Last time

- Looked at general issue of **consensus**:  
  - How to get processes to agree on something  
  - (FLP says “impossible” in asynchronous networks with at least 1 failure ... but in practice we’re ok ;-)  
  - General idea useful for distributed mutual exclusion, leader election: relies being able to detect failures  
- Also looked at **distributed transactions**:  
  - Need to commit a set of “sub-transactions” across multiple servers – want all-or-nothing semantics  
  - Use **atomic commit** protocol like 2PC  
- Started on **replication**: using multiple copies to gain performance, load-balancing & fault tolerance
Replication in Distributed Systems

- Have some number of servers \((S_2, S_2, S_3, \ldots)\)
  - 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 one)
- Can easily extend to allow updates by \(P\)
  - When updating object \(O\), send invalidate(\(O\)) to all \(S_i\)
  - (Or add just tag all objects with ‘valid-until’ field)
- In essence, this is how web caching / CDNs work today

Replication and Consistency

- Gets more challenging if clients can perform updates
- For example, imagine \(x\) has value 3 (in all replicas)
  - \(C_1\) requests \texttt{write}(\(x, 5\)) from \(S_4\)
  - \(C_2\) requests \texttt{read}(\(x\)) from \(S_3\)
  - What should occur?
- With strong consistency, the distributed system behaves as if there is no replication present:
  - i.e. in above, \(C_2\) should get the value 5
  - requires coordination between all servers
- With weak consistency, \(C_2\) may get 3 or 5 (or ...?)
  - Less satisfactory, but much easier to implement
Achieving Strong Consistency

• Need to ensure any update propagates to all replicas before allow any subsequent reads
• One solution:
  – When $S_i$ receives request to update $x$, first locks $x$ at all other replicas
  – Once successful, $S_i$ makes update, and propagates to all other replicas, who acknowledge
  – Finally, $S_i$ instructs all replicas to unlock
• Need to handle failure (of replica, or network)
  – Add step to tentatively apply update, and only actually apply (“commit”) update if all replicas agree
• We’ve reinvented distributed transactions & 2PC ;-) 

Quorum Systems

• Transactional consistency works, but:
  – High overhead, and
  – Poor availability during update (worse if crash!)
• An alternative is a quorum system:
  – Imagine there are $N$ replicas, a write quorum $Q_w$, and a read quorum $Q_r$, where $Q_w > N/2$ and $(Q_w + Q_r) > N$
• To perform a write, must update $Q_w$ replicas
  – Ensures a majority of replicas have new value
• To perform a read, must read $Q_r$ replicas
  – Ensures that we read at least one updated value
Example

• Seven replicas (N=7), $Q_w = 5$, $Q_r = 3$
• All objects have associated version (T, S)
  – T is logical timestamp, initialized to zero
  – S is a server ID (used to break ties)
• Any write will update at least $Q_w$ replicas
• Performing a read is easy:
  – Choose replicas to read from until get $Q_r$ responses
  – Correct value is the one with highest version

Quorum Systems: Writes

• Performing a write is trickier:
  – Must ensure get entire quorum, or cannot update
  – Hence need a commit protocol (as before)
• In fact, transactional consistency is a quorum protocol with $Q_w = N$ and $Q_r = 1$!
  – But when $Q_w < N$, additional complexity since must bring replicas up-to-date before updating
• Quorum systems are good when expect failures
  – Additional work on update, additional work on reads...
  – ... but increased availability during failure
Weak Consistency

- Maintaining strong consistency has costs:
  - Need to coordinate updates to all (or \( Q_w \)) replicas
  - Slow... and will block other accesses for the duration
- **Weak consistency** provides fewer guarantees:
  - e.g. C1 updates (replica of) object x at S3
  - S3 lazily propagates changes to other replicas
  - Other clients can potentially read old (“stale”) value
- Considerably **more efficient**:
  - Write is simpler, and doesn’t need to wait for communication with lots of other replicas...
  - ... hence is also **more available** (i.e. fault tolerant)

FIFO Consistency

- As with group communication primitives, various ordering guarantees possible
- **FIFO consistency**: all updates at \( S_i \) occur in the same order at all other replicas
  - As with FIFO multicast, can buffer for as long as we like!
  - But says nothing about how \( S_i \)’s updates are interleaved with \( S_j \)’s at another replica (may put \( S_j \) first, or \( S_i \), or mix)
- Still useful in some circumstances
  - e.g. single user accessing different replicas at disjoint times
  - Essentially primary replication with primary=last accessed
Eventual Consistency

