PNUTS (some material cribbed from RTM's MIT course)

Context

- servers at data centers in several locations around the world
    - why? both performance (low latency to nearest server) and availability (want to be able to survive data center outages)

- many different web applications sharing data
    - why? is Yahoo! – has a number of web properties (mail, IM, news, finance, calendar, group, …), and several data sets in common (e.g., user profile)
    - app might be local to a data center, or global across multiple data centers

- some strong "ilities" needed
    - incremental scalability: want to incrementally grow both the storage capacity and throughput of the service. contrast to non-incremental "forklift" upgrades

    - availability: care about providing uninterrupted service even in the face of component (or data center) failures. Paper claims that some applications want to be able to both read and write data during failures, even at the cost of data consistency. Q: when is this the case?

    - latency: typically want reads, and writes, to be processed locally if possible, only going to remote data centers when absolutely necessary

    - weak consistency: fully serializable transactions across all data is too strong, since it "too expensive" and would require relatively slow coordination mechanisms. They instead pick "per-record timeline consistency," which is really per-key sequential consistency.
        - implicit: parallel DBs are not the answer, because of expensive two-phase commit, focus on powerful but not terribly scalable joins, and the fact that reads are often remote
        - implicit: got to be careful about an architecture that includes locks, since they are expensive to hold, especially if have to wait for failure recovery to release a lock

Data model

Many tables
- composed of many records
    - with several attributes (columns)

- looks like relational DB so far...

Queries only by primary key, which must be unique
- insert, update, delete must specify exact primary key
- lookup can specify PK range, for "ordered" tables
- mostly a "put/get" or "key/value" interface (maybe you can scan a range of PKs)

What *doesn't* the query model support?

- joins, filters (only PK range)
- any other fancy SQL construct like SELECT ... IN (SELECT ...)
- basically you get no queries other than single row fetch/update

in practice, do we think put/get is all web apps ever want/need?

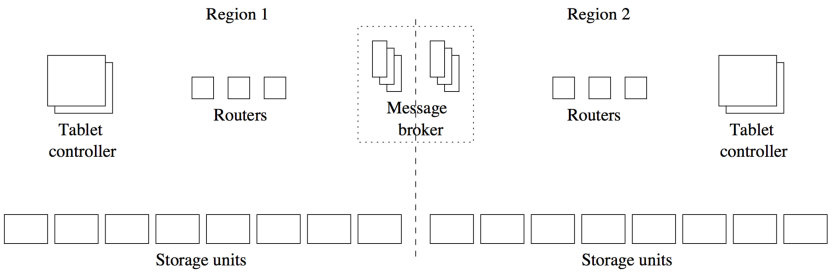
- it's enough for "what is bob's email address"
- but not enough for
  - recommend me a book that my friends like
  - show me comments on this blog article

so how would you write web apps on top of PNUTS?

- issue lots of queries; e.g. do their own equi-join
- don't obsess about correctness
  - maybe just a book that *some* of my friends like
- keep multiple copies keyed in different ways
  - comments sorted by user
  - comments sorted by commented-on article
  - basically, need to build your own ad-hoc index

Architecture

A region is basically a site. Each region contains a full copy of the data.



Data is partitioned over SU's by primary key. Routers know the partitioning; truth of the partitioning is stored in the tablet controller, which is a single master/standby pair.

A read is serviced locally within the cluster:

- the app sends the PK to the router to learn which local SU serves that key
- the app sends the read to the SU; SU replies with the data

Updates are more complicated

- an update has to propagate to all sites!
- Q: why not just have applications send updates to all sites?
  - what if concurrent updates to the same record by different app instances?
    - if updates are applied in different orders at different replicas, get permanent divergence, which is bad
  - also, might be hard to ensure reliable delivery, unless apps are sophisticated
    - can also lead do divergence through missing updates

- idea: each record has a “record master”
    - all updates for that record start with the record master site
        - implies writes may need to cross the WAN some of the time; have incentive to induce locality between writers and record masters
    - the responsible SU issues updates one-at-a-time per record
        - this is what gives you per-key sequential consistency
        - record master and SU together decide on order of updates to a particular key
    - each record has a hidden column indicating which site is the record master

