executed
    - checked and updated in local transaction of hop
- origin ordering
  - sequencer*[i,j]* = how many txs starting at server *i* server *j* should wait for
- serializability
  - from sc-graph analysis

related work

- sagas, step-decomposed transactions
- view maintenance schemes

stepping back: broader question

- what are the right abstractions for concurrent programming in a geo-distributed setting?

geo-migration

migrating data across datacenters

why geo-migration

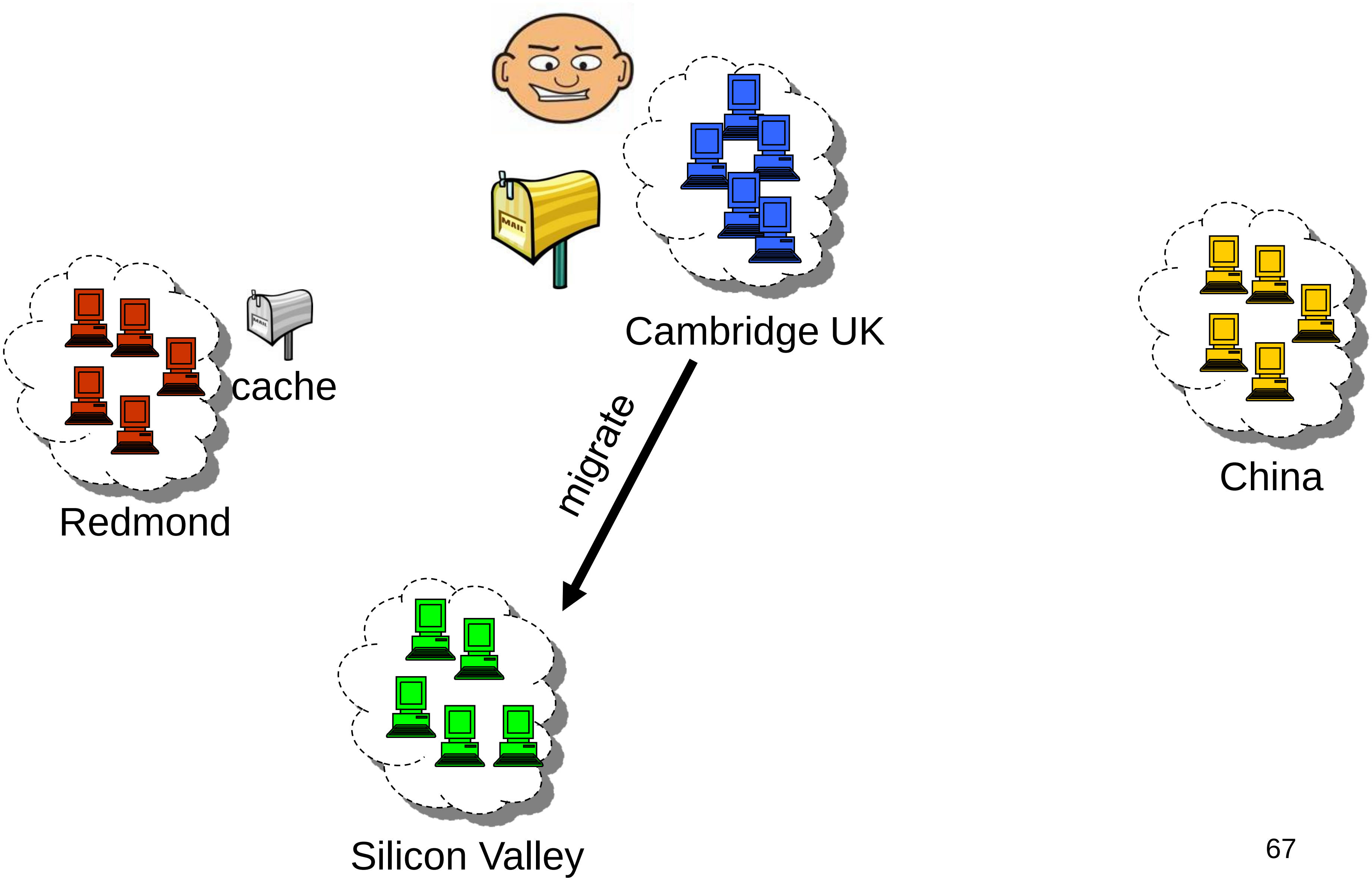
- best site for data may change
  - users relocate
  - workloads change
  - infrastructure changes: new sites, new network links

