CS 194: Lecture 15

Midterm Review

Notation

- Red titles mean start of new topic
- They don't indicate increased importance…..

Clock Synchronization

- Distributed systems often require roughly consistent notions of time
- Usually the requirement isn't that time is accurate (UTC), only that it is synchronized
- However, synchronizing machines with UTC automatically synchronizes all machines with each other
- Two well-known methods:
  - Cristian's algorithm (UTC time-server based)
  - Berkeley algorithm (no UTC signal, but master)

Clock Synchronization (Cristian)

- Client polls time server (which has external UTC source)
- Gets time from server
- Adjusts received time by adding estimate of one-way delay
  - Estimates travel time as 1/2 of RTT
  - Adds this to server time
- Errors introduced not by delays, but by asymmetry in delays (path to server and path from server)

Clock Synchronization (Berkeley)

- Time master polls clients (master has no UTC)
- Gets time from each client, and averages
- Sends back message to each client with a recommended adjustment
- Clocks are synchronized, but not UTC
- Errors arise when nodes have different delays from master

Logical Clocks

- Most algorithms don't require tightly synchronized clocks, but they often require a common notion of causality
- That is, events can be ordered arbitrarily, as long as causality isn't violated
- For example, it doesn't matter whether I updated my password in Japan before or after someone saved a file in Chile, as long as no messages or other interactions occurred between the two systems
- Lamport captured this notion of causality

Lamport Timestamps

- When message arrives, if process time is less than timestamp s, then jump process time to s+1
- Clock must tick once between every two events
- If A → B then must have L(A) < L(B)
    - logical clock ordering never violates causaility
- If L(A) < L(B), it does NOT follow that A → B
    - Lamport clocks leave some causal ambiguity

Vector Timestamps Definition

- $V_I[I]$: number of events occurred in process I
    - Not using Lamport's rule of jumping clocks ahead!
- $V_I[J] = K$: process I knows that K events have occurred at process J
- All messages carry vectors
- When J receives vector v, for each K it sets $V_J[K] = v[K]$ if it is larger than its current value
- It then updates $V_J[J]$ by one (to reflect recv event)

Questions

- Can a message from I to J have v[J] greater than the current value of $V_J[J]$?
- Can it be equal? (not after J updates after receipt!)
- Right after a message from I to J is received and V_J is updated, can you have $v[K] > V_J[K]$?
- Therefore, after a message from I to J arrives, V_J dominates V_I
    - Greater than or equal in every entry

Vector Timestamps Properties

- A → B, if and only if the vector associated with B dominates that of A
- A and B are concurrent if and only if the vectors from A and B are not comparable:
    - At least one element from A greater than that of B
    - At least one element from B greater than that of A

Elections

- Need to select a special node, that all other nodes agree on
- Assume all nodes have unique ID
- Example methods for picking node with highest ID
    - Bully algorithm
    - Gossip method

Exclusion

- Ensuring that a critical resource is accessed by no more than one process at the same time
- Methods:
    - Centralized coordinator: ask, get permission, release
    - Distributed coordinator: treat all nodes as coordinator
        - If two nodes are competing, timestamps resolve conflict
    - Interlocking permission sets: Every node I asks permission from set P[I], where P[I] and P[J] always have nonempty intersections

Concurrency Control

- Want to allow several transactions to be in progress
- But the result must be the same as some sequential order of transactions
- Use locking policies:
  - Grab and hold
  - Grab and unlock when not needed
  - Lock when first needed, unlock when done
  - Two-phase locking
- Which policies can have deadlock?

Alternative to Locking

- Use timestamp ordering
  - Retrying an aborted transaction uses new timestamp
- Data items have:
  - Read timestamp tR: timestamp of transaction that last read it
  - Write timestamp tW: timestamp of transaction that last wrote it
- Pessimistic timestamp ordering:
  - When reading, abort if ts<tW(A)
  - When writing, abort if ts<tR(A)
- Optimistic: do all your work, then check to make sure no timestamp conditions are violated

Data Replication and Consistency

- Scalability requires replicated data
- Application correctness requires some form of consistency
  - Here we focus on individual operations, not transactions
- How do we reconcile these two requirements?

