Distributed systems

Lecture 13: Distributed mutual exclusion, distributed transactions, and replication

Michaelmas 2019

Dr Martin Kleppmann

(With thanks to Dr Robert N. M. Watson and Dr Steven Hand)

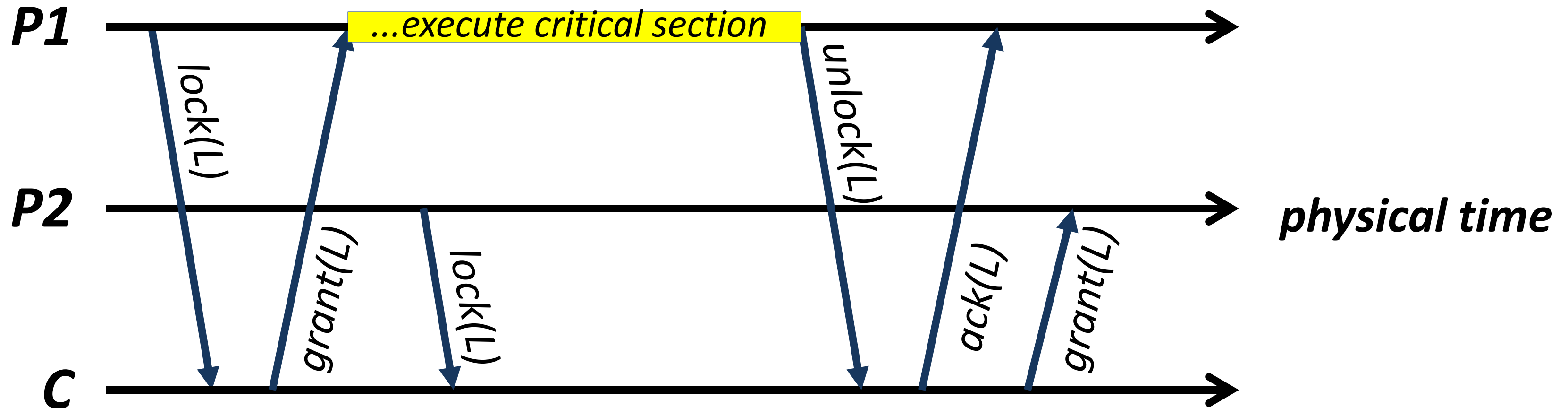
Last time

- Logical time
  - **Lamport clocks** (total order consistent with causality)
  - **Vector clocks** (can tell which events are concurrent)
- Saw how we can build **ordered multicast**
  - Messages between processes in a group
  - Need to distinguish **receipt** and **delivery**
  - Several ordering options: **FIFO**, **causal** or **total**

Distributed mutual exclusion

- In first part of course, saw need to coordinate concurrent processes / threads
  - In particular, considered how to ensure **mutual exclusion**: allow only 1 thread in a critical section
- A variety of schemes possible:
  - test-and-set locks; semaphores; monitors; active objects
- But most of these ultimately rely on hardware support (atomic operations, or disabling interrupts…)
  - not available across an entire distributed system
- Assuming we have some shared distributed resources, how can we provide mutual exclusion in this case?

