ACID and BASE

ACID

Atomicity: a transaction happens or it does not

ACID

Atomicity: a transaction happens or it does not

Consistency: a correct database is still correct afterwards
i.e. money balanced, no dangling pointers

ACID

Atomicity: a transaction happens or it does not

Consistency: a correct database is still correct afterwards
i.e. money balanced, no dangling pointers

Isolation: in-progress transactions cannot see each other

ACID

Atomicity: a transaction happens or it does not

Consistency: a correct database is still correct afterwards
i.e. money balanced, no dangling pointers

Isolation: in-progress transactions cannot see each other

Durability: committed data survives power loss, water, ...

ACID Transactions

```
BEGIN;
UPDATE employees SET status = 'retired' WHERE name = 'Tony';
UPDATE customers SET rep = 'Bob' WHERE rep = 'Tony';
COMMIT;
```

Database condition: sales reps are not retired.
Users always see this.

ACID is useful

Database enforces rules.

Developers do not need to worry about partially complete transactions.

Failures are cleanly handled.

Implementing ACID

Provide illusion of serial execution:

- Row-level locking
- Concurrent versions/snapshots

Expensive:

- Maintaining locks
- Waiting for locks
- History of each item being edited
- Transaction reads old data, another updates: abort and redo

Replication/distribution and ACID

Relatively easy: single writer, no conflicts

Hard: multiple writers, possibly overlapping, conflicts

$\rightarrow$ Try to shard so that there are single writers i.e. GMail shards by user

Replication cost in ACID

ACID cost is $O(n^2)$ where n is number of replicas.
Naïve ACID implementation costs $O(n^5)$

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

This motivates BASE

- 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 did not 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

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

Not a model, but 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 does not 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

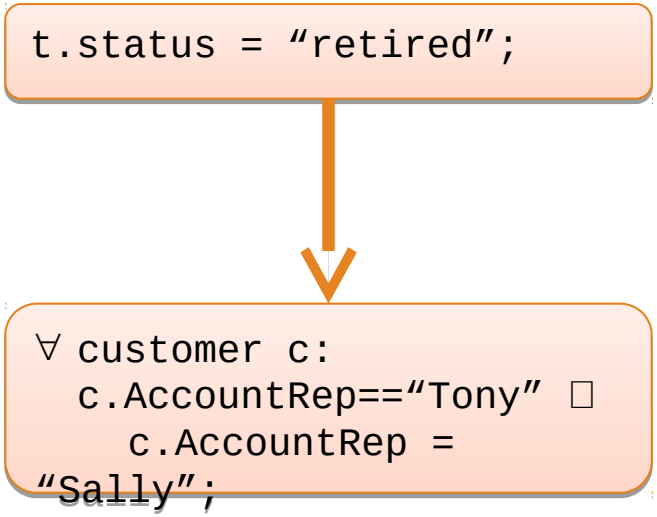
- Soft state service: Runs in first tier
  - Cannot store any permanent data
  - Restarts in a clean state after a crash
  - To remember data either replicate it in memory in enough copies to never lose all in any crash or pass it to some other service that keeps hard state