• FIFO consistency doesn’t provide very nice semantics:
  – e.g. we write first version of file f to S₁
  – later we read f from S₂, and write version 2
  – later again we read f from S₃ – changes lost!
• What happened?
  – Update from S₁ arrived to S₃ after those from S₂, who thus
    overwrote them (stoooopid S₃)
• A desirable property in weakly consistent systems is
  that they converge to a more correct state
  – i.e. in the absence of further updates, every replica will
    eventually end up with the same latest version
• This is called **eventual consistency**

Implementing Eventual Consistency

• Servers Sᵢ keep a **version vector** Vᵢ(∫) for each object
  – For each update of O on Sᵢ, increment Vᵢ(∫)[∫]
  – (essentially a vector clock reused as a version number)
• Servers synchronize pair-wise from time to time
  – For each object O, compare Vᵢ(∫) to Vⱼ(∫)
  – If Vᵢ(∫) < Vⱼ(∫), Sᵢ gets an up-to-date copy from Sⱼ;
    if Vⱼ(∫) < Vᵢ(∫), Sⱼ gets an up-to-date copy from Sᵢ.
• If Vᵢ(∫) ~ Vⱼ(∫) we have a **write-conflict**:
  – Concurrent updates have occurred at 2 or more servers
  – Must apply some kind of reconciliation method
  – (similar to revision control systems, and equally painful)
Example: Amazon’s Dynamo

- Storage service used within Amazon’s WS
  - By Amazon itself, and by 3rd party service providers
- Designed to emphasize availability above consistency:
  - SLA to ensure bounded response time 99.99% of the time
  - if customer wants to add something to shopping basket and there’s a failure... still want addition to ‘work’
  - Even if get (temporarily) inconsistent view... fix later!
- Built around notion of a so-called sloppy quorum:
  - Have N, Q_w, Q_r as before ... but don’t actually require that Q_w > N/2, or that (Q_w + Q_r) > N
  - Instead make tunable: lower Q values = higher availability
  - Also let system continue during failure; add a new replica

Session Guarantees

- Eventual consistency seems great, but how can you program to it?
  - Need to know something about what guarantees are provided to the client
- These are called session guarantees:
  - Not system wide, just for one (identified) client
  - Client must be a more active participant, e.g. client maintains version vectors of objects it has read & written
- Example: Read Your Writes (RYW):
  - if C_i writes a new value to x, a subsequent read of x should see this update ... even if C_i is now reading from a different replica
  - Need C_i to remember highest id of any update it made
  - Only read from a server if it has seen that update
Session Guarantees & Availability

• There are a variety of session guarantees
  – All deal with allowable state on replica given history of accesses by a specific client
  – (further examples included in additional, non-examinable material downloadable from course web page)
• Session guarantees are weaker than strong consistency, but stronger than ‘pure’ weak consistency:
  – But this means that they sacrifice availability
  – i.e. choosing not to allow a read or write if it would break a session guarantee means not allowing that operation!
  – ‘pure’ weak consistency would allow the operation
• Can we get the best of both worlds?

Consistency, Availability & Partitions

• Short answer: No ;-)
• The CAP Theorem (Brewer 2000, Gilbert & Lynch 2002) says you can only guarantee two of:
  – Consistent data, Availability, Partition-tolerance
• … in a single system.
• In local-area systems, can sometimes drop partition-tolerance by using redundant networks
• In the wide-area, this is not an option:
  – Must choose between consistency & availability
  – Most internet-scale systems ditch consistency
• NB: this doesn’t mean that things are always inconsistent, just that they’re not always guaranteed to be consistent
Replication and Fault-Tolerance

• Can also use replication for a service:
  • Easiest is for **stateless services**:
    -- Simply duplicate functionality in K machines
    -- Clients use any (e.g. closest), fail over to another
  • Very few totally stateless services, but e.g. much of the web only has per-session soft-state:
    -- State generated per-client, lost when client leaves
  • Commonly used to scale multi-tier web farms:
    -- First and second tiers (web servers and app servers) only have per-session soft-state \(\Rightarrow\) trivial to replicate
    -- (clients are independent, so no coordination needed)
    -- Third tier (storage/db tier) either partitioned (disjoint clients on different servers), or implements consistent replication

Primary/Backup (Passive) Replication

• A solution for stateful services is **primary/backup**:
  -- Backup server takes over in case of failure
• Based around persistent logs and 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 – just needs to load primary state
  -- **Hot-standby**: backup server mirrors primary work, but output is discarded; on failure, enable output
Active Replication

- Have K replicas running at all times
- Front-end server acts as an ordering node:
  - Receives requests from client and forwards them to all replicas using totally ordered multicast
  - Replicas each perform operation and respond to front-end
  - Front-end gathers responses, and replies to client
- Typically require replicas to be “state machines”:
  - i.e. act deterministically based on input
  - Idea is that all replicas operate ‘in lock step’
- Active replication is expensive (in terms of resources)...
  - ... and not really worth it in the common case.
  - However valuable if consider Byzantine failures