Solution #1: central lock server



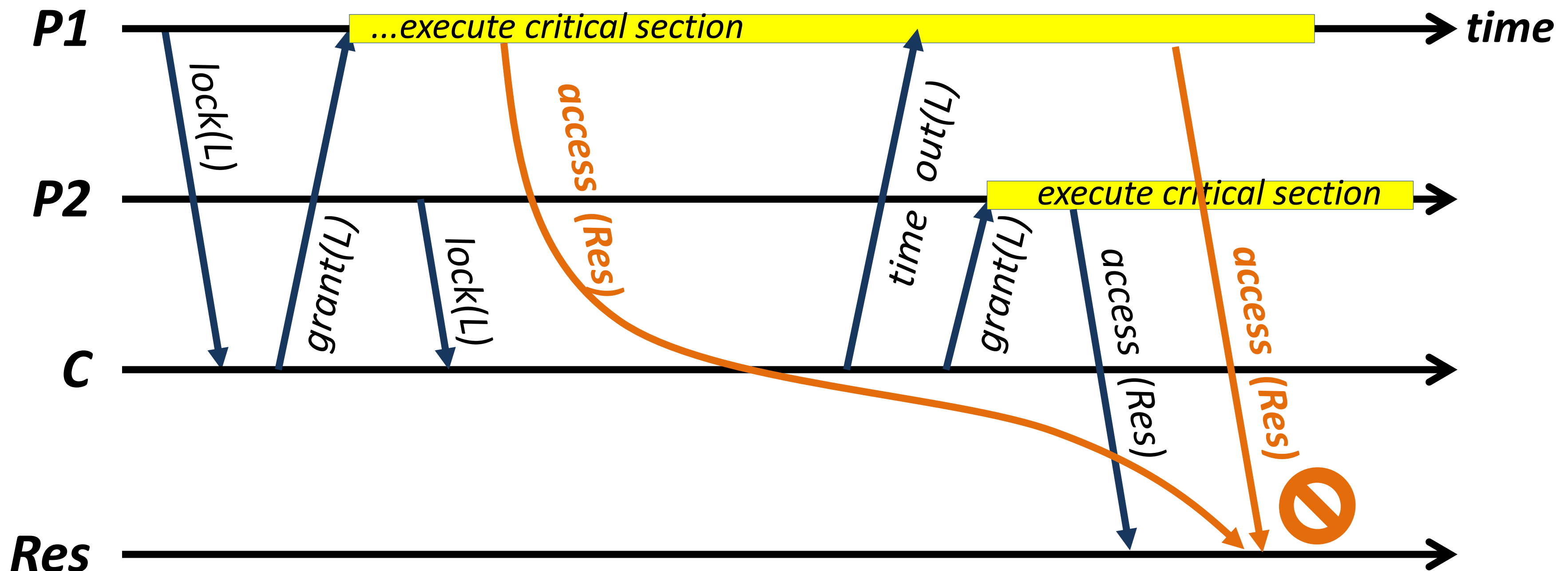
- Nominate one process C as coordinator
  - If $\mathbf{P_i}$ wants to enter critical section, simply sends ***lock*** message to **C**, and waits for a reply
  - If resource free, **C** replies to $\mathbf{P_i}$ with a ***grant*** message; otherwise **C** adds $\mathbf{P_i}$ to a wait queue
  - When finished, P_i sends ***unlock*** message to **C**
  - **C** sends ***grant*** message to first process in wait queue

Central lock server: pros and cons

- Central lock server has some good properties:
  - **Simple** to understand and verify
  - **Live** (providing delays are bounded, and no failure)
  - **Fair** (if queue is fair, e.g. FIFO), and easily supports priorities if we want them
  - **Decent performance**: lock acquire takes one round-trip, and release is ‘free’ with asynchronous messages
- But **C** can become a performance bottleneck…
- … and can’t distinguish crash of **C** from long wait
  - can add additional messages, at some cost