- separation
  - mechanisms for geo-migration: how
  - policies for geo-migration: when and where

desirable properties
for migration mechanism

- works online: data available during migration
- supports cancellation of migration or change of target
- works well with caching and replication

sample use case



distributed data overlays

(Nomad storage system)

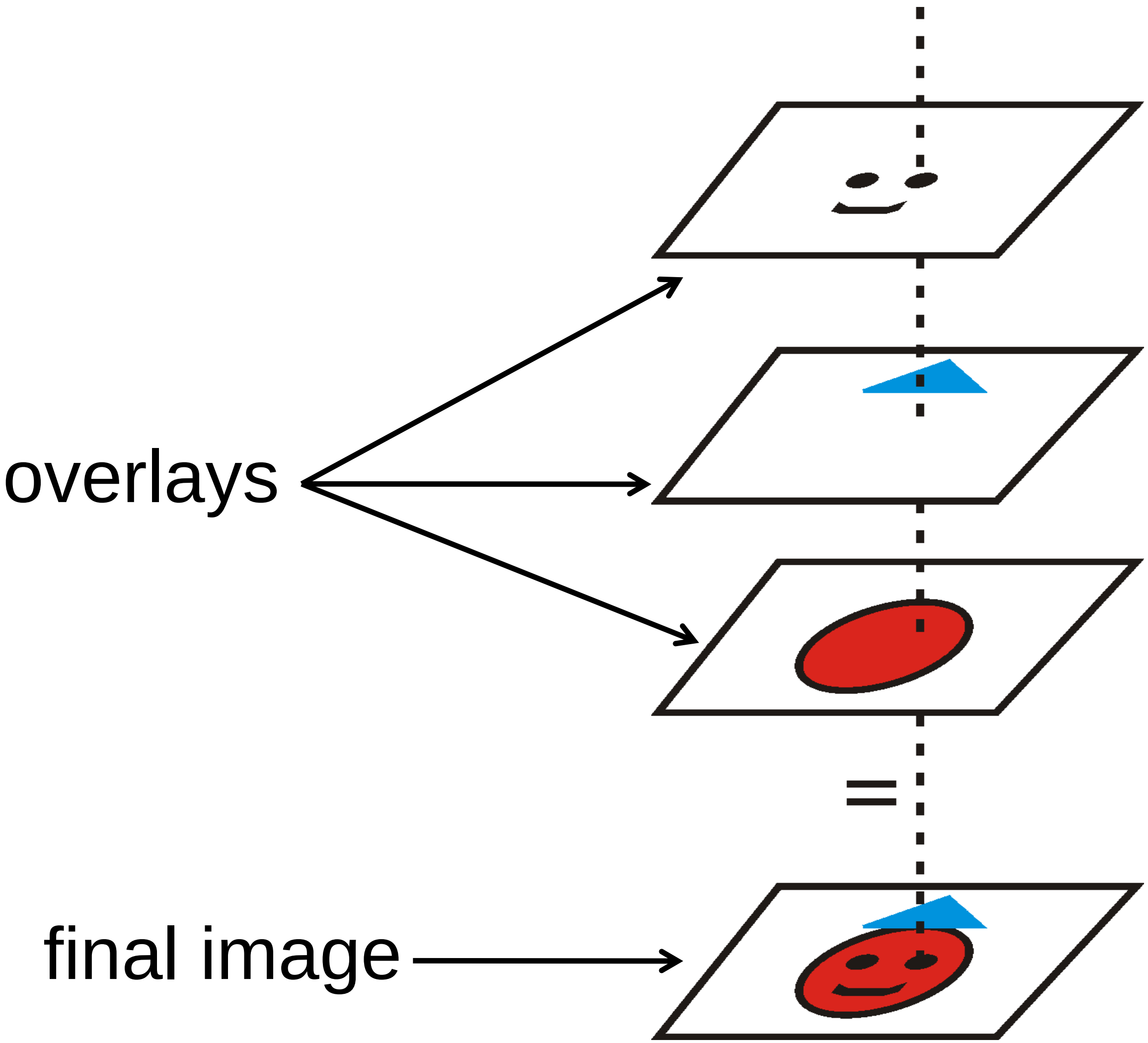
data is accessible at all times

migration benefit is realized quickly

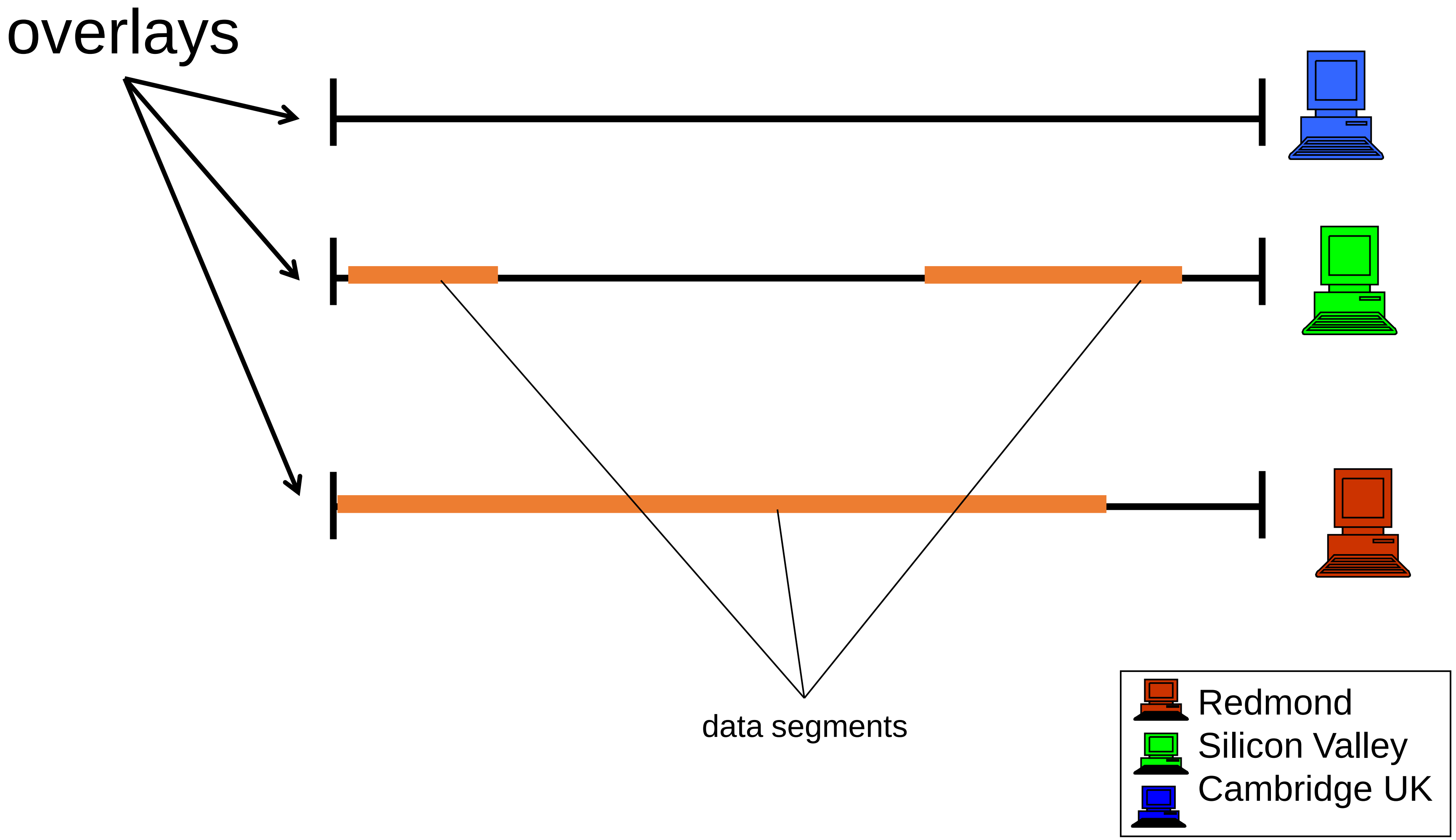
- writes go to new site instantly
- reads are served at new site as soon as possible
- intuition: read first at new site
  and redirect if data not there