Models of Consistency

- Strict consistency (in your dreams…)
- Linearizable (in your proofs….)
- Sequential consistency: same order of operations
- Causal consistency: all causal operations ordered
- FIFO consistency: operations within process ordered

Mechanisms for Sequential Consistency

- Local cache replicas: pull, push, lease
  - Why does this produce sequential consistency?
- Primary-based replication protocols: [won’t ask]
- Replicated-write protocols: quorum techniques
- Cache-coherence protocols [didn’t cover]

Quorum-based Protocols

- Assign a number of votes V(I) to each replica I
  - Let V be the total number of votes
- VR=read quorum, VW=write quorum
- Requirements: VR+VW > V and VW > V/2
- Examples:
  - Read-one, write-all
  - Majority

Scaling

- None of these protocols scale
- To read or write, you have to either
  - (a) contact a primary copy
  - (b) contact over half of the replicas
- All this complication is to ensure sequential consistency
- Can we weaken sequential consistency without losing some important features?

Eventual Consistency

- Rather than insisting that the order of operations meet some standard, we ask only that in the end all nodes eventually agree
  - If updates are stopped, will mechanism produce uniform replicas?
- Some of the previous notions of consistency did not produce this!
  - FIFO, and causal

Implementing Eventual Consistency

- All writes eventually propagate to all replicas
- Writes, when they arrive, are applied in the same order at all replicas
  - Easily done with timestamps and “undo”

Update Propagation

- Rumor or epidemic stage:
  - Attempt to spread an update quickly
  - Willing to tolerate incompletely coverage in return for reduced traffic overhead
  - Push/pull methods spreading methods (pull better than push)
- Correcting omissions:
  - Making sure that replicas that weren’t updated during the rumor stage get the update
  - Anti-entropy

Bayou

Will NOT be on midterm!

Bayou Design Choices

- Variable connectivity ⇒ Flexible update propagation
  - Incremental progress, pairwise communication
- Variable end-nodes ⇒ Flexible notion of clients and servers
  - Some nodes keep state (servers), some don’t (clients)
  - Laptops could have both, PDAs probably just clients
- Availability crucial ⇒ Must allow disconnected operation
  - Conflicts inevitable
  - Use application-specific conflict detection and resolution

Components of Design

- Update propagation
- Conflict detection
- Conflict resolution
- Session guarantees

The CAP Theorem

- Perspective on tradeoffs in distributed systems
- Asks why there are different design philosophies

BASE or ACID?

- Classic distributed systems: focused on ACID semantics
  - A: Atomic
  - C: Consistent
  - I: Isolated
  - D: Durable
- Modern Internet systems: focused on BASE
  - Basically Available
  - Soft-state (or scalable)
  - Eventually consistent

Why the Divide?

- What goals might you want from a shared-date system?
  - C, A, P
- **Strong Consistency**: all clients see the same view, even in the presence of updates
- **High Availability**: all clients can find some replica of the data, even in the presence of failures
- **Partition-tolerance**: system as a whole can survive partition

CAP Theorem

- You can only have two out of these three properties
- The choice of which feature to discard determines the nature of your system

Consistency and Availability

- Comment:
  - Providing transactional semantics requires all functioning nodes to be in contact with each other
- Examples:
  - Single-site and clustered databases
  - Other cluster-based designs
- Typical Features:
  - Two-phase commit
  - Cache invalidation protocols
  - Classic DS style

Consistency and Partition-Tolerance

- Comment:
  - If one is willing to tolerate system-wide blocking, then can provide consistency even when there are temporary partitions
- Examples:
  - Distributed databases
  - Distributed locking
  - Quorum (majority) protocols
- Typical Features:
  - Pessimistic locking
  - Minority partitions unavailable
  - Also common DS style
    - Voting vs primary replicas

Partition-Tolerance and Availability

- Comment:
  - Once consistency is sacrificed, life is easy….
- Examples:
  - DNS
  - Web caches
  - Coda
  - Bayou
- Typical Features:
  - TTLs and lease cache management
  - Optimistic updating with conflict resolution
  - This is the “Internet design style”

Summary of Techniques/Tradeoffs

▪ Expiration-based caching:	AP not C
▪ Quorum/majority algorithms:	PC not A
▪ Two-phase commit:	AC not P