Spanner

Doug Woos
(based on slides by Dan Ports)

Bigtable in retrospect

- Definitely a useful, scalable system!
- Still in use at Google, motivated lots of NoSQL DBs
- Biggest mistake in design (per Jeff Dean, Google): not supporting distributed transactions!
  - became really important w/ incremental updates
  - users wanted them, implemented themselves, often incorrectly!
  - at least 3 papers later fixed this — two next week!

Transactions

- Important concept for simplifying reasoning about complex actions
- Goal: group a set of individual operations (reads and writes) into an atomic unit
  - e.g., checking_balance -= 100, savings_balance += 100
- Don’t want to see one without the others
  - even if the system crashes (atomicity/durability)
  - even if other transactions are running concurrently (isolation)

Traditional transactions

- as found in a single-node database
- atomicity/durability: write-ahead logging
  - write each operation into a log on disk
  - write a commit record that makes all ops commit
  - only tell client op is done after commit record written
  - after a crash, scan log and redo any transaction with a commit record; undo any without

Traditional transactions

- isolation: concurrency control
  - simplest option: only run one transaction at a time!
  - standard (better) option: two-phase locking
    - keep a lock per object / DB row,
      usually single-writer / multi-reader
    - when reading or writing, acquire lock
    - hold all locks until after commit, then release

Transactions are hard

- definitely oversimplifying: see a database textbook on how to get the single-node case right
- …but let's jump to an even harder problem:
distributed transactions!
- What makes distributed transactions hard?
    - savings_bal and checking_bal might be stored on different nodes
    - they might each be replicated or cached
    - need to coordinate the ordering of operations across copies of data too!

Correctness for isolation

- usual definition: serializability
  each transaction's reads and writes are consistent with running them in a serial order, *one transaction at a time*

- sometimes: strict serializability = linearizability
  same definition + real time component

- two-phase locking on a single-node system provides strict serializability!

Weaker isolation?

- we had weaker levels of consistency:
  causal consistency, eventual consistency, etc
- we can also have weaker levels of *isolation*
- these allow various anomalies:
  behavior not consistent with executing serially
- snapshot isolation, repeatable read,
  read committed, etc

Weak isolation vs weak consistency

- at strong consistency levels, these are the same: serializability, linearizability/strict serializability

- weaker isolation: operations aren't necessarily atomic
  A: savings -= 100 checking += 100
  B: read savings, checking
  but all agree on what sequence of events occurred!

- weaker consistency: operations are atomic, but different clients might see different order
  A sees: s -= 100; c += 100; read s,c
  B sees: read s,c; s -= 100; c += 100

Two-phase commit

- model: DB partitioned over different hosts, still only one copy of each data item; one coordinator per transaction
- during execution: use two-phase locking as before; acquire locks on all data read/written
- to commit, coordinator first sends prepare message to all shards; they respond prepare_ok or abort
  - if prepare_ok, they *must* be able to commit transaction later; past last chance to abort.
  - Usually requires writing to durable log.
- if all prepare_ok, coordinator sends commit to all; they write commit record and release locks

Is this the end of the story?

- Availability: what do we do if either some shard or the coordinator fails?
  - generally: 2PC is a blocking protocol, can't make progress until it comes back up
  - some protocols to handle specific situations, e.g., coordinator recovery
- Performance: can we really afford to take locks and hold them for the entire commit process?

Spanner

- Backend for the F1 database, which runs the ad system
- Basic model: 2PC over Paxos
- Uses physical clocks for performance

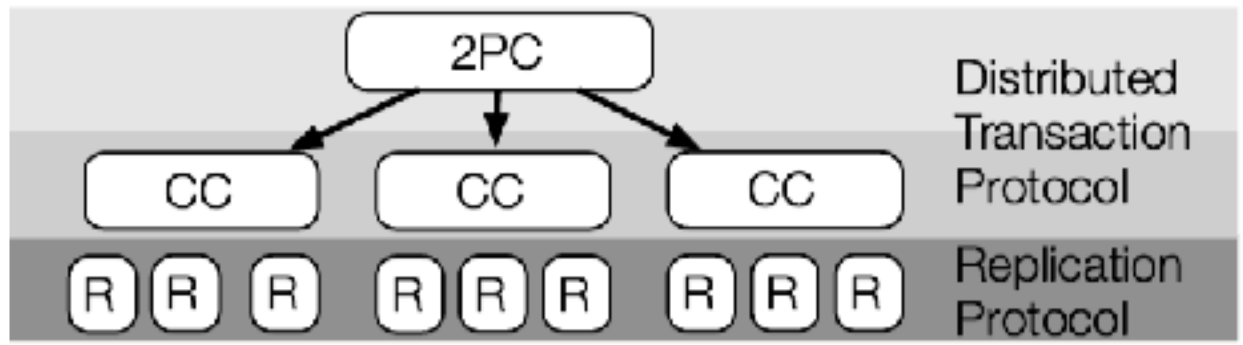
Example: social network

- simple schema: user posts, and friends lists
- but sharded across thousands of machines
- each replicated across multiple continents

