CSE535: Distributed Systems (Fall 2021)
Scott D. Stoller, Stony Brook University
Problem Set 2 (version 2021-11-26), Due 8 December 2021
ALGOMANIACS — Shubham Agrawal - [redacted], Sumeet Pannu - [redacted], Pratik Nagelia - [redacted]

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

Incase if a write operation, if we go from the start, the cleint would ask the master which chunkserver holds the current lease for the chunk and the locations of the other replicas, incase it does not have the info already cached. Now the master replies with the identity of the primary as well as the secondary servers. We know the chunck version numbers would be incremented by the master if it grants the lease, or already been incremented if the lease was already granted.

Now if a secondary fails to apply the mutation, the client code handles such errors by retrying the failed mutation. (Section 3.1 Point 7 of the GFS paper). Hence the client would retry untill all the replicas come in a consistent state. If the mutuation still fails, the client falls back and retries from the start of the write, which means , again it picks up the lock and begins the write operation.

Now as per the question, lets assume if a replica somehow misses applying the mutation and retains the chunk version number. Now since clients cache chunk locations, there could be a scenario where the client tries to read from from the stale replica before information is refreshed. Hence, client could get stale info till the window for client's cache timeout. Also the stale replicas are garbage collected by the master at the earliest opportunity and they are not given to the client when queries to master about the chunk locations. Reference : 2.7.1 Last section

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

High-level pseudo-code for the tablet location lookup algorithm in the Bigtable client library

```
def tablet_lookup (table, row_key):
    // First lookup in client cache
    if client_cache is not empty :
        row = get_usertable_location_from_metadata (cache.metadata_tablets)
        // Check if Cache is stale
        if row is null or not accessible :
            // Check if root tablet is not stale
            if root is accessible and not null :
                metadata_tablets = root_tablet.get_metadata_tablets()
                update_client_cache(metadata_tablets)

            // Root tablet is not accessible
            else :
                root_tablet = chubby.get_root_tablet()
                metadata_tablets = root_tablet.get_metadata_tablets()
                update_client_cache(metadata_tablets)
            row = get_location_from_metadata ( metadata_tablets, table, row_key )

    // Client Cache is empty, traverse hierarchy top to down
    else :
        root_tablet = chubby.get_root_tablets()
        metadata_tablets = root_tablet.get_metadata_tablets()
        update_client_cache(metadata_tablets)
        row = get_location_from_metadata ( metadata_tablets, table, row_key )
    return row.usertable_location // ( Including Primary Server and its replicas)


def get_location_from_metadata( metadata_tablets, table, row_key ) :
    //Iterate in metadata tablets and rows to identify usertable location
    for tablet in metadata_tablets :
        for row in tablet:
            if row.table == table and row_key ¡= row.endrow_key:
                return user_tablet_location
```

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 3

A) Since in paper its been already mentioned on 3.3 that Snapshot reads picks timestamp from the last known fully applied transaction, hence a replica need not ask its coordinator whether it is up-to-date for the entity group being read. It simply reads from the last known fully applied transaction, even if some committed transactions have not yet been applied.

B)
Scenario 1 : This is the scenario when there are some pending transactions to be committed or the replica selected for the read is lagging begin. Now when we do snapshot read vs current read, we get difference in the output.

1. Somehow read is requested to a replica which is lagging behind
2. We do a snapshot read, it returns the snapshot of data from the latest timestamp.
3. Now we do a current read, on the same replica.
4. It finds that the local replica is not up-to-date replica by the **Majority read** method.
5. Now the latest replica is selected, ensuring its caught up to the latest log postion and read is serviced.

Hence, the current read would differ with the snapshot read

Scenario 2 : This scenario talks about a case where a network partition cases inconsistency between replicas, as a result of which snapshot read might be ahead of current read.
1. Lets assume, leader replica is committing some writes, and network partition happends before its propagated to other replicas.
2. Now a snapshot read is requested on leader replica which is serviced by local read.
3. If a current read is requested on the other replica, since the leader replica's coordinator is not accessible, the read is services with the latest timestamp of the remaining nodes.
4. The Majority read of the remaining replica stays behind the local read of the leader read.
Hence we get a difference between snapshot read and current read.

