CS5412: THE BASE METHODOLOGY VERSUS THE ACID MODEL

Lecture VIII

Ken Birman

Today's lecture will be a bit short

- We have a guest with us today: Kate Jenkins from Akamai
  - The world's top "content hosting" company
  - They make the web fast and Kate leads a group that using sophisticated mathematical models to optimize the way the company manages that content
  - Issue is to offer snappy response while also making the best possible use of internal communication bandwidth and storage
- Kate is also interviewing job applicants for a number of Akamai openings
- After her 30-minute talk I'll tell you about BASE and Dynamo

Methodology versus model?

- Today's lecture is about an apples and oranges debate that has gripped the cloud community
  - A methodology is a "way of doing" something
    - For example, there is a methodology for starting fires without matches using flint and other materials
  - A model is really a mathematical construction
    - We give a set of definitions (i.e. fault-tolerance)
    - Provide protocols that provably satisfy the definitions
    - Properties of model, hopefully, translate to application-level guarantees

The ACID model

- A model for correct behavior of databases
- Name was coined (no surprise) in California in 60’s
  - **Atomicity**: even if “transactions” have multiple operations, does them to completion (commit) or rolls back so that they leave no effect (abort)
  - **Consistency**: A transaction that runs on a correct database leaves it in a correct (“consistent”) state
  - **Isolation**: It looks as if each transaction ran all by itself. Basically says “we’ll hide any concurrency”
  - **Durability**: Once a transaction commits, updates can’t be lost or rolled back

ACID as a methodology

Body of the transaction performs reads and writes, sometimes called queries and updates

- We teach it all the time in our database courses
- Students write tra

Begin signals the start of the transaction

```
Begin
    let employee t = Emp.Record("Tony");
    t.status = "retired";
    ∀ customer c: c.A
        c.AccountR
Commit;
```

Commit asks the database to make the effects permanent. If a crash happens before this, or if the code executes **Abort**, the transaction rolls back and leaves no trace

- System executes this code in an all-or-nothing way

Why ACID is helpful

- Developer doesn't need to worry about a transaction leaving some sort of partial state
  - For example, showing Tony as retired and yet leaving some customer accounts with him as the account rep
- Similarly, a transaction can't glimpse a partially completed state of some concurrent transaction
  - Eliminates worry about transient database inconsistency that might cause a transaction to crash
  - Analogous situation: thread A is updating a linked list and thread B tries to scan the list while A is running

Serial and Serializable executions

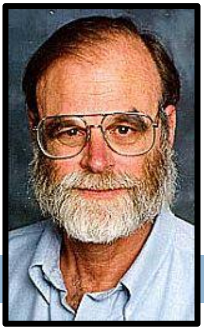
- A "serial" execution is one in which there is at most one transaction running at a time, and it always completes via commit or abort before another starts
- "Serializability" is the "illusion" of a serial execution
  - Transactions execute concurrently and their operations interleave at the level of the database files
  - Yet database is designed to guarantee an outcome identical to some serial execution: it masks concurrency
  - Will revisit this topic in April and see how they do it
  - In past they used locking; these days "snapshot isolation"

All ACID implementations have costs

- Locking mechanisms involve competing for locks and there are overheads associated with how long they are held and how they are released at Commit
- Snapshot isolation mechanisms using locking for updates but also have an additional *version* based way of handing reads
  - Forces database to keep a history of each data item
  - As a transaction executes, picks the versions of each item on which it will run
- So… there are costs, not so small

Dangers of Replication

[The Dangers of Replication and a Solution . Jim Gray, Pat Helland, Dennis Shasha. Proc. 1996 ACM SIGMOD.]



- Investigated the costs of transactional ACID model on replicated data in "typical" settings
  - Found two cases
    - Embarrassingly easy ones: transactions that don't conflict at all (like Facebook updates by a single owner to a page that others might read but never change)
    - Conflict-prone ones: transactions that sometimes interfere and in which replicas could be left in conflicting states if care isn't taken to order the updates
  - Scalability for the latter case will be *terrible*
- Solutions they recommend involve sharding and coding transactions to favor the first case

Approach?

