Advanced Topics in Computer Systems, CS262B
Prof. Eric A. Brewer

OceanStore (Pond)

April 8, 2004

I. Background

DHT is just part of the story. A useful application is wide-area reliable storage

- Two classes of nodes: inner ring and everyone else
- Don't trust any single node or router => byzantine agreement for updates
- sequence of read-only version => never invalidate a cache
- Archive forever all versions
- Reads easy, writes hard: caching works for reads, writes must be serialized (byzantine)

Moore's Law for disk is non-existent. (What exactly is Moore's law?)

- ... but it is true that disk capacity is improving at about 100%/year

Global names:

- AGUID: the name of the data object, includes all of its versions, owner's public key is part of the name (name changes if the owner changes!)
- VGUID: the name of the root block of the current version
- BGUID: the name of a data block -- just a secure hash of its contents

Basic read: lookup AGUID to get [AGUID, VGUID, timestamp, seq number](signed)

- if happy with timestamp or seq number, then use VGUID to get data

II. Design Issues

Primary replica:

- coordinates and serializes concurrent writes
- potential single point of failure
- solved by BFT

Two-tier solution:

- inner ring manages serialization, archiving, and BFT
- secondaries just serve blocks (DHT from BGUID -> block data)
- multicast tree to spread updates from inner ring out to secondaries (more below)

BFT:

- 3f+1 members of the inner ring must agree on serialization

- note that view change not implemented!
- which 3f+1? presumably a deterministic function of the AGUID
- the BFT members, B, set timestamp and version number and VGUID, and sign it -> "heartbeat"
- you can't really know that you have the latest version
  - best case: send nonce to B and they will include it with the latest heartbeat
  - ... but they could do an update before you receive the message
  - solution: optimistic concurrency (apply an update if your dependent reads are still valid)

Optimistic concurrency (like Bayou):

- updates have predicates: verify predicate before execution, else retry
- reads can have predicates too (useful for ensuring a multi-object invariants)
- how to do a multi-object transaction?
  - just as in DBMS with optimistic concurrency
  - 1) read all data and record version numbers
  - 2) compute new versions
  - 3) 2PC to apply updates (with predicates to verify read versions) at each object
  - may fail => retry
  - may retry forever => livelock
- the goal of complex writes is to reduce the frequency of retries
  - ex: append should always work (even if the data has changed since you read it)
  - most changes may not effect the correctness of the write; the predicate should be narrow enough to increase you chances of success (e.g. CVS)

Who are the members of B and how does it change over time?

- problem with BFT: the f faulty nodes are for the lifetime of the system
  - long-lived systems accumulate faulty nodes
  - need to change members over time (more than just a view change)
- B as a *group* is the "primary replica" -- the group needs to sign updates
- Part 1: use symmetric keys among members of B (all pairs), but this doesn't work for secondaries (too many of them to have sym keys with each one) [this may not be true]
- Part 2: use public key to sign heartbeats => need a public key for B as a group
- Part 3: use SHA to make blocks self certifying => no interaction with inner ring to verify a block (only to verify metadata like current version number)
- How to get a public key for B?
  - proactive threshold signatures
  - idea: break a public key into L private shares, such that any k of the L can sign something with that key (there is no single private key!)

- choose L = 3f+1 and k= f+1, so that we know that we need byzantine agreement to sign something
- “proactive” => can create a new set of L shares whenever we need to (such as when we change membership), k of the old set still work, as do k of the set, but you can’t mix them!
- Since k>f, if we change sets, there can be at most f using the old set, which is not enough to sign
- key point: after changing the membership, we have new key shares for the new members, but the public key remains the same!
- this is not completely implemented in Pond....

Need to have duplicates in the DHT namespace (and they really should be independent)

Can’t really do this directly with Chord: route to the ID and then the replicas are successors

Archiving:

- key idea: erasure code all updates to save storage over time
- reads are very expensive (have to reassemble blocks), but caching works well
- performance suggestion: don’t archive immediately (in the style of AutoRAID)
    - ensure that at least a few copies exist (this is *easy* with a dissemination tree)
    - if the version is still interesting later, archive it at that time
    - this reduces the cost of the update and makes archiving a background task
    - open question: do we want *all* versions? probably namespace specific and most often a periodic snapshot would be fine (which is easy to do in a time-travel system!)
- is the current version faster to read than an old version?

Dissemination tree:

- idea: push out updated heartbeats and VGUID blocks via a multicast tree
- is this a good idea?
- only if you have lots of reads...
- a write-mostly system might prefer non-local reads, but with later versions and thus more successful updates
- dissemination tree is expensive

Is a public key signature a good idea for blocks?

- very expensive for small updates -- 7x all of the overhead put together
- might be able to use sym keys for small groups of readers (which is very common)
    - e-mail has typically one reader
    - files that have only user or group access might also prefer sym keys

Access control:

- deeply tied into how you would do dissemination and signing!
- probably want some options either at object creation time, or at namespace creation time
- how are different namespaces handled?