CSE535: Distributed Systems (Fall 2021)
Scott D. Stoller, Stony Brook University
Problem Set 2 (version 2021-11-26), Due 8 December 2021 - Solution

INSTRUCTIONS: Do this assignment with your project teammates. **Justify your answer for every problem**, except where directed otherwise. All problems have the same weight. Exactly one member of each team should submit the assignment on Blackboard by 11:59pm on the due date. The pdf file should be formatted for 8.5"×11" paper and should include the team name and names of team members at the top of page 1; also, the filename should include the team name. I recommend that each team member work individually on all problems, then the team can meet to pool everyone's thoughts and organize the final write-up. This will help everyone be better prepared for the exam.

Problem 1: GFS

Section 4.5 of the Google Filesystem (GFS) paper implies that chunk version numbers are incremented only when a lease is granted by the master, not when each mutation is serialized by the primary. Consequently, it seems that a secondary that successfully increments its version number for a chunk, and then misses a mutation because of communication failures or a crash failure, could still have the correct version number for the chunk. How is this issue dealt with in GFS? For example, what would prevent a client from reading a chunk from such a secondary?

For simplicity, consider only write operations. Ignore append. The situation for append is complicated by GFS's weak semantics for append (recall that GFS re-tries a failed append at a different offset).

Answer: Recall from Section 3.1, step 7, that "In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write." In case a secondary replica A for a chunk C does not acknowledge completion of a write to a chunk C to the primary (step 6 in section 3.1) during the retries of steps (3)–7, presumably the primary (or the client) reports this to the master. The master should then remove A from its list of chunkservers with replicas of C, and replicate A on another chunkserver (to maintain the required number of replicas). The client may then retry the write from the beginning and hope to succeed with the new set of replicas of C. Although this behavior is not stated explicitly in the paper, it seems to be a reasonable approach.

Removing A from the list of replicas of C ensures that future operations on C will not use A's copy (e.g., a client that wants to read C, and asks the master for the current set of replicas of C, will not be given a set including A), except possibly for operations by clients that have unexpired cached meta-data for C, which is undesirable but tolerable.

Although the master removes A from the list of chunkservers with replicas of C, there is a danger that A could rejoin that list without updating its copy of C. Suppose A crashes immediately after incrementing its version number for C on stable storage, and then A misses an update to C while A is down, e.g., the client performing the above write retries the write from the beginning, gets the new list of replicas of C in step 1, and successfully performs the write, without changing the version number of C. This seems possible, because no statement in the paper implies that the master needs to issue a new lease for C during this process; a lease seems to be valid as long as it has not expired, and the primary to which it was issued is responsive. Suppose A then recovers and announces to the master that it has a replica of C with the current version number. It seems that the master might then add A to the list of chunkservers with replicas of C. This would be incorrect.

One might claim that this is not so bad, because A would provide stale results only until the next time the version number is incremented. However, the version number might not get incremented for a long time, if, after the current lease for C expires, no more writes to C occur for a long time.

Here is a simple and inexpensive way to prevent this problem. When the master learns that a secondary did not acknowledge completion of a mutation on chunk C, it immediately revokes the current lease for C and notifies the primary of this. Any future update to C, including the retry from the beginning by the same client, will require issuing a new lease with an incremented version number for C, so A's copy of C will be considered stale. The downside of this approach is that A's copy is considered stale even if A did not miss any updates. Such unnecessary invalidations are probably rare enough in practice to be acceptable. Again, the paper does not explicitly describe this behavior, but it seems reasonable.

Note that, if the master does immediately revoke the lease for C when A fails to acknowledge an attempted write, then removal of A from the set of replicas of C could be deferred until just before the master issues the next lease for C. this is safe because C cannot be updated before that, and A's copy is not out-of-date until C is updated. (If the master does not revoke the lease, then the master might not be contacted during the next write to C and hence wouldn't have the opportunity to remove A at that point.) However, it is may be better to pro-actively remove A and create a new replica of C in advance, so the next write is more likely to succeed.

Problem 2: Bigtable

Write high-level pseudo-code for the tablet location lookup algorithm in the Bigtable client library. The inputs to the algorithm are the table and row key. The output is the identity of a server with a replica of the tablet containing the specified row of the specified table. Remember that some relevant information might already be cached at the client. Your pseudocode should reflect failure handling when a queried server does not reply. Consider only failure of tablet servers; do not consider failure of the network, the master, or Chubby.