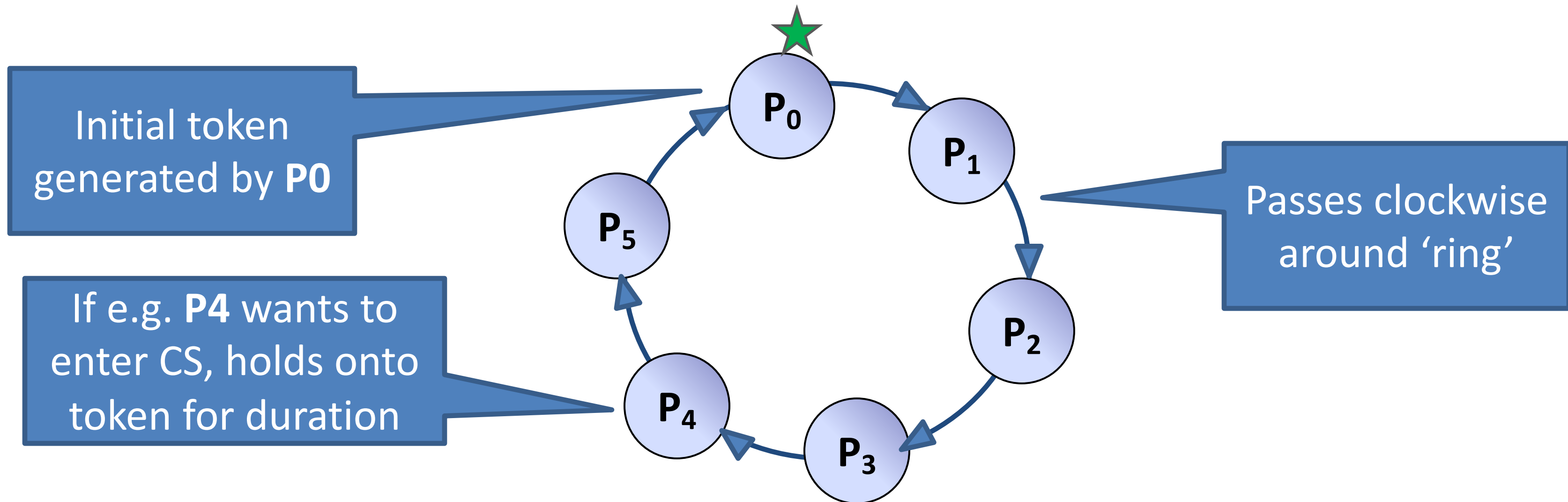
Danger with distributed locks

- P1 is unresponsive. Time out the lock, or wait forever?



- P1 may wake up and continue executing, not knowing that its lock has timed out
  - or request to access resource is delayed in the network
  - in general, no preemption possible in distributed system

Solution #2: token passing



- Avoid central bottleneck
- Arrange processes in a logical ring
  - Each process knows its predecessor & successor
  - Single token passes continuously around ring
  - Can only enter critical section when possess token; pass token on when finished (or if don't need to enter critical section)

Token passing: pros and cons

- Several advantages:
  - Simple to understand: only 1 process ever has token => mutual exclusion guaranteed by construction
  - No central server bottleneck
  - Liveness guaranteed (in the absence of failure)
  - So-so performance (between 0 and N messages until a waiting process enters, 1 message to leave)
- But:
  - Doesn’t guarantee fairness (FIFO order)
  - If a process crashes must repair ring (route around)
  - And worse: may need to regenerate token – tricky!
  - Constant network traffic

Solution #3: total-order multicast

- Due to Lamport [1978]
- Each process maintains its own copy of a request queue
  - P_i wants resource: send "**request P_i**" to all processes (including itself)
  - P_j receives "**request P_i**": enqueue request in queue
  - P_i releases resource: send "**release P_i**" to all processes
  - P_j receives "**release P_i**": remove P_i's request from queue
- Resource is granted to **P_i** when "**request P_i**" is at head of queue
  - Due to total order delivery, all processes agree on order of requests in the queue, and which one is at the head
- Can implement this using Lamport timestamps
  - BUT: no progress if any one process fails!

