FAB: Federated Array of Bricks
Yasushi Saito, et al. HP Labs
ASPLOS 2004

What are they trying to build?
  A fault-tolerant storage system.
     scalable, flexible, built from cheap commodity disks, CPUs and network cards.

1. Data structure

Diagram of connected bricks.
One brick with global volume-->seggroup mapping
               timestamp table

-One logical disk with many logical volumes.
-brick = physical machine
-Any brick can be a coordinator for a request.
-All bricks have the global map of logical volume to seggroup to bricks.
This mapping is Paxos replicated.

-Each seggroup is replicated under the same policy at the same places.
They use voting-based replication for seggroups.
-Also erasure codes the block/seggroup?

Timestamp table --> stores timestamp info for recently modified blocks.

2. Replication

The goal of FAB is to tolerate failures such as power failures to a CPU,
network partitions, and disk failures.

To achieve fault-tolerance, FAB replicates each block on several bricks.

Diagram of two bricks A, B want to tolerate the failure of one brick
   -a write returns after finished on one brick.
   -read returns after finished on one brick.
Is there anything wrong with this scheme?

What technique does FAB uses to guarantee linearizability?
   Voting, Quorums -- each read/write operation has to complete at a
                      majority of the bricks. Read picks the latest value.
   Each block has 2 persistent timestamps: ordTs and valTs

What is the purpose of valTs?
   If there is no valTs or ordTs:

   Diagram of 3 bricks: A   B   C
                        v1 v1 v1
   coordinator1   v2 v2
   coordinator2   v3 v3 v3

v2

valT prevents writes of old values from overwriting new values.

What is the purpose of ordTs?

If there is only valTs and not ordTs:
Diagram of 3 bricks: A B C
v1 v1 v1
write1 v2
read1 from A and B --> v2
read2 from B and C --> v1

Describe write and read protocol with ordT.

ordT prevents concurrent reads and writes from seeing value out-of-order.

So, a block has two timestamps and they need to be persistent.

For performance, FAB keeps timestamps in NVRAM.
What is the potential problem in keeping timestamps this way?
-Run out of space. Takes 48GB of timestamps for 1TB of data.
What is FAB's solution to this problem?
-The timestamp table in NVRAM keeps only timestamps + block numbers of
recently modified blocks.
-The coordinator sends a GC (garbage collect) message to the bricks
after all bricks acknowledged an update. The bricks then removes the corresponding
timestamp entry from the table.

What if want to read or write a block that does not have an entry in the
timestamp table? How can we use the pseudocode in figure 4?
For garbage collected blocks, assign valTs = ordTs <-- time of latest GC

In pseudocode in Figure 4, in proc read, in the order phase,
why does it use current timestamp instead of ordTs?
Because there maybe incomplete order phases, so want to pick a timestamp
that is greater than any incomplete timestamp.

****
Write takes 2 rounds
1. The coordinator writes the block's ordT on each replica brick, wait for a majority to respond, and return.
2. The coordinator writes the block's value and update valT on each brick.
If any of those steps fail, ABORT.

Read normally takes 1 round
1. If majority returns with same timestamp, return val.

Otherwise, read takes additional 2 rounds.
  1. Write the latest timestamp ts to ordT on all replica.
  2. If majority succeeds, newval <-- val with highest valT from replica
  3. Write <val, ts> to all replica. Return successfully if majority said yes;
otherwise ABORT.

How does ABORT enable strict linearizability?
****

3. Reconfiguration

To tolerate long-term or permanent failures, FAB uses reconfiguration.

Why do we need view change?
   Diagram three nodes, one dies
      Can we still operate?
        What is value was written only to A?
        Yes, we still have a majority. 3/2+1=2.
      Can we stay like this forever?
        No, eventually NVRAM with fill up because cannot garbage collect the
      timestamps.
      Any other reasons?
        If another brick dies soon after the first brick, we might not have
      all the updates on the remaining brick. Therefore, need to reconfigure
      so that 1) both nodes synchronize data with one another
              2) prevent NVRAM from filling up.
        Now, have 2 bricks in the configuration. Updates will go to both bricks,
      so if one brick dies, still have all updates on the remaining.
        Need to reconfigure to increase bricks eventually to maintain MTTDL.

Can we use the same sync pseudocode in figure 9 to synchronize newly added bricks?
  No, start from empty disk.
What if the newly-added brick used to be in the configuration and still has data from before?
  No, might have garbage collected timestamps in the current configuration.

Reconfiguration protocol works similar to Paxos.
   In this specific case, requires that accept a proposed view only if it intersects
a majority of the current view and each of the ambiguous views.
   What is the reason for this requirement?
    1) Intersect with current+proposed view so that only one new view is chosen
    2) Intersect with current view so that the synchronization works,
  because need at least one brick to have the newest value.

What if a dead node that has been reconfigured out comes back online?

Diagram of 3 nodes. A B C. C dies. Reconfigure to A B.
  Then C comes back online. If C answers queries, can it mess things up?
     No, because A and B are still the majority.

  What if reconfigure to add another node D to A B, then network partitioned so
than A D and B C are in different partitions? Can it mess things up?
  No, if the protocol includes a configuration number with each reply.
  Paper did not mention that.

****
3.1 View change

Three rounds. A brick (in the same seggroup?) that detects another brick failure
becomes the leader. It computes a candidate view and witnesses = vote view.

Round 1: Leader sends vote view to bricks in vote view.
Bricks received the view and adds it to the ambiguous view.
What do witnesses do?

Round 2: If leader receives acceptance from majority (including witnesses??),
it proposes the candidate view to the bricks in the candidate view.
A brick accepts the view iff the view contains a majority of the current view
and each view in the ambiguous view list.

Round 3: If leader receives acceptance from ALL bricks in the candidate view,
it sends a message to the view to update its current view to the new view and
delete the ambiguous view list.

Why does the candidate view have to intersect a majority of each ambiguous view?

3.2 State synchronization

Round 1: Sync leader finds all the blocks that are in the timestamp table at
each brick that replicates a segment.

Round 2: The leader sends a SyncPoll message for each recently modified block in
the oldView. Upon receiving info from an m-quorum (same as majority?) pick the
maxOrdTs, maxValTs and maxVal.

Round 3: The leader sends this info the bricks in the newView. When received reply
from m-quorum, declare synchronization done.

Optimizations

1. If every quorum in the oldView is a superset of another quorum in the newView,
then don't need to sync.

2. Separate blocks that must be synced before removing the oldView, and the ones
that may be synced after oldView is removed. The "must" blocks are the ones whose
set of bricks that have been successfully written to is not a quorum of the newView.
The "may" blocks are the ones whose set of bricks that have been written to is a
quorum of, but not a superset of, the newView.
****
****