Answer: The tablet location lookup algorithm is described in Section 5.1 of the Bigtable paper. Let T denote the specified table identifier. Let k denote the specified row key. Note that the tablet-location cache, like the METADATA table itself, should "store the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row".

1. Look in client's tablet-location cache for an entry with the location of the tablet containing the specified row k of table T (note: To identify the desired tablet, the row key k is compared to the end rows of tablets for T). If an entry for the desired tablet is found, return the tablet identifier (i.e., the table name and the tablet's end row) and location (which is a tablet server). The client will try to fetch the specified row from that location. If that tablet server does not reply or does not have the tablet, then delete the cache entry for that tablet and go to the next step.

2. This step is reached if the tablet-location cache does not contain an entry for the tablet with the desired row. Look in the client's tablet-location cache for the location of the METADATA tablet with an entry for the desired tablet. If such an entry is present, query that server to obtain the tablet identifier and location of the desired tablet, cache the answer, and go to the previous step. If that server does not respond or does not have that METADATA tablet, delete the cache entry for that METADATA tablet, and go to the next step.

3. This step is reached if the tablet-location cache does not contain an entry for the relevant METADATA tablet. Look in the client's tablet-location cache for the location of the root tablet. If such an entry is present, query that server to obtain the location of the relevant METADATA tablet, cache the answer, and go to the previous step. If that server does not respond or does not have the root tablet, delete the cache entry for the root tablet, and go to the next step.

4. This step is reached if the tablet-location cache does not contain an entry for the root tablet. Query Chubby to obtain the location of the root tablet, cache the answer, and then go to the previous step.

The above description is not quite as detailed as pseudocode, but it is close.

The above description does not reflect that "the client library prefetch[es] tablet locations: it reads the metadata for more than one tablet whenever it reads the METADATA table." This is easy to incorporate.

The client could report unresponsive servers to the master, to hasten failure detection. This is probably a good idea but not essential, since the master will detect those failures itself using the failure detection mechanism described in Section 5.2.

The above description does not check whether cached tablet-location information has expired, because the paper says "stale cache entries are only discovered upon misses' (which we expect to be infrequent because METADATA tablets should not move very frequently)." In other words, cached tablet-location information never expires.

Note that locality groups do not affect tablet location: tablets for all locality groups for a specified key range of a specified table are located on the same tablet server.

Problem 3: Megastore

This question is about Megastore. (A) As part of processing a snapshot read request, does a replica need to ask its coordinator whether it is up-to-date for the entity group being read? If the paper directly addresses this question, your answer should quote the relevant passage. Otherwise, you should propose an answer and explain why it is the most reasonable approach. (B) Describe different kinds of scenarios in which a snapshot read and a current read would return different results. Describe as many different kinds of such scenarios as you can think of. Hint: there is at least one kind and at most two kinds. Note that a scenario should be described as a *sequence of events*.

Answer:

(A) The paper does not directly address this. The most reasonable answer is "no", for the following reasons. (1) The description of snapshot read uses the phrase "last *known* fully applied transaction" to emphasize that the information is relative to a specific entity's knowledge (the description of current read does not contain "known"). The most sensible interpretation is that this entity is the local replica. (2) The result of a snapshot read presumably includes "the timestamp of the last *known* fully applied transaction", so the client knows how up-to-date the result is. If the client decides the timestamp is not sufficiently recent, it can perform a current read. (3) If a snapshot read required bringing the local replica up-to-date, then the cost of a snapshot read would be similar to the cost of a current read, and supporting both operations might not be worthwhile.

(B) Scenario 1: The paper says "For a snapshot read, the system picks up the timestamp of the last known fully applied transaction and reads from there, even if some committed transactions have not yet been applied." As this indicates, a snapshot read will return different results if there are some committed transactions have not yet been applied, because the current read will apply them first. here is a scenario based on this. Client c1 initiates a write to entity group g. the write completes step 3 in the algorithm in sec 4.6.3, with a quorum of replicas accepting the write, so the write has committed. let S be a replica in the quorum that accepted the write. before S receives the Apply message sent in step 5, S receives a snapshot read request from client c2. this snapshot read returns a different value than a current read would. note that this scenario can occur even if all replicas accept the write in step 3.

Scenario 2: Client c1 initiates a write to entity group g. the write completes the entire algorithm in sec 4.6.3, with a quorum of replicas participating, and one replica S not participating. S's coordinator is notified that S is out-of-date for the EG. S receives a snapshot read request from client c2 and returns its stale data, because as discussed in part A, it does not check with its coordinator and learn that it is out-of-date for g. a current read would cause S to learn from its coordinator that it is out-of-date and then bring itself up to date before reading the data and replying to the client, so it would return a different value.