Aside on consensus

- Locking is a specific example of a more general problem: **consensus**
  - Given a set of **N** processes in a distributed system, how can we get them all to agree on something?
  - e.g. agree on which process holds the lock
- Every process $\mathbf{P_i}$ proposes something (a value $\mathbf{V_i}$)
- A correct solution to consensus must satisfy:
  - **Agreement**: all nodes arrive at the same answer
  - **Validity**: answer is one that was proposed by someone
  - **Termination**: all nodes eventually decide
- e.g. Paxos + variants, Raft, Viewstamped Replication, …

“Consensus is impossible”

- Famous result due to Fischer, Lynch & Patterson (1985)
  - Focuses on an **asynchronous network** (unbounded delays) with at least one process failure
  - Shows that it is possible to get an infinite sequence of states, and hence **never terminate**
  - Given the Internet is an asynchronous network, then this seems to have major consequences!
- Not really:
  - Result actually says we can’t **always guarantee** consensus, **not** that we can **never achieve** consensus
  - And in practice, we can use tricks to mask failures (such as reboot, or replication), and to ignore asynchrony
  - Have seen solutions already, and will see more later

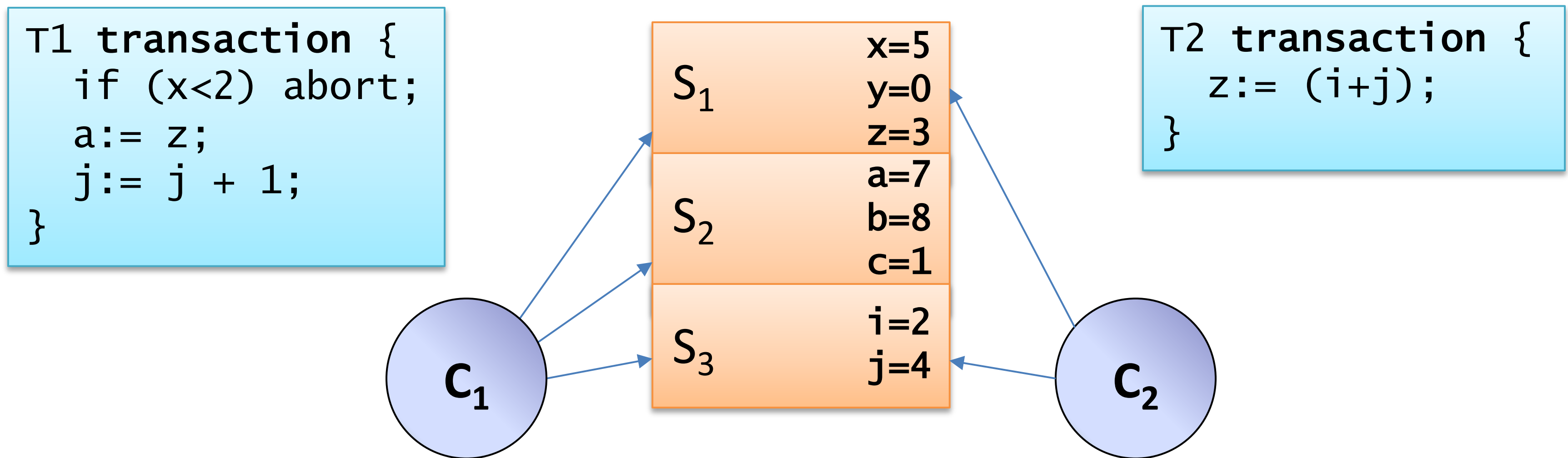
Transaction processing systems

- Earlier looked at **transactions**:
  - **ACID** properties
  - Support for **composite operations** (i.e. a collection of reads and updates to a set of objects)
- A transaction is **atomic** (“all-or-nothing”)
  - If it **commits**, all operations are applied
  - If it **aborts**, it’s as if nothing ever happened
- A committed transaction moves system from one **consistent** state to another
- Transaction processing systems also provide:
  - **isolation** (between concurrent transactions)
  - **durability** (committed transactions survive a crash)
- **Q: Can we bring the {scalability, fault tolerance, …} benefits of distributed systems to transaction processing?**

Distributed transactions

- Scheme described earlier was client/server:
  - E.g., a program (client) accessing a database (server)
- However **distributed transactions** are those which span **multiple** transaction processing servers
- e.g. exactly-once message processing
  - Processing a message has side-effects: updating data in a database
  - Want changes in database to take effect **iff** message is marked as processed
  - Atomically commit side-effects (in database) and message-delivery status (in message broker)
  - If either fails, transaction is aborted in both systems, and message processing can be safely retried