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 4: Chord

Psuedocode for the version of the Chord that uses successor lists of length r:

```
// called periodically, verify n's immediate successor,
// updates the successor list based on the response,
// and tells the successor about n
n.stabilize():
    x = successor.predecessor;
    // if the first node in the successor list is x
    if (n.successor[0]==x)
```

```
            successor = x;
            // copying x's successor list, removes the last entry and prepending s to it
            n.successor_list = new successor_list()
            n.successor_list[0] = x;
            x.successor_list.remove(last_entry)
            n.successor_list.append(x.successor_list)
        else
        // iterating through the other nodes in the successor_list and reconciling the successor_list
        for i=1 to len(successor_list)
            if (communication_successful(successor_list[i]))
                successor = successor_list[i];
                // copying successor list, removes the last entry and prepending successor_list[i]
                n.successor_list = new successor_list()
                n.successor_list[0] = successor_list[i];
                successor_list[i].successor_list.remove(last_entry)
                n.successor_list.append(successor_list[i].successor_list)
                break;
        successor.notify(n);
```

```
// ask node n to find the successor of id
n.find_successor(id):
        // if the id to be checked is in between the current node and the first node in the successor_list
        if (id E (n, successor_list[0]) AND communication_successful(successor_list[0]))
            return successor_list[0]
        // else check the closest_preceding_node
        return n.find_successor(closest_preceding_node(id))
```

```
// search the finger_table for the highest predecessor of id
n.closest_preceding_node(id):
        for i=m downto 1
            if (finger[i] E (n,id))
                node = finger[i];
                // if found in the finger table, checking the successor_list to find the closest preceding
node with the given id
                for j=i+1 to len(successor_list)
                    if (successor_list[j] < id AND communication_successful(successor_list[j]))
                        node = successor_list[i];
                return node;
        return n;
```

There seems to be a trade off between the cost of stabilization and the speed with which the knowledge of changes propagates through the system. When we are stabilizing a particular node and encounter that some nodes are unavailable, if we don't propagate the change then, it will try to stabilize even those nodes which are not being able to communicate. Hence, when we know that successor lists are changing due to communication failures, we can remove those nodes and skip their stabilization for sometime. Paralelly, we can notify the predecessor of the current node to change the successor list as well since the current node's successor list has changed. This will enable to propagate the changes through the system faster instead of waiting for the stabilize method to be called. Therefore, if we give priority to propagate the changes through the system, the stabilization will get effected and vice-versa. This trade off completely depends on the property of the application.

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 5: TAO

(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?

1. Slave Region Follower to Slave Region Leader Cache - L_0
2. Slave Region Leader Cache to Master Region Leader Cache - L_1
3. Master Leader Cache to Master DB - L_0
4. Master DB to Master Leader Cache - L_0
5. Master Leader Cache to Slave Region Leader Cache - L_1
6. Slave Region Leader Cache to Slave Region Follower - L_0

Therefore, total: $2L_1 + 4L_0$

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

1. Slave Region Next Follower in the same region to Slave Region Leader Cache - L_0
2. Slave Region Leader Cache to Master Region Leader Cache - L_1
3. Master Leader Cache to Master DB - L_0
4. Master DB to Master Leader Cache - L_0
5. Master Leader Cache to Slave Region Leader Cache - L_1
6. Slave Region Leader Cache to Slave Region Follower which forwarded in Step 1 - L_0

Therefore, total: $2L_1 + 4L_0$

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

1. Slave Region Follower to Slave Replacement Leader Cache in the same tier - L_0
2. Slave Replacement Leader Cache to Master Region Leader Cache - L_1
3. Master Leader Cache to Master DB - L_0
4. Master DB to Master Leader Cache - L_0
5. Master Leader Cache to Slave Replacement Leader Cache - L_1

6. Slave replacement Leader Cache to Slave Region Follower - L_0

Therefore, total: $2L_1 + 4L_0$

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

1. Slave Region Follower to Slave Region Leader Cache - L_0
2. Slave Region Leader Cache to Master Region Leader Cache - L_1
3. Master Region Leader Cache to Another Master Leader Cache for inverse association - L_0 4. Master Leader Cache to Master DB (inverse association)- L_0
5. Master DB to Master Leader Cache (inverse association)- L_0
6. Master Leader Cache for inverse association sends a confirmation to Master Leader Cache which contains forward association - L_0 7. Master Leader Cache to Master DB (forward association)- L_0
8. Master DB to Master Leader Cache (forward association)- L_0
9. Master Leader Cache to Slave Region Leader Cache - L_1
10. Slave Region Leader Cache to Another Slave Leader Cache for inverse association - L_0
11. Slave Region Leader Cache (inverse association) sends an acknowledgment back to Slave Leader Cache (forward association) to maintain synchronization - L_0
12. Slave Region Leader Cache to Slave Region Follower (inverse association and forward association) - L_0
13. Slave Region Follower sends inverse association to another slave region follower - L_0
14. Slave Region Follower sends acknowledgment of inverse association to originator slave region follower to maintain synchronization - L_0

Therefore, total: $2L_1 + 12L_0$

PS: Here we are not taking into account the communication latency between the web server and the follower nodes. Also, we are not considering the asynchronous synchronization between the slave and master databases.