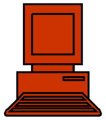
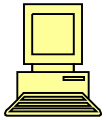
seamlessly support caching and replication

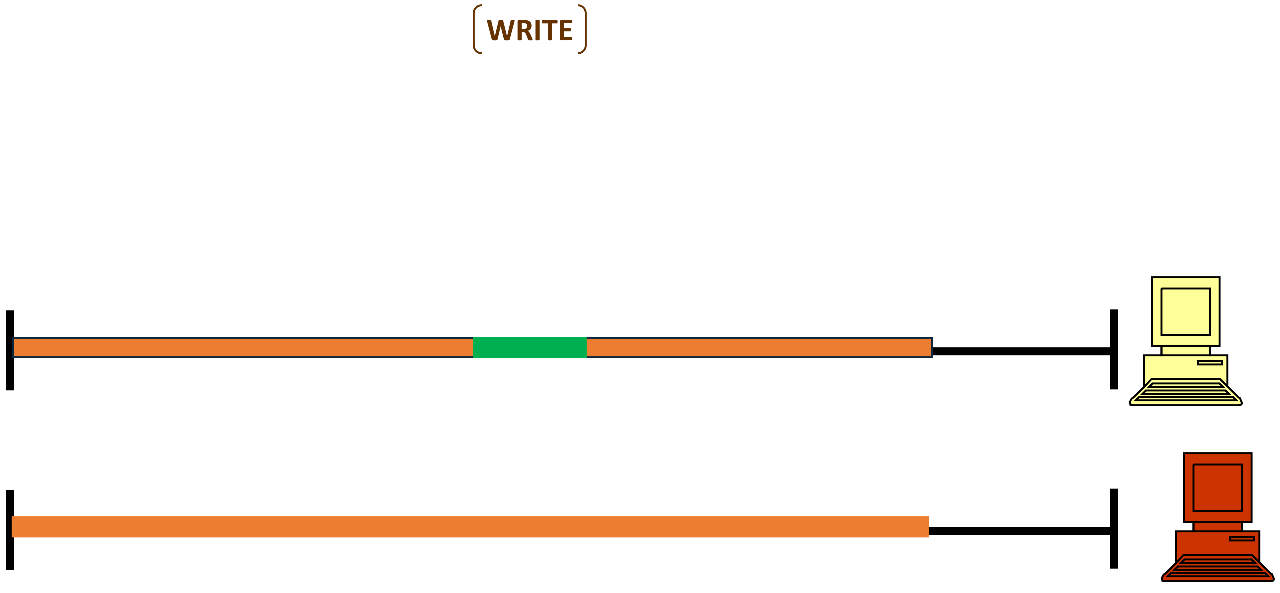
overlay: a simple abstraction



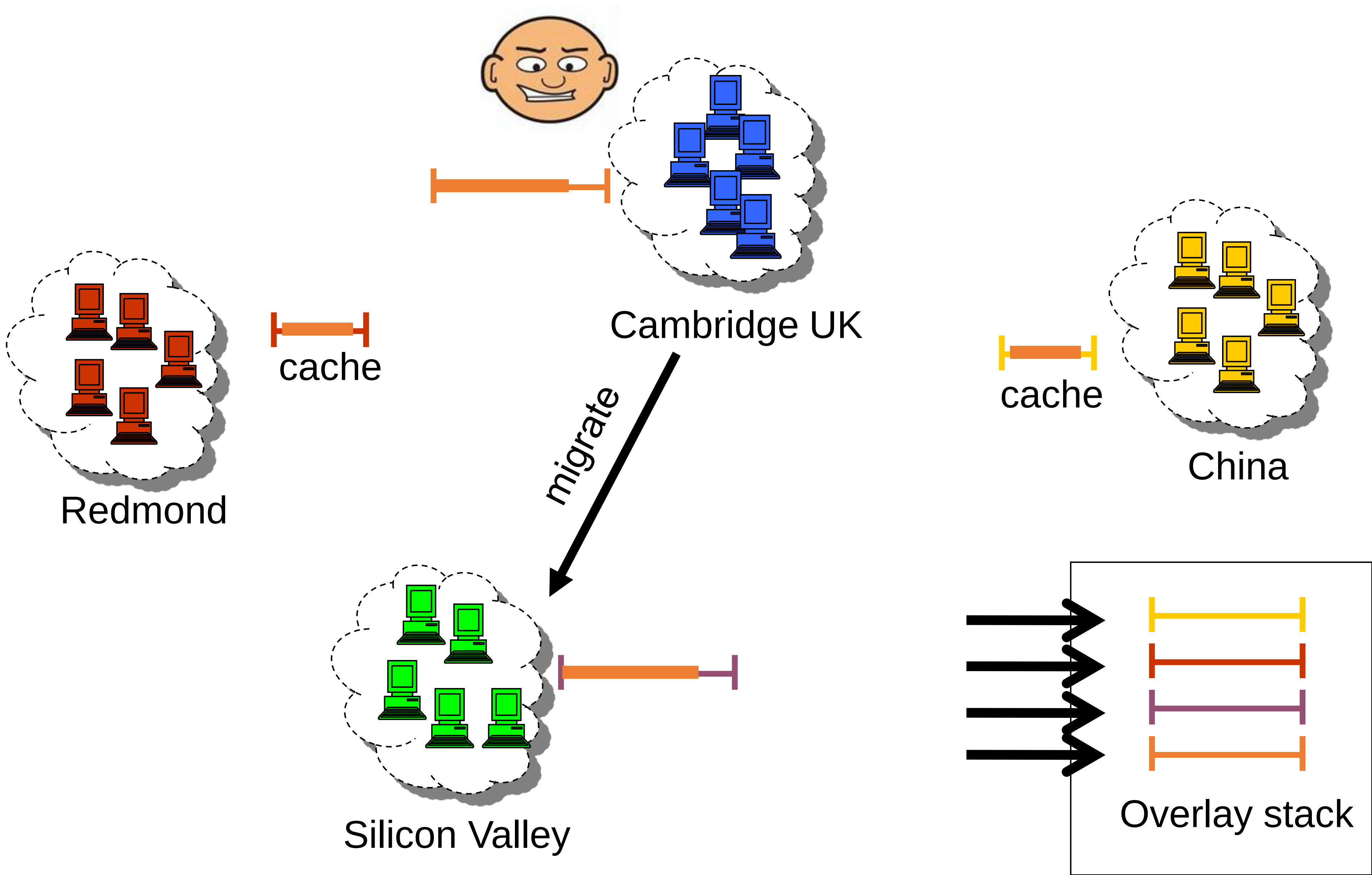
overlay stack structure for an object



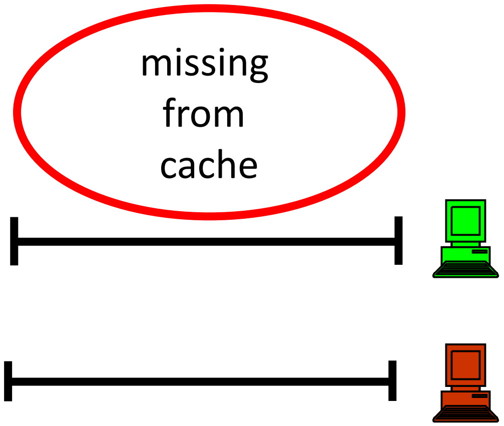
migrating from  to 