A model of distributed transactions



- Multiple servers (**S_1**, **S_2**, **S_3**, …), each holding some objects which can be **read** and **written** within client transactions
- Multiple concurrent clients (**C_1**, **C_2**, …) who perform transactions that interact with one or more servers
  - E.g. **T1** reads **x**, **z** from **S_1**, writes **a** on **S_2**, reads+writes **j** on **S_3**
  - E.g. **T2** reads **i**, **j** from **S_3**, then writes **z** on **S_1**
- A successful **commit** implies agreement at all servers

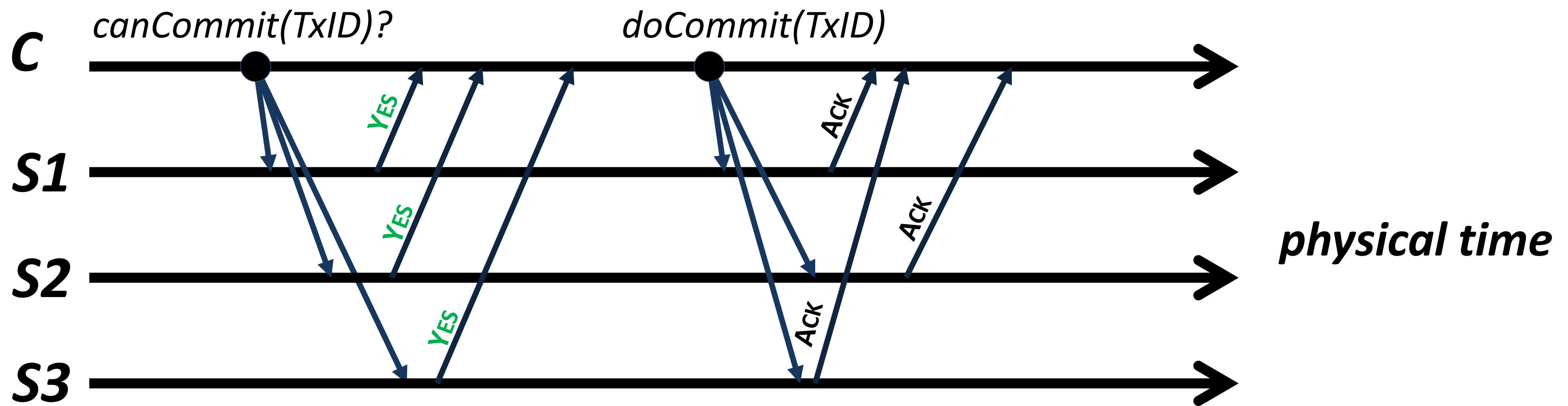
Implementing distributed transactions

- Can build on top of solution for single server:
  - e.g. use **locking** or **shadowing** to provide **isolation**
  - e.g. use **write-ahead log** for durability
- Need to coordinate to either **commit** or **abort**
  - Assume clients create **unique transaction ID**: **TxID**
  - Uses **TxID** in every read or write request to a server S_i
  - First time S_i sees a given **TxID**, it starts a tentative transaction associated with that transaction ID
  - When client wants to commit, must perform **atomic commit** of all tentative transactions across all servers

Atomic commit protocols

- A naïve solution would have client simply invoke **commit**(**TxID**) on each server in turn
  - Will work only if no concurrent conflicting clients, every server commits (or aborts), and no server crashes
- To handle **concurrent clients**, introduce a **coordinator**:
  - A designated machine (can be one of the servers)
  - Clients ask coordinator to commit on their behalf… and hence coordinator can **serialize** concurrent commits
- To handle **inconsistency/crashes**, the coordinator:
  - Asks all involved servers if they *could* commit **TxID**
  - Servers $\mathbf{S_i}$ reply with a vote $\mathbf{V_i}$ = { **COMMIT**, **ABORT** }
  - If all $\mathbf{V_i}$ = COMMIT, coordinator multicasts **doCommit**(**TxID**)
  - Otherwise, coordinator multicasts **doAbort**(**TxID**)