- 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

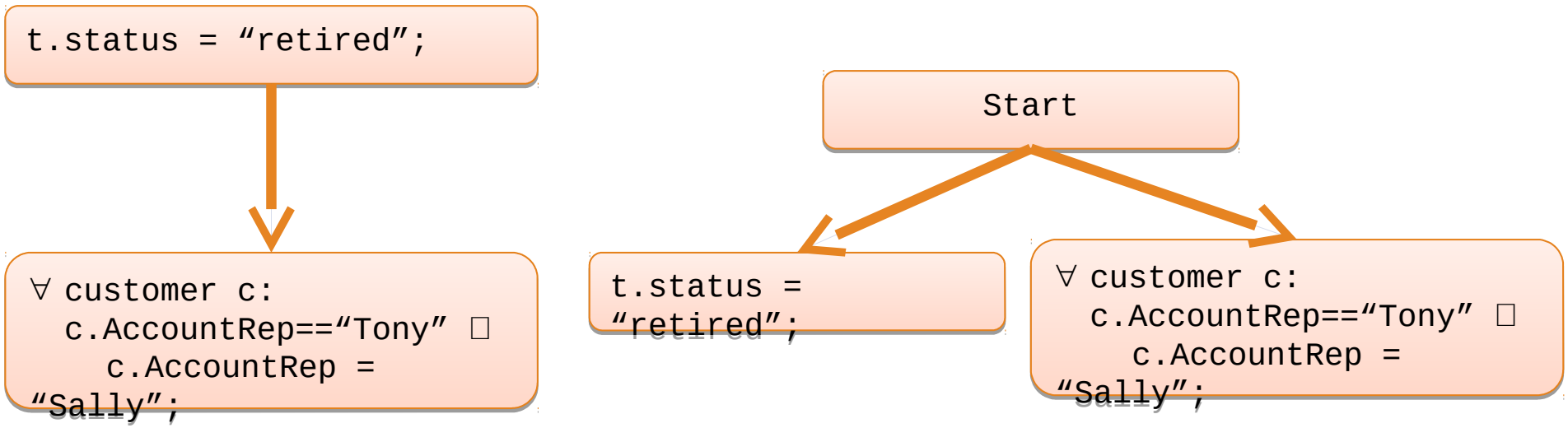
- Start with a transaction, but remove Begin/Commit
  - Now fragment it into steps that can be done in parallel, as much as possible
  - Ideally each step can be associated with a single event that triggers that step: usually, delivery of a multicast
- Leader that runs the transaction stores these events 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”
□
Commit;
```

BASE in action



- 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

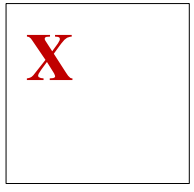
- Code was often much too slow
  - Poor scalability
  - End-users waited a long time for responses

- 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 should not hurt much, and if this makes eBay 10x faster, the speed may be worth the slight change in behaviour!

- 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 has posted a photo from the party
  - You may see an 

THE UNIVERSITY of EDINBURGH
informatics

AMAZON DYNAMO

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, was not scalable enough
  - Dynamo project created a new version from scratch

Dynamo approach

- Amazon made an initial decision to base Dynamo on a Chord-like Distributed Hash Table (DHT) structure
  - Recall Chord and its O(log n) routing ability
- The plan was to run this DHT in tier 2 of the Amazon cloud system
  - One instance of Dynamo in each Amazon data centre and no linkage between them
- This works because each data centre has ownership for some set of customers and handles all of that person's purchases locally
  - Coarse-grained sharding/partitioning

The challenge

- Amazon quickly had their version of Chord up and running, but then encountered a problem
- Chord was not very tolerant to delays
  - If a component gets slow or overloaded, the hash table was heavily impacted
- Yet delays are common in the cloud (not just due to failures, although failure is one reason for problems)
- So how could Dynamo tolerate delays?

The Dynamo idea

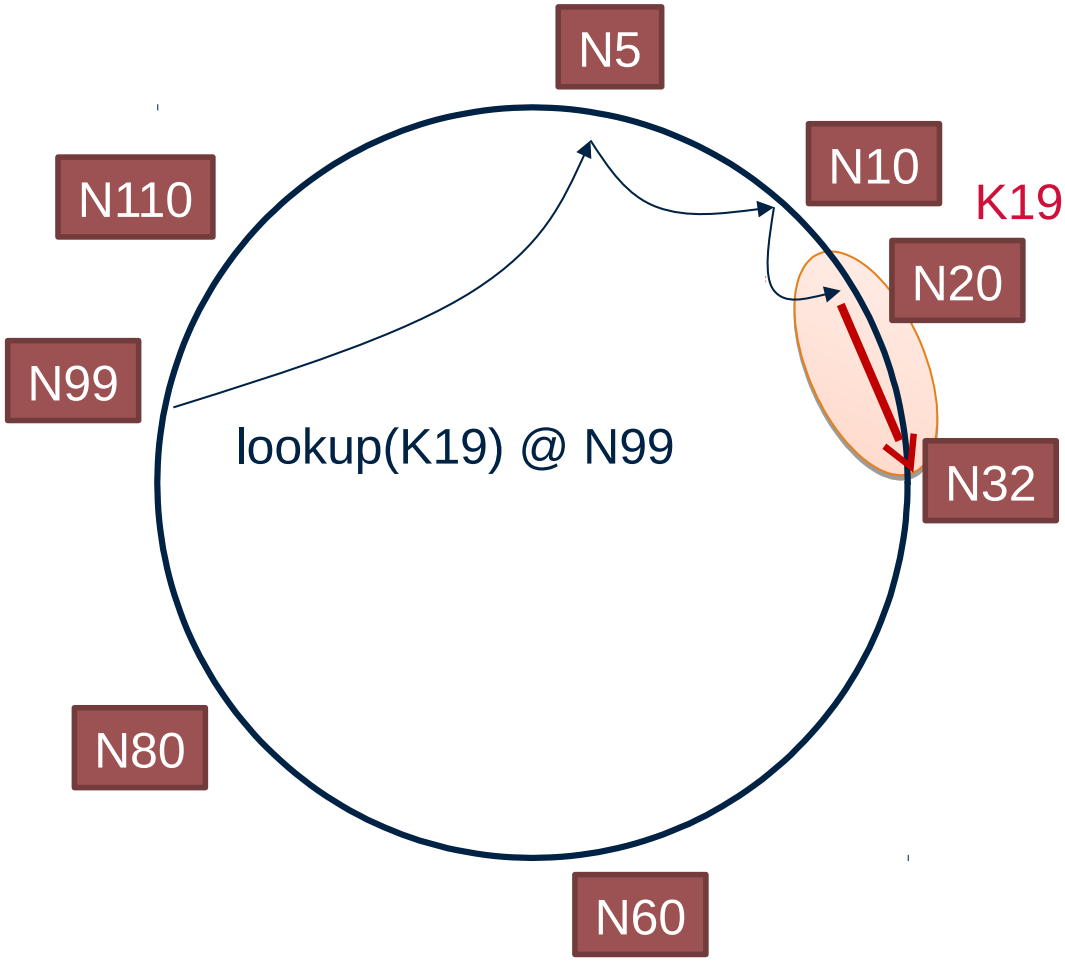
- The key issue is to find the node on which to store a key-value tuple, or one that has the value
- Routing can tolerate delay fairly easily
  - Suppose node *K* wants to use the finger table to route to node $K+2^i$ and gets no acknowledgement
  - Then Dynamo just tries again with node $K+2^{i-1}$
  - This works at the cost of a slight stretch in the routing path, in the rare cases when it occurs

What if the actual owner 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 does not 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 centres
- Then they do the Get/Put on the next node that actually responds even if this is the wrong one
  - Chord will repair

Dynamo example

- Ideally, this strategy works perfectly
  - 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 of misrouting (and miss-storing)

- 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

Summary

- BASE is a widely popular alternative to transactions
  - Basically Available Soft-State Services with Eventual Consistency
- Used (mostly) for first tier cloud applications
- Weakens consistency for faster response, later cleans up
  - Consistency is eventual, not immediate
- eBay, Amazon Dynamo shopping cart both use BASE