Problem 4: Chord

Write pseudo-code for n.stabilize(), n.find_successor(id) and n.closest_preceding_node(id) for the version of Chord that uses successor lists of length r. Include explanatory comments in your pseudo-code. Discuss any interesting design decisions, e.g., if there is a trade-off between the cost of stabilization and the speed with which knowledge of changes propagates through the system.

Answer:

n.stabilize(). Here is a first attempt based on the description on page 24 column 2 of the Chord paper. This pseudo-code uses Python slice notation and 0-based list indexing.

```
n.stabilize():
  while True:
    try
      # update successors[0] if a node was added between n and its successor
      x = successors[0].predecessor
      if x in (n, successors[0]]: successors[0] = x
      # update the rest of n's successor list based on n's successor's successor list.
      successors[1:] = successors[0].successors[0:r-1]
      # tell my successor about myself
      successors[0].notify(n)
      return
    catch timeout waiting for reply:
       # left-shift contents of successor list, and try again.
       successors = successors[1:]
       if isEmpty(successors): return  # give up
```

This pseudocode, like the pseudocode for n.stabilize() in the paper, contains communication expressed as accesses to remote variables: a variable access of the form n.v, where n is a node, means to ask n for its value of variable v, and wait for the result. This is analogous to the notation for RPCs, e.g., n.notify(...).

Consider what happens if x in (n, successors[0]] holds and then x, which has been assigned to successors[1], does not respond when we ask it for its successor list. A reasonable approach is to forget about x, restore the previous value of successors[0] (i.e., the node that told us about x), and then continue (i.e., ask that node for its successor list). Does the above pseudocode do this? Typically yes but not always. Let n_0 and n_1 denote successors[0] and successors[1], respectively, when n.stabilize() is first called. Let n' denote n_0's predecessor, which gets assigned to successors[0], overwriting n_0. When n' fails to send its successor list, the above code will go to the catch block, left-shift the successor list, thereby discarding n' and setting successors[0]=n_1, ask n_1 for its predecessor, which typically will be n_0, in which case the code will set successors[0] back to n_0, ask n_0 for its successor list, etc., as desired.

However, if some node n'' has been added between n_0 and n_1, then n_2 will report that n'' is its predecessor, and n will incorrectly take n'' as its immediate successor in this call to stabilize. This is undesirable and can be avoided with negligible additional cost, by querying x for its successor list (and thereby also checking that x is alive) before assigning x to successors[1]. Note that this is an example of one of the kinds of corrections discussed in the slides on Verification of Chord: "Second, before incorporating a pointer to a node into its state, a member usually checks that it is live. *[In other words, she added such checks.]*"

Here is one way to do this; there might be a simpler way.

```
n.stabilize():
  while True:
    try
      # update successors[0] if a node was added between n and its successor
      x = successors[0].predecessor;
      tail = null
      if x in (n, successors[0]]:
         try
            tail = x.successors[0:r-1]
            succcessors[0] = x
         catch timeout waiting for reply:
            skip  // note: tail and successors[0] are unchanged if x does not respond
      if tail == null: tail = successors[0].successors[0:r-1]
      # update the rest of n's successor list based on n's successor's successor list.
      successors[1:] = tail
      # tell my successor about myself
      successors[0].notify(n)
      return;
    catch timeout waiting for reply:
       # left-shift contents of successor list, and try again.
       successors = successors[1:]
       if isEmpty(successors): return  # give up
```

The pseudocode to update the immediate successor can be iterated, i.e., replace

```
         x = successors[0].predecessor;
         if x in (n, successors[0]] then successors[0] = x;
```

with

```
         x = successors[0].predecessor;
         while x in (n, successors[0]]
            successors[0] = x;
            x = successors[0].predecessor;
```

(This change is shown for the r-successor versions of stabilize(), but is also applicable to the original version of stabilize().) This change causes information to propagate more quickly if multiple nodes have been added between n and its current successor.