- They do a paper-and-pencil analysis
  - Estimate how much work will be done as transactions execute, roll-back
  - Count costs associated with doing/undoing operations and also delays due to lock conflicts that force waits
- Show that even under very optimistic assumptions slowdown will be $O(n^2)$ in size of replica set (shard)
- If approach is naïve, $O(n^5)$ slowdown is possible!

This motivates BASE

[D. Pritchett. BASE: An Acid Alternative. ACM Queue, July 28, 2008.]

- Proposed by eBay researchers
  - Found that many eBay employees came from transactional database backgrounds and were used to the transactional style of "thinking"
  - But the resulting applications didn't scale well and performed poorly on their cloud infrastructure
- Goal was to guide that kind of programmer to a cloud solution that performs much better
  - BASE reflects experience with real cloud applications
  - "Opposite" of ACID

A "methodology"

- BASE involves step-by-step transformation of a transactional application into one that will be far more concurrent and less rigid
  - But it doesn't guarantee ACID properties
  - Argument parallels (and actually cites) CAP: they believe that ACID is too costly and often, not needed
  - BASE stands for "**Basically Available Soft-State Services with Eventual Consistency**".

Terminology

- **Basically Available:** Like CAP, goal is to promote rapid responses.
  - BASE papers point out that in data centers partitioning faults are very rare and are mapped to crash failures by forcing the isolated machines to reboot
  - But we may need rapid responses even when some replicas can't be contacted on the critical path

Terminology

- **Basically Available:** Fast response even if some replicas are slow or crashed
- **Soft State Service:** Runs in first tier
  - Can't store any permanent data
  - Restarts in a "clean" state after a crash
  - To remember data either replicate it in memory in enough copies to never lose all in any crash or pass it to some other service that keeps "hard state"

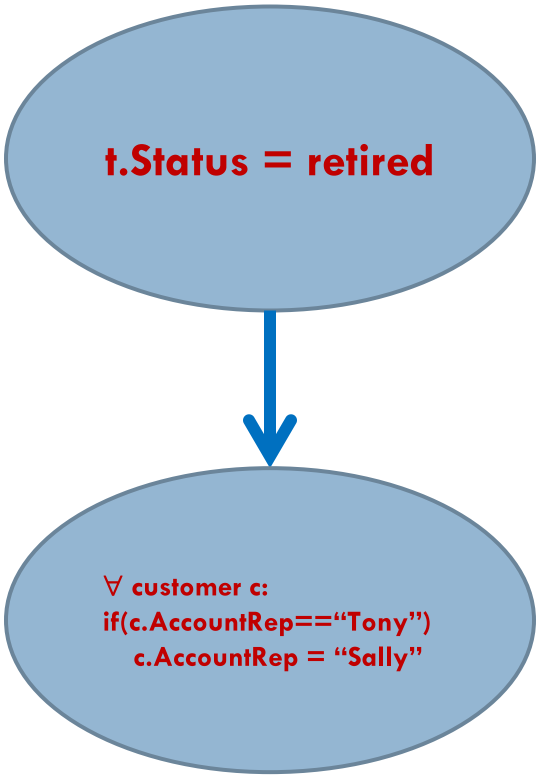
Terminology

- **Basically Available:** Fast response even if some replicas are slow or crashed
- **Soft State Service:** No durable memory
- **Eventual Consistency:** OK to send "optimistic" answers to the external client
  - Could use cached data (without checking for staleness)
  - Could guess at what the outcome of an update will be
  - Might skip locks, hoping that no conflicts will happen
  - Later, if needed, correct any inconsistencies in an offline cleanup activity