Two-phase commit (2PC)



- This scheme is called **two-phase commit (2PC)**:
  - First phase is **voting**: collect votes from all parties
  - Second phase is **completion**: either abort or commit
- Doesn't require ordered multicast, but needs reliability
  - If server fails to respond by timeout, implicit vote to **abort**
- Once all ACKs received, inform client of commit success

2PC: additional details

- Client (or any server) can abort during execution: simply multicasts **doAbort**(**TxID**) to all servers
  - E.g., if client transaction explicitly aborts or server fails
- If a server votes **NO**, can **abort** at once locally
- If a server votes **YES**, it **must** be able to commit if subsequently asked by coordinator:
  - Before voting to commit, server will **prepare** by writing entries into log and flushing to disk
  - Records all requests from/responses to coordinator
  - Hence even if crashes **after** voting to commit, will be able to recover on reboot

2PC: coordinator crashes

- Coordinator must also **persistently log** events:
  - Including initial message from client, requesting votes, receiving replies, and final decision made
  - Lets it reply if (restarted) client or server asks for outcome
  - Also lets coordinator recover from reboot, e.g. re-send any vote requests without responses, or reply to client
- One additional problem occurs if coordinator crashes **after phase 1**, but **before initiating phase 2**:
  - Servers will be uncertain of outcome…
  - If voted to commit, will have to continue to hold locks, etc
- Can implement fault-tolerant distributed coordinator using consensus algorithm (e.g. Paxos)

Replication

- Many distributed systems involve **replication**
  - Multiple copies of some object stored at different servers
  - Multiple servers capable of providing some operation(s)
- Three key advantages:
  - **Load-Balancing**: if have many replicas, then can spread out work from clients between them
  - **Lower Latency**: if replicate an object/server close to a client, will get better performance
  - **Fault-Tolerance**: can tolerate the failure of some replicas and still provide service
- Examples include DNS, web & file caching (& content-distribution networks), replicated databases, …

Replication in a single system

- A good single-system example is **RAID**:
  - RAID = Redundant Array of Inexpensive Disks
  - Disks are cheap, so use several instead of just one
  - If replicate data across disks, can tolerate disk crash
  - If don't replicate data, appearance of a single larger disk
- A variety of different configurations (levels)
  - RAID 0: **stripe** data across disks, i.e. block 0 to disk 0, block 1 to disk 1, block 2 to disk 0, and so on
  - RAID 1: **mirror** (replicate) data across disks, i.e. block 0 written on disk 0 and disk 1
  - RAID 5: **parity** – write block 0 to disk 0, block 1 to disk 1, and (block 0 XOR block 1) to disk 2
- Improved performance as can access disks in parallel
- With RAID 1, 5 also get fault-tolerance
- NB: More disks **increase risk of single-disk failure** while **reducing probability of fatal multi-disk failure**