The successor list could be updated more aggressively by requesting information from all r successors, instead of just the immediate successor. However, this significantly increases the cost of stabilize() in all cases and provides no benefit in the common case that there are no changes to the successor list. Therefore, the additional cost is probably not worthwhile, even though it would increase fault-tolerance (in some cases).

n.find_successor(id) and n.closest_preceding_node(id). Recall that successor[0] = finger[1]. The remaining successors could be equal to or interleaved with the remaining finger entries. The simplest approach is to search in both lists separately, and use the closer node among the two results. A plausble small optimization is to check whether the given id precedes the last entry in the successor list, and if so, return the closest preceding entry in the successor list, without checking the finger table. this is correct if the successor list actually contains the node's current r successors. However, the finger table and successor list are updated separately, and one could be more up-to-date than the other, and it is safer to omit this small optimization.

```
n.find_successor(id):
  if id in (n,successor[0]):
    return successor[0]
  else
    n' = closest_preceding_node(id)
    while True:
      try
        return n'.find_successor(id)
      catch timeout waiting for reply:
        n' = closest_preceding_node(n')

n.closest_preceding_node(id):
  result = n # in case we don't find anything better
  for i = m downto 1:
    if finger[i] in (n,id):
      result = finger[i]
  for i = r-1 downto 0:
    if successors[i] in (n,id) and successors[i] > result:
      result = successors[i]
  return result
```

Problem 5: TAO

Consider a web server W assigned a primary follower tier T1 and a secondary follower tier T2 in a slave region R.

(a) What is the latency of a write by W to an object id_1 or an association $(id_1, type, id_2)$ without an inverse, in the absence of failures?

(b) Same as (a), except while the follower F in tier T1 responsible for id_1 is down.

(c) Same as (a), except while the leader L in region R responsible for id_1 is down.

(d) Same as (a), except for a write to an association with an inverse.

In each case, express the latency in terms of intra-region message latency L_0 and inter-region message latency L_1. Justify your answer by describing the relevant message sequence. Assume TAO servers and database (MySQL) servers are on different hosts. Ignore local processing time. Consider only the latency within TAO, i.e., start the timer when a follower receives the request from a web server, and stop the timer when a follower sends the response to the web server. For the cases involving failures, assume the failure has already been detected, so you do not need to consider the timeout used in failure detection.

Answer: Notation: $\rightarrow_0$ denotes an intra-region msg. $\rightarrow_1$ denotes an inter-region msg. "master leader" is a leader in the master region. "master DB" is a database server in the master region.

(a) The message sequence is: follower $\rightarrow_0$ leader $\rightarrow_1$ master leader $\rightarrow_0$ master DB $\rightarrow_0$ master leader $\rightarrow_1$ leader $\rightarrow_0$ follower. the latency is $4L_0 + 2L_1$.

(b) Same as (a), because the request is handled by a follower in a different tier in the same region.

(c) Same as (a), because the request is handled by a different leader in the same tier (hence in the same region).

(d) The message pattern is the same as for problem (a) except as follows, where the message numbers are based on slide 28 "Writes involving inverse associations" in the lecture notes on TAO. The master leader for id1 waits for messages 3.1 and 3.2 [or 3.1 through 3.4, if we include contacting the member of the leader tier in the master region who is responsible for id2, as discussed on Blackboard] in order to contact before sending message 4; if it didn't wait, the client could be notified of completion of the operation even if the database update to the inverse association aborts due to a conflict. The slave region leader and slave region follower wait for the messages denoted 6.1, 6.2 and 7.1, 7.2, respectively; if they didn't wait, the client could be notified that the operation is complete before those updates are applied, violating the "read-after-write consistency within a single tier" requirement. These additional messages increase the latency by $6L_0$ or $8L_0$ to $10L_0 + 2L_1$ or $12L_0 + 2L_1$, respectively. Both answers are acceptable.

The message pattern for "Writes involving inverse associations" on slide 28 is based mainly on this passage in section 6.1: “If an inverse type is configured for an association, then writes to associations of that type may affect both the id1’s and the id2’s shard. In these cases, the changeset returned by the master leader contains both updates, and the slave leader (if any) and the follower that forwarded the write must each send the changeset to the id2’s shard in their respective tiers, before returning to the caller.” Another relevant passage is in section 4.2: “The tier member that receives the query from the client issues an RPC call to the member hosting id2, which will contact the database to create the inverse association. Once the inverse write is complete, the caching server issues a write to the database for id1.” This passage is somewhat vague, because the distinction between leader and follower is introduced later in the paper. Apparently this passage is talking about the leader tier in the master region.

In each case, the latency is not affected by updating the slave DB and sending invalidation/refill messages. Those tasks are done asynchronously, consistent with TAO’s requirement for “read-after-write consistency within a single tier” (section 6.1). As confirmation that the slave DB is updated asynchronously, note that the paper (section 4.5) says “If a forwarded write is successful then the local leader will update its cache with the fresh value, even though the local slave database probably has not yet been updated by the asynchronous replication stream.”