Example: social network

- example: generate page of friends' recent posts
- what if I remove friend X, post mean comment?
  - maybe he sees old version of friends list,
    new version of my posts?
- How can we solve this with locking?
  - acquire read locks on friends list, and on each friend's posts
  - prevents them from being modified concurrently
  - but potentially really slow?

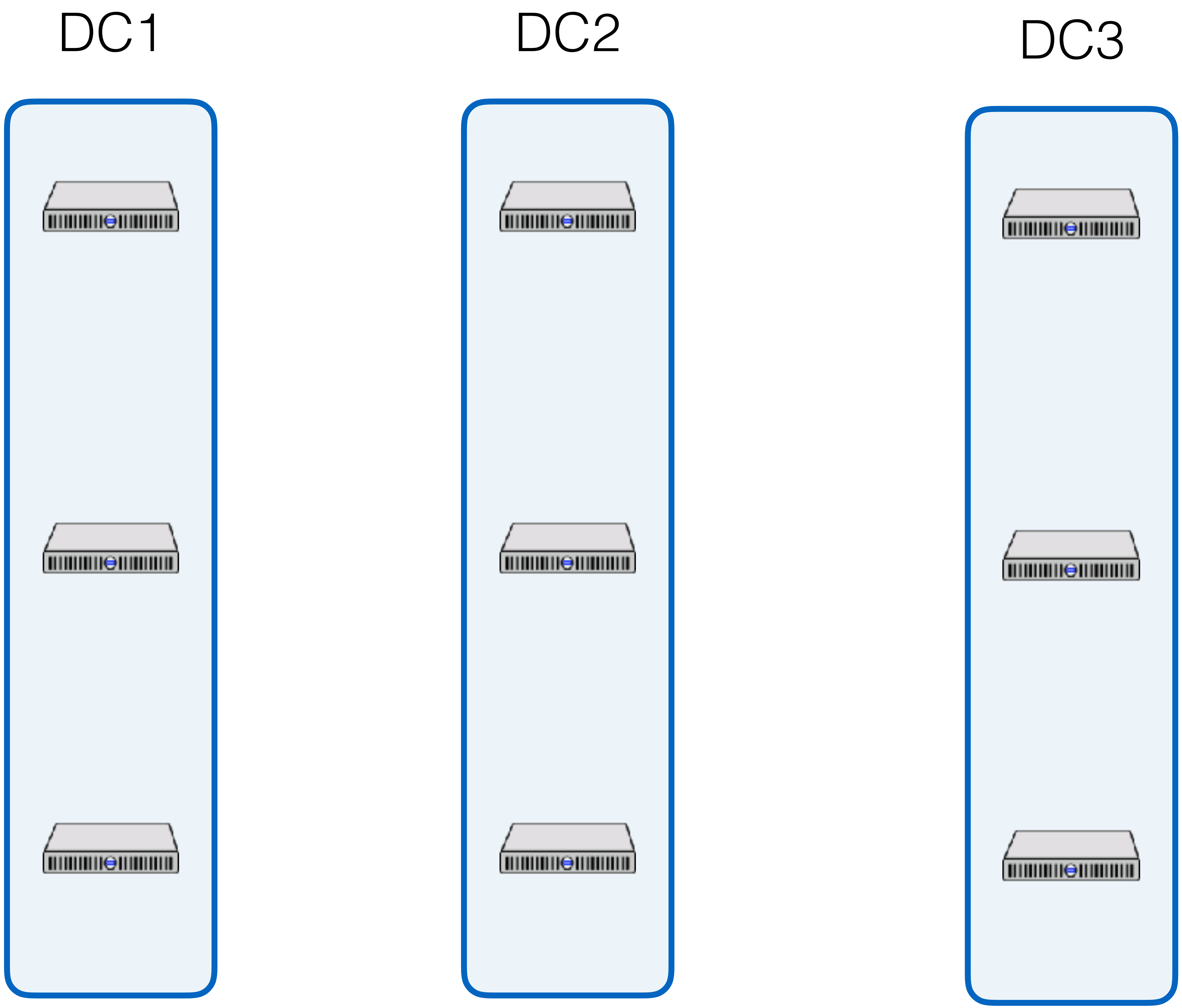
Spanner architecture

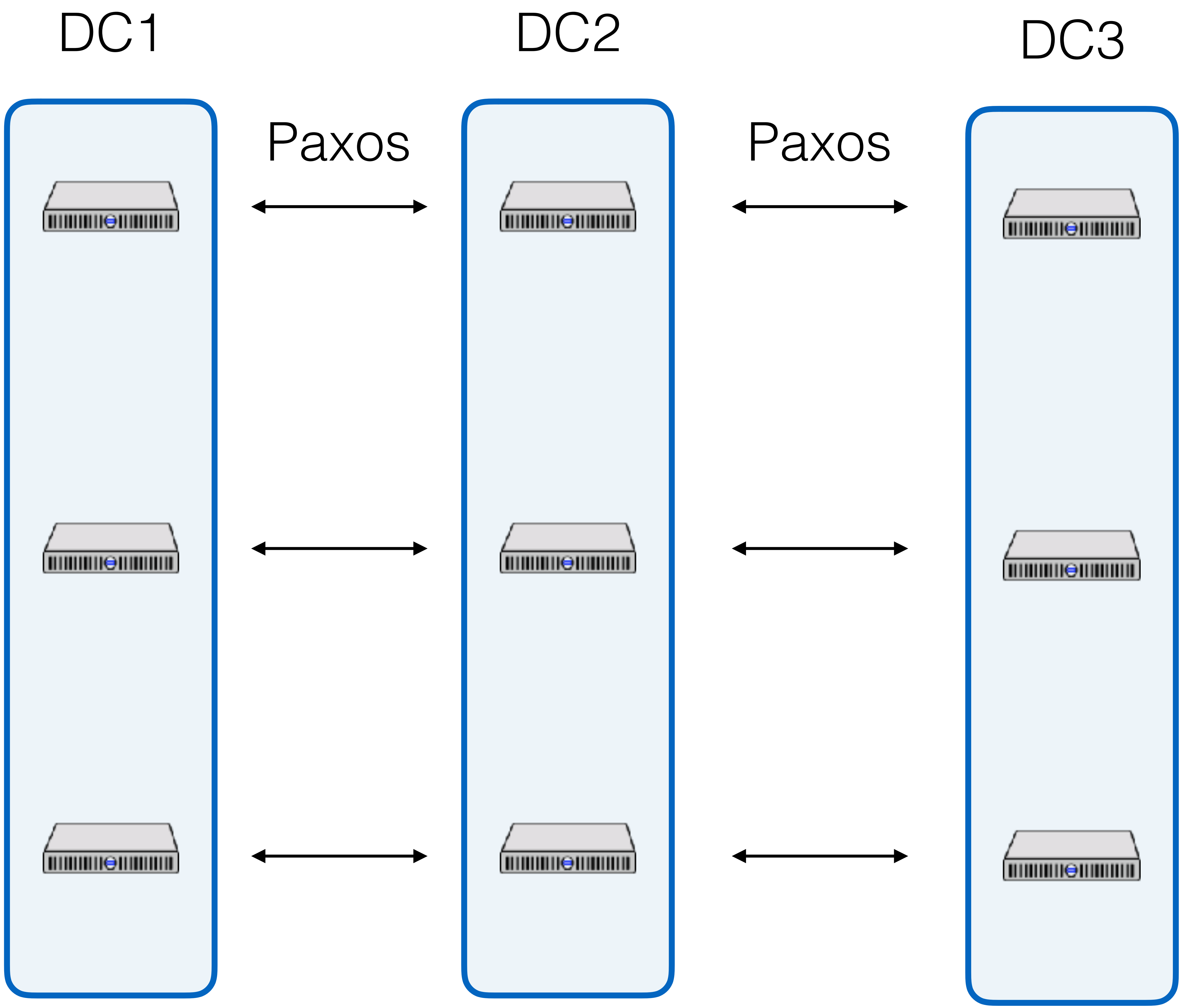


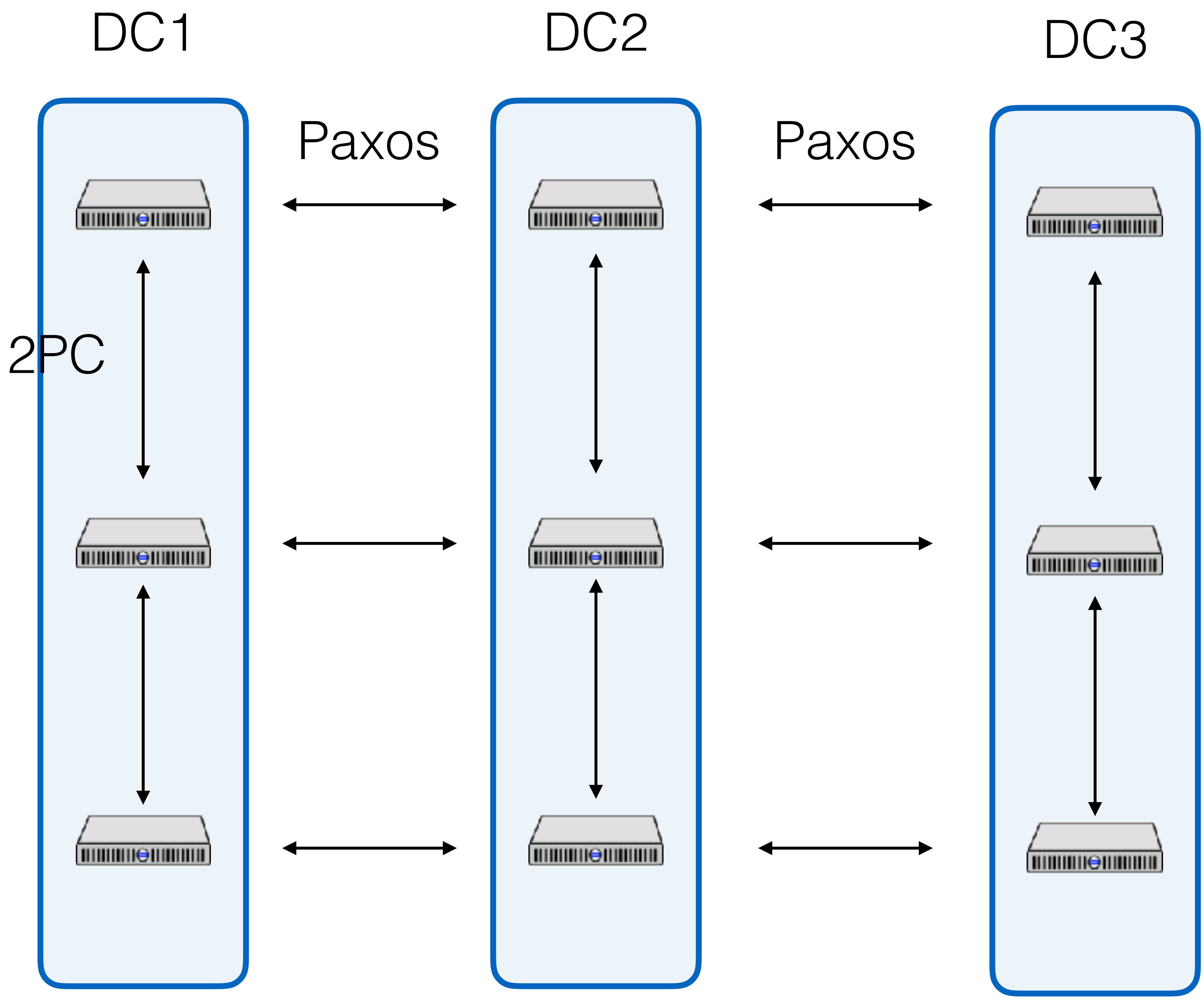
- Each shard is stored in a Paxos group
    - replicated across data centers
    - has a (relatively long-lived) leader
- Transactions span Paxos groups using 2PC
    - use 2PC for transactions
    - leader of each Paxos group tracks locks
    - one group leader becomes the 2PC coordinator, others participants

Basic 2PC/Paxos approach

- during execution, read and write objects
  - contact the appropriate Paxos group leader, acquire locks
- client decides to commit, notifies the coordinator
  - coordinator contacts all shards, sends PREPARE message
  - they Paxos-replicate a prepare log entry (including locks),
  - vote either ok or abort
- if all shards vote OK, coordinator sends commit message
  - each shard Paxos-replicates commit entry
  - leader releases locks


DC1
DC2
DC3


DC1
DC2
DC3
Paxos
Paxos


DC1
DC2
DC3
Paxos
Paxos
2PC

Basic 2PC/Paxos approach

- Note that this is really the same as basic 2PC from before
- Just replaced writes to a log on disk with writes to a Paxos replicated log!
- It is linearizable (= strict serializable = externally consistent)

- So what’s left?
    - Lock-free read-only transactions