How BASE is used

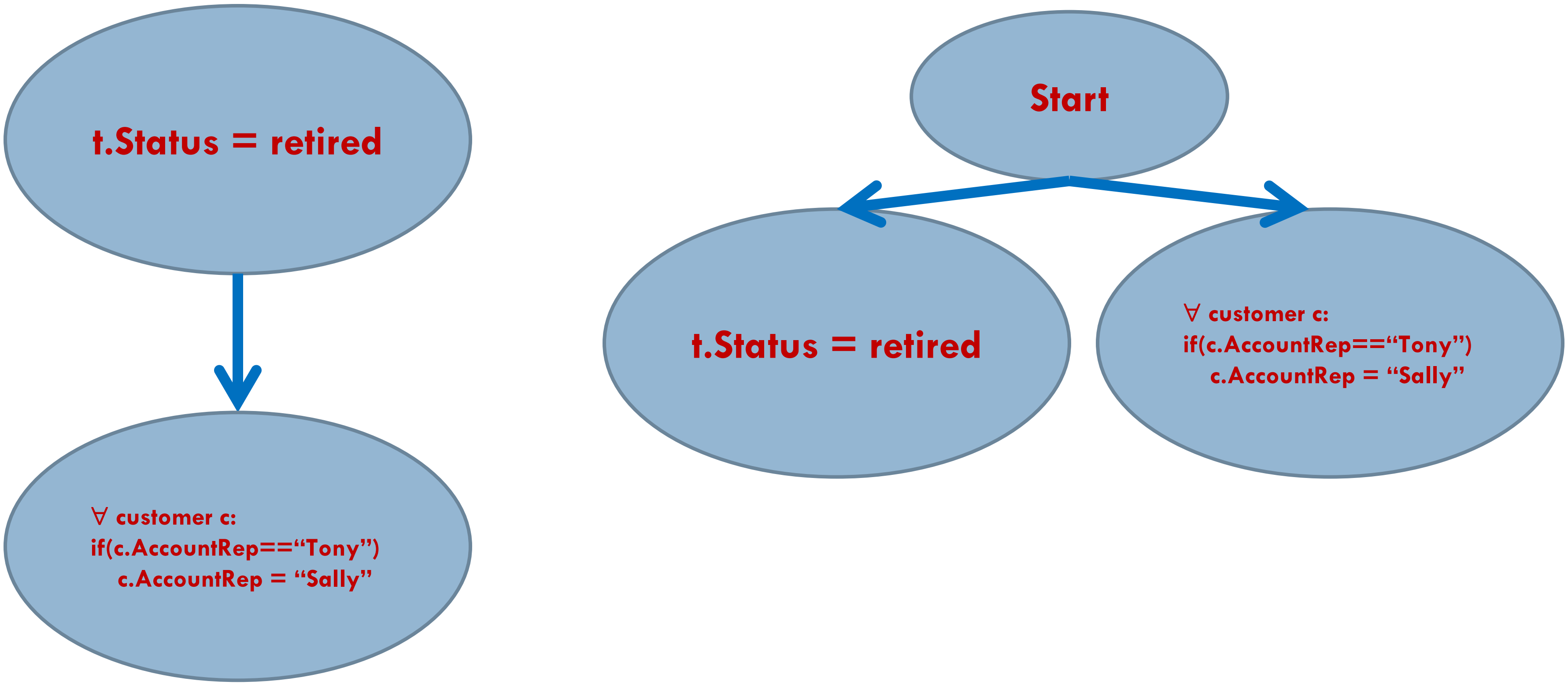
- Start with a transaction, but remove Begin/Commit
  - Now fragment it into "steps" that can be done in parallel, as much as possible
  - Ideally each step can be associated with a single event that triggers that step: usually, delivery of a multicast
- Leader that runs the transaction stores these events in a "message queuing middleware" system
  - Like an email service for programs
  - Events are delivered by the message queuing system
  - This gives a kind of all-or-nothing behavior

Base in action




```
Begin
    let employee t = Emp.Record(“Tony”);
    t.status = “retired”;
    ∀ customer c: c.AccountRep==“Tony”
            c.AccountRep = “Sally”
Commit;
```

Base in action



More BASE suggestions

- Consider sending the reply to the user before finishing the operation
- Modify the end-user application to mask any asynchronous side-effects that might be noticeable
  - In effect, "weaken" the semantics of the operation and code the application to work properly anyhow
- Developer ends up thinking hard and working hard!

