Advanced Topics in Computer Systems, CS262B

Prof Eric A. Brewer

Distributed Data Structures

Gribble et al.

Goal: new persistent storage layer that is a better fit for Internet services

- separate service logic from durability, availability, consistency issues
- interface is that of a data structure, not a SQL query (intentionally navigational)
- automate replication and partitioning across the cluster
- automate recovery and high availability
- CAP: CA (not P) -- depends on a machine room with a redundant network

Nit: abstract's perf numbers have too much precision! (should have been limited to 2-3 significant digits)

Alternatives for persistent storage:

- DBMS: chooses C over A, interface is inherently slow: parse and plan each query.
- Enterprise Java Beans (EJB): map Java objects onto tables. Typical use has an array of objects map to a table with one row for each object; row stores the serialized bytes for that object => expensive copy to get the object in and out of the database. Also, storage is never local and there is little control over partitioning and replication (which nodes). Should be 10x or more slower, but no reliable data.
- File system: also has an expensive interface and very hard to provide fine-grain atomicity. (primary atomic operation is file rename, which implies copying the whole file). Fine choice for large objects, but not for small ones...

Internet requirements:

- extreme scale: billion request/day
- high availability
- unknown but potential fast growth -- must be able to add capacity quickly
- overload will occur -- need graceful degradation

Cluster properties:

- Incremental scalability -- add nodes over time
- Potential for high availability -- but needs to be written!
- Natural parallelism for both I/O and CPUs
- High BW, low-latency, *partition free* network (CAP)
- Machine room properties: security, administration, reliable power, AC, networking (these generally don't apply to P2P systems in practice)

DDS design decisions:

- hash table interface: 64-bit keys, byte array values, put/get/delete elements, create/destroy hash tables

- operations are atomic, sequences of operations (trans-actions) are not
- two-phase commit across all replicas for consistency of writes
- read any replica (=> higher read throughput than write throughput)
- event-driven design for high concurrency. this was a mistake and led to Capriccio. It is being rebuilt from scratch in C over Capriccio. (see Java problems below)
- assume no network partitions (CAP)

Partitions:

- key idea: break hash tables into many small fixed size "partitions"
- replicate partitions (not tables); replica groups are a set of partitions with the same data
- recovery: recover each partition independently
- small partitions mean that you can lock the whole partition during recovery (other partitions operate independently)
- also implies that you can lazily recover each partition, or you can be proactive.

Fault tolerance

- DDS library: handles two-phase commit across replica group.
    - commit occurs when at least one replica receives a commit message
    - failure before this and all replicas will time out and discover there was no commit by contacting each other
    - failure after this and the replicas will learn about the commit from the member that recorded it, and will then all commit
- Brick:
    - failure before commit will cause 2PC to fail and everyone aborts
    - failure after agreement to commit, but before commit: other replica's commit, this node will get the new value during recovery
- Other service code:
    - if all durable state is in the DDS, this is recovered automatically on restart
    - soft state can be rebuilt (e.g. caches)
    - if local durable state (not in DDS), then service is on its own (e.g. might store some data in a DBMS also)

Data Partitioning Map:

- maps HT key to partition id
- uses a trie so that it is easy to split or merge partitions as needed (increment data repartitioning!)
- replicated on all DDS clients, eventual consistency (stale data causes a repair and a retry); staleness is detected by comparing a hash of the maps to the current values.

Replica Group Map:

- maps partition id to the set of replicas
- replicated on all DDS clients, eventual consistency as above

- o writes are 2PC to all replicas
- o reads go to one replica. which one? want to spread out reads in the same way to maximize effective cache space!

Recovery:

- o Node failure takes down all bricks on that node; brick failure takes down all partitions on that brick
- o a recovering brick must catch up all its partitions
- o key idea 1: allow a brick to say "no" -- this simplifies the code greatly! DDS library will retry in a bit. "NO" means that brick can be inconsistent for a while (but not too long)
- o all committed operations must use up-to-date maps (DP and RG), else retry
- o bricks catch up one partition at a time: just copy that partition from another replica.
- o Updates to a partition stop during recovery, but this is OK but partitions are small!
- o Must decide how proactive to be in recovery; OK to be completely lazy (wait for partition to be accessed, but hurts latency)
- o see recovery graph (figure 8)

Performance:

- o great scalability, availability, graceful degradation for reads (less so for writes)
- o Problems: Java GC can screw up performance! One a node gets behind it needs more memory (for growing queues), so it tends to get worse and worse! Need admisstion control, control over GC, or probably both. SEDA would probably fix this...
- o Events make this much worse! Event driven systems (in Java) create a lot of garbage, as events are just passed up through layer and then GC'd. Threads would have most of this state on the stack and thus create very little gargage... (this would also be a bad place for functional languages, which need to copy a great deal)
- o serious problems with Java: GC issues, extra copying, asynchronous I/O... We later fixed async I/O, but the other two issues remain. In general, Java is too high level for this kind of work...
- o