- app wants to update some columns of a record, knows PK
    - 1. sends PK and update to local router, router decides PK on SU1
    - 2. router forwards to SU1, which knows record master for PK: SI2
    - 3. SU1 sends update request to router at SI2
    - 4. router at SI2 forwards update to local SU2 for PK
    - 5. SU2 sends update to local Message Broker (MB)
    - 6. MB logs to disk, says "OK"
    - 7. SU2 updates the record locally, sends vers # to client
    - 8. MB sends a copy to router at every site
    - 9. every site updates local copy

Per-key transactional update provided by record-level test-and-set-write(version, new value)
- master rejects the write if current version# != version#
- so if concurrent updates, one will lose and have to retry

No other atomicity provided – can’t do multi-key atomic updates, which is kind of problematic for complicated applications.
- may see state in the middle of someone's complex update

Fault tolerance

SU crashes and restarts, or loses net connection for 30 seconds
- it may have missed some updates!
- MB system will keep sending them

SU loses disk contents
- now that site is missing some data!
- Q: can it serve reads from local web servers by forwarding to other sites?
    - unclear
- need to restore disk content from SUs at other sites
    - subscribe to MB feed, and save them for now
    - copy content from SUs at other sites
    - replay saved MB updates

MB crashes after accepting update
- logs to disks on two MB servers before ACKing
- recovery looks at log, (re)sends logged messages
- record master may re-send an update if MB crash before ACK
    - record version #s will allow SUs to ignore duplicate

record master crashes and recovers during update?

- log to disk before ACKing

record master crashes permanently

- might lose most recent update (only in log of crashed machine)
  - may take a while to restore master's content
- need to switch to a new master to let new updates proceed!
- how to switch while preserving order?
  - maybe consult RM's MB for last version #
  - or wait for MB system to finish with all RM's updates
  - now we know version # and record content for new record master to start with
- site of failed RM probably has to be in charge of this
- don't want two different sites taking over!

record master's entire site loses network connection

- can other sites designate a replacement RM?
  - no: original RM may still be processing updates
  - don't want *two* RMs!
- do other sites have to wait indefinitely?
  - this is what the end of section 2.2 is about -- Dynamo envy, wishing for relaxed consistency with eventual convergence

Evaluation

section 5.2: time for a single insert

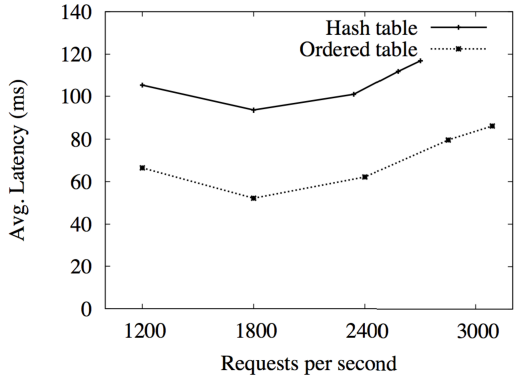
- RM local: 75.6 ms
- RM nearby: 131.5 ms
- RM other coast: 315.5 ms

- what is 5.2 measuring? from what to what?
  - maybe web server starts insert, to RM replies w/ new version?
  - probably not time for MB to propagate to all sites

- where does the time go?
  - 40 ms in RM (disk, locking, &c)
  - what about the rest of the 75.6?
  - why 60 ms extra to talk to RM on the same coast? too much for one RTT...
  - why 240 ms extra to talk to RM on other coast?

5.3: vary the offered load, measure the response latency

- go over what we'd expect for this graph
  - increase latency up to capacity, then unbounded latency
- what do they show?
  - kind of weird, and clearly not driving to capacity



5.4: showing writes are more expensive than reads, as expected

- involves MB, propagating updates to all sites, etc.

stepping back, what were PNUTS key design decisions?

1. primary-backup replication
   - all write ops to primary, it picks order, sends to backup
     a. only use backup if sure primary is dead
   - pro: keeps replicas identical, enforces serial order on updates, easy to reason about
   - con: hard to make progress if primary is up but partitioned
   - alternate design: eventual consistency (Dynamo):
     - always allow updates,
     - tree of versions if network partitions,
     - readers must reconcile versions
2. put/get query model (no SQL)
3. not transactional