Before BASE… and after

- Code was often much too slow, and scaled poorly, and end-user waited a long time for responses

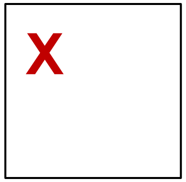
- With BASE
  - Code itself is way more concurrent, hence faster
  - Elimination of locking, early responses, all make end-user experience snappy and positive
  - But we do sometimes notice oddities when we look hard

BASE side-effects

- Suppose an eBay auction is running fast and furious
  - Does every single bidder necessarily see every bid?
  - And do they see them in the identical order?
- Clearly, everyone needs to see the winning bid
- But slightly different bidding histories shouldn't hurt much, and if this makes eBay 10x faster, the speed may be worth the slight change in behavior!

BASE side-effects

- Upload a YouTube video, then search for it
  - You may not see it immediately
- Change the "initial frame" (they let you pick)
  - Update might not be visible for an hour
- Access a FaceBook page when your friend says she's posted a photo from the party
  - You may see an



BASE in action: Dynamo

- Amazon was interested in improving the scalability of their shopping cart service

- A core component widely used within their system
  - Functions as a kind of key-value storage solution
  - Previous version was a transactional database and, just as the BASE folks predicted, wasn't scalable enough
  - Dynamo project created a new version from scratch

Dynamo approach

- They made an initial decision to base Dynamo on a Chord-like DHT structure
- Plan was to run this DHT in tier 2 of the Amazon cloud system, with one instance of Dynamo in each Amazon data center and no "linkage" between them
- This works because each data center has "ownership" for some set of customers and handles all of that person's purchases locally.

The challenge

- Amazon quickly had their version of Chord up and running, but then encountered a problem

- Chord isn't very "delay tolerant"
  - So if a component gets slow or overloaded, Chord was very impacted
  - Yet delays are common in the cloud (not just due to failures, although failure is one reason for problems)

- Team asked: how can Dynamo tolerate delay?

Idea they had

- Key issue is to find the node on which to store a key-value tuple, or one that has the value