using overlays

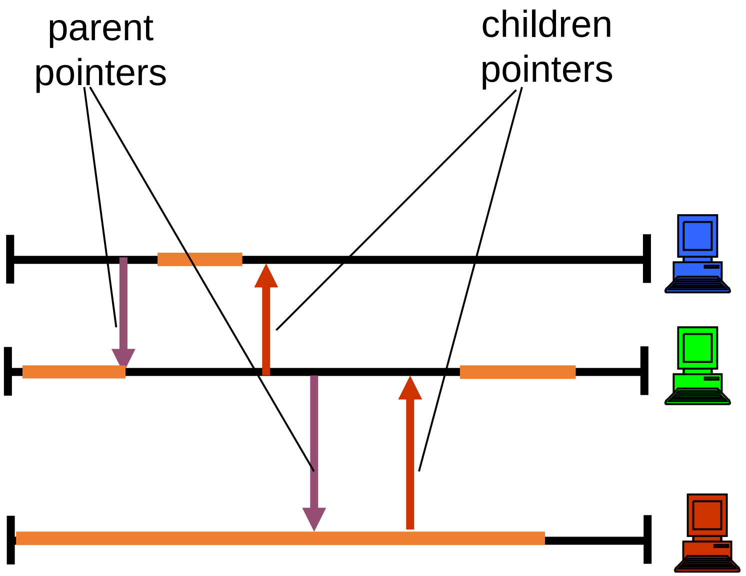


overlay implementation



client side

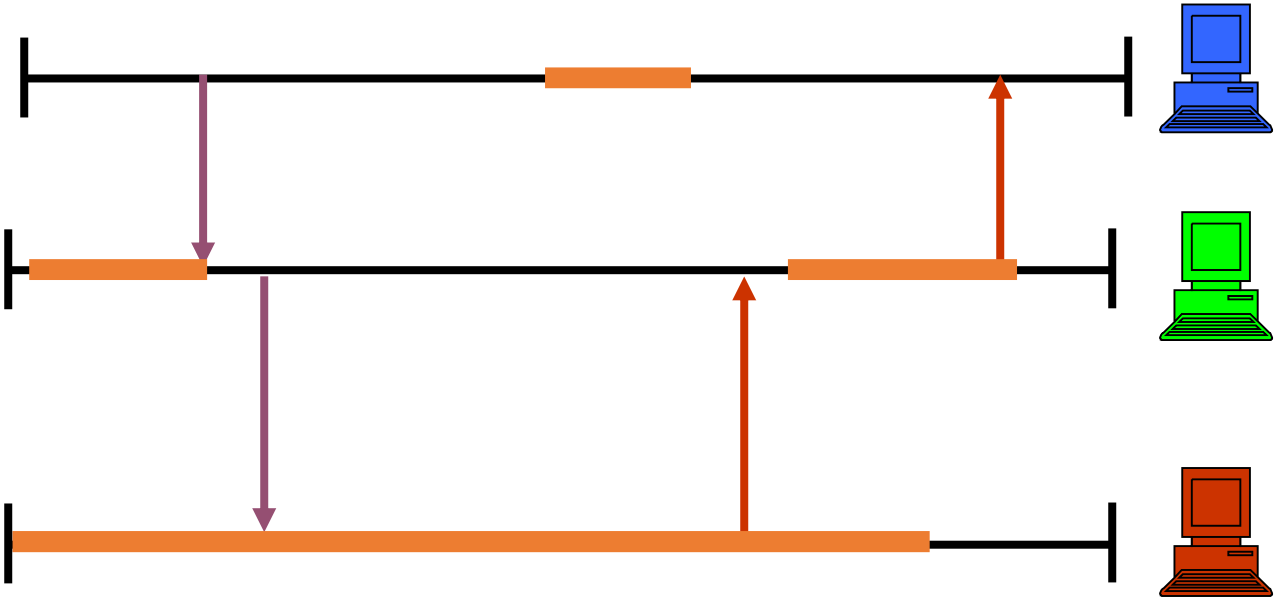
cache the overlay
stack structure



server side

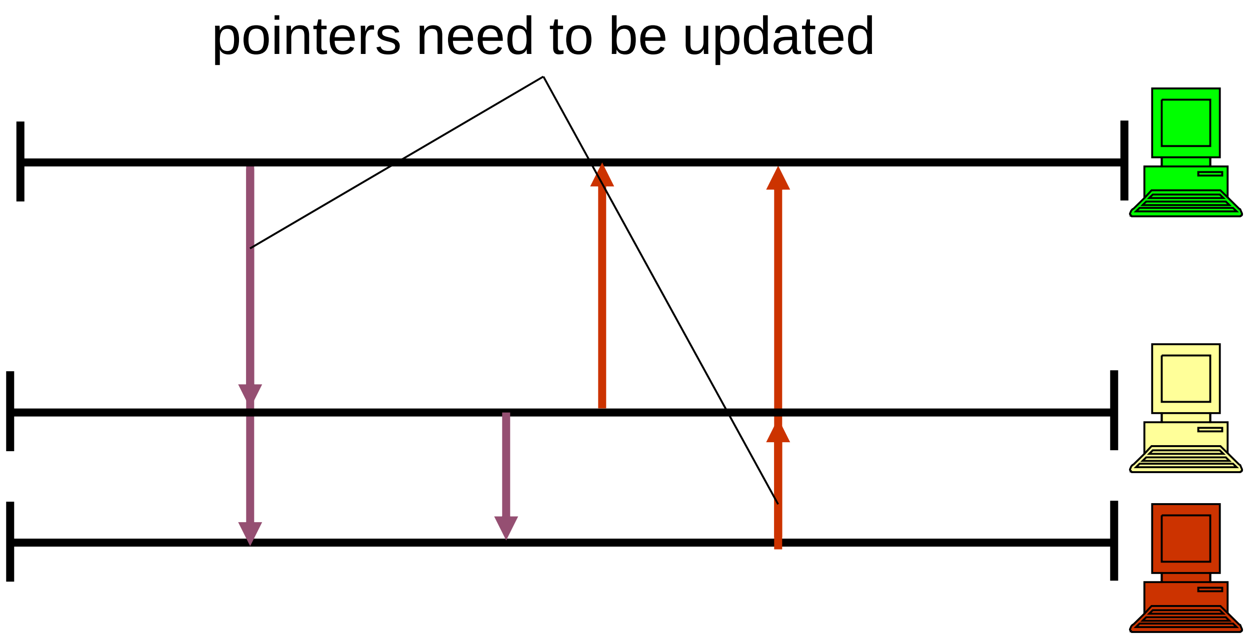
at each overlay, maintain local pointers
to the above and below overlays

local pointers are used to redirect R/W



WRITE

update pointers in CREATE operation



challenges: pointers are at different machines.
do we need 2PC? answer: NO

update order: create pointers at the new overlay before pointers at its parent, before pointer at its child

acknowledgements

- Mahesh Balakrishnan
- Brian Cho
- Jinyang Li
- Russell Power
- Yair Sovran
- Nguyen Tran
- Yang Zhang
- Siyuan Zhou

conclusion

- geo-distributed datacenters: an exciting setting
  - large latency, low bandwidth across datacenters
  - apps require fast response
  - network may partition
- raises new practical and conceptual problems
- related to many areas
  - distributed computing
  - distributed systems
  - networking
  - database systems