Distributed data replication

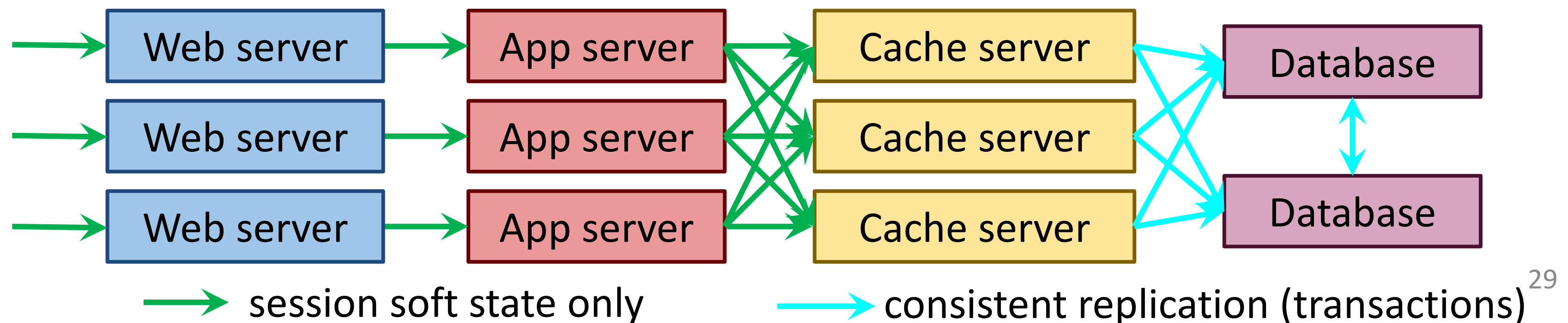
- Have some number of servers (**S_1, S_2, S_3**, …)
  - Each holds a copy of all objects
- Each client **C_i** can access any replica (any **S_i**)
  - E.g. clients can choose closest, or least loaded
- If objects are **read-only**, then trivial:
  - Start with one primary server **P** having all data
  - If client asks **S_i** for an object, **S_i** returns a copy
  - (**S_i** fetches a copy from **P** if it doesn't already have a fresh one)
- Can easily extend to allow updates by **P**
  - When updating object **O**, send **invalidate(O)** to all **S_i**
- In essence, this is how web caching / CDNs work today
- **But what if clients can perform updates?**

Replication and consistency

- More challenging if clients can perform updates
- For example, imagine **x** has value **3** (in all replicas)
  - **C1** requests **write(x, 5)** from **S4**
  - **C2** requests **read(x)** from **S3** (after C1's request has completed)
  - What should occur?
- With **strong consistency/linearizability**, the system behaves as if there was **no replication**:
  - That is, **C2** should read the value **5**
  - Requires coordination between all servers
- With **weak consistency**, **C2** may get 3 or 5 (or …?)
  - Harder to reason about, but better performance
  - Recall **close-to-open consistency** in NFS

Replication for fault tolerance

- Replication for **services**, not just data objects
- Easiest is for a **stateless service**:
  - Simply duplicate functionality over ***k*** machines
  - Clients use any (e.g. closest), fail over to another
- Very few totally stateless services
  - But e.g. many web apps have per-session soft state
  - State generated per-client, lost when client leaves
- For example: multi-tier web farms (Facebook, …):



Passive replication

- Stateful services can use **primary/backup**:
  - Backup server takes over in case of failure
- Based on **persistent logs**, **system checkpoints**:
  - Periodically (or continuously) checkpoint primary
  - If detect failure, start backup from checkpoint
- A few variants trade-off fail-over time:
  - **Cold-standby**: backup server must start service (software), load checkpoint & parse logs
  - **Warm-standby**: backup server has software running in anticipation, must load primary state
  - **Hot-standby**: backup server mirrors primary work, but output is discarded; on failure, enable output

Active replication

- **Alternative**: each of the replicas independently executes each operation
- Use **total order multicast**:
  - Client (or frontend server) sends each request to all replicas by total order multicast
  - All replicas receive operations in the same order, apply them in the same order, then respond
- This is known as **state machine replication**:
  - Replicas must act **deterministically** based on input
  - Same input + same processing = same state
  - Beware of sources of nondeterminism: random numbers, current time, result order…
  - Any errors/transaction aborts must also be made deterministic. What if a replica crashes?

Summary + next time

- Distributed locking + distributed consensus
- Distributed transactions + atomic commit protocols
- Replication + consistency

- (More) replication and consistency
  - Strong consistency
  - Quorum-based systems
  - Weaker consistency
- Consistency, availability and partitions
- Further replication models
- Amazon/Google case studies