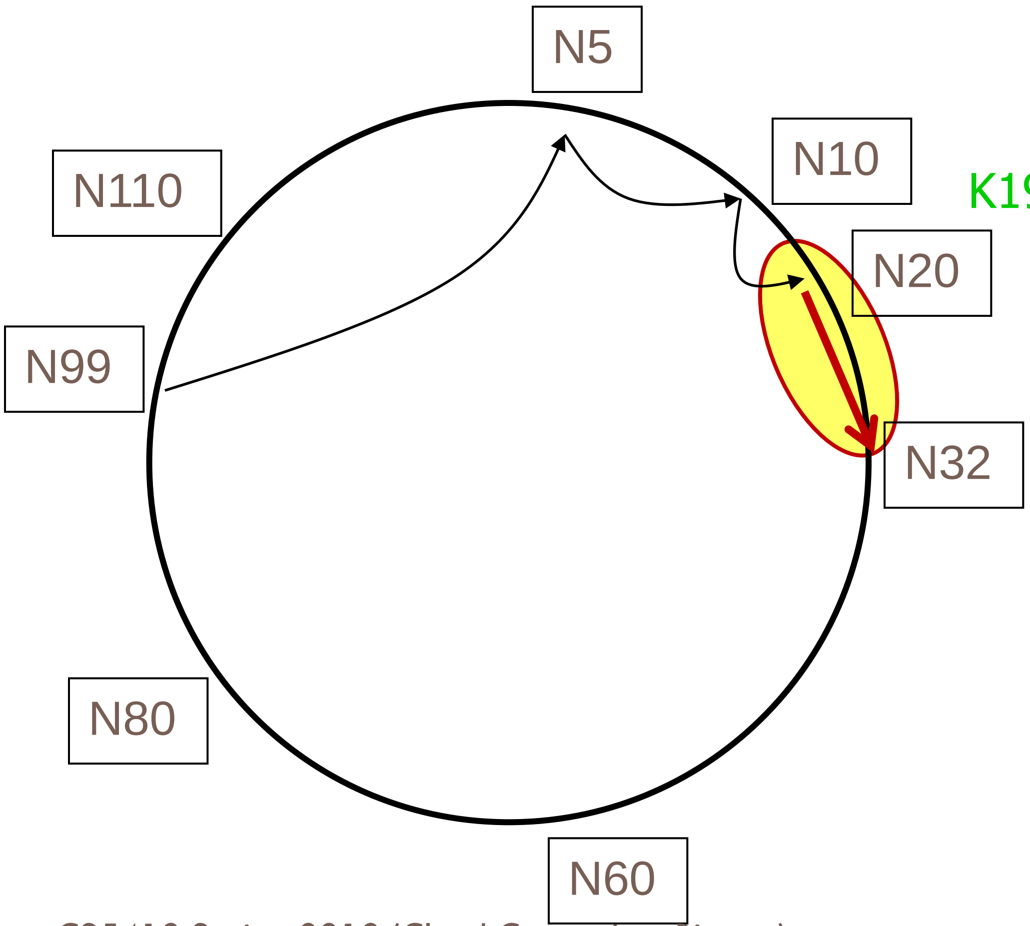
- Routing can tolerate delay fairly easily
  - Suppose node K wants to use the finger to node $K+2^i$ and gets no acknowledgement
  - Then Dynamo just tries again with node $K+2^{i-1}$
  - This works at the "cost" of slight stretch in the routing path in the rare cases when it occurs

What if the actual “home” node fails?

- Suppose that we reach the point at which the next hop should take us to the owner for the hashed key
- But the target doesn’t respond
  - It may have crashed, or have a scheduling problem (overloaded), or be suffering some kind of burst of network loss
  - All common issues in Amazon’s data centers
- Then they do the Get/Put on the *next node that actually responds* even if this is the “wrong” one!

Dynamo example: picture

Lookup(K19)



Dynamo example in pictures

- Notice: Ideally, this strategy works perfectly
  - Recall that Chord normally replicates a key-value pair on a few nodes, so we would expect to see several nodes that "know" the current mapping: a shard
  - After the intended target recovers the repair code will bring it back up to date by copying key-value tuples

- But sometimes Dynamo jumps beyond the target "range" and ends up in the wrong shard

Consequences?

- If this happens, Dynamo will eventually repair itself
  - … But meanwhile, some slightly confusing things happen
- Put might succeed, yet a Get might fail on the key
- Could cause user to "buy" the same item twice
  - This is a risk they are willing to take because the event is rare and the problem can usually be corrected before products are shipped in duplicate

Werner Vogels on BASE

- He argues that delays as small as 100ms have a measurable impact on Amazon's income!
  - People wander off before making purchases
  - So snappy response is king
- True, Dynamo has weak consistency and may incur some delay to achieve consistency
  - There isn't any real delay "bound"
  - But they can hide most of the resulting errors by making sure that applications which use Dynamo don't make unreasonable assumptions about how Dynamo will behave

Conclusion?

- BASE is a widely popular alternative to transactions
  - Used (mostly) for first tier cloud applications
  - Weakens consistency for faster response, later cleans up
  - eBay, Amazon Dynamo shopping cart both use BASE

- Later we'll see that strongly consistent options do exist
  - In-memory chain-replication
  - Send+Flush using $Isis^2$
  - Snapshot-isolation instead of full ACID transactions

- Will look more closely at latter two in a few weeks