CS5412 CLOUD COMPUTING: PRELIM EXAM

Open book, open notes. 90 minutes plus 45 minutes "grace period", hence 2h 15m maximum working time.

SOLUTION SET

In class we often used smart highway (SH) systems as an example of edge computing. Suppose you are building a SH system that acts like an air traffic controller, giving cars permission to drive fast in designated lanes. It scales, dealing with huge numbers of such lanes, all over the country. The input includes photos, video, lidar, car tag information, etc.

1. **Think of CAP. Describe a "sub-task" (a μ-service) needed in such a system where we might relax "C" to gain higher "A and P". Be sure to tell us <u>what the task is</u>.**

As described, the system works with a lot of image data, such as images of cars, and car tags. All of this data is fixed and immutable: a specific photo won't change after it is captured. So, an image service would be an example of a plausible sub-task, and for that purpose, we could definitely cache image data safely.

2. **Tell us what "C" <u>means</u> for this task.**

For images, "C" (consistency) means "this is the correct image" corresponding to that URL, or whatever form of image identifiers the system is using.

3. **Tell us <u>what it means to relax "C" and why doing so is safe.</u>**

In this setting, the sense in which we relax "C" is that we don't need any check against the storage layer to make sure the image is correct. Normally, a cached system is only consistent if it has a way to invalidate entries that change, or if it has a way to check that the data has not changed since it was loaded into the cache. But images do not change, once captured. So, for this task, we don't need to check and yet we can be sure that if the image is cached, it is the right copy. If the image is not in cache, it could be loaded from the storage system, or we could let the application run without a copy of the image. This will be safe in the sense that the application never runs using the <u>wrong</u> image.

4. **For data used by the SH task described in your answers to Q1-Q3, would you employ a DHT (a scalable key-value store) to hold that data? If so, explain exactly what the DHT key would be, and what the values would be. If not, explain.**

For image data, a Distributed Hash Table (DHT) is ideal. The key would be a unique identifier naming the image, and the hashing function would simply reduce the key to a unique number, always the same for that particular image. The value would be the image data itself, or (if the DHT prefers smaller values), could be a URL tell us where to find the image in the file system or database.

5. **With respect to your answer to Q4, why would this technology avoid a scalability limit that would be a show-stopper for the system?**

By using a DHT to store images, we spread the workload very broadly over the set of machines that implement the key-value architecture. Every machine has a roughly equal load, and if they load is too high, we can just add more machines and make the DHT larger, redistributing key-value pairs accordingly (the method used in Chord and Dynamo can be used, for example). Thus if we test the system in the lab with, say, 50 machines holding the DHT, and later want to run it in California with 1,000x more load, we could expand the DHT to 50,000 machines.

With a DHT approach, if an edge computer needs to compare an incoming image with a saved one, it could look up that saved image and do the comparison locally. The only load is on the edge node itself, its local GPU if it uses hardware for the comparison, and on the specific DHT node it talks to when fetching the image. Of course it would also need a way to know which cars are in the segment of highway, but this list could also be saved into the DHT.

So, our solution should scale very cleanly in the number of machines we allocate to it.

6. Now assume we decided to use a DHT (no matter what you answered in Q4). Define the concept of a DHT "hot spot".

We say that a specific DHT node has become a hot spot if it has an unfair proportion of the load. For example, with N nodes in the DHT, suppose that the load is R requests per second, but instead of every node having more or less R/N of the requests, some machine has 10x more load, and some other machine has 10x lower (a "cold" spot).

There is no specific threshold for declaring something to be a hot spot, although many researchers focus on the sigma value of the load distribution and adopt the view that a node is a hot spot if it is more than 2 sigmas out from the norm. Another model some systems use is to talk about the 99% load region, or the 95% load region.

My answer focused on query loads, but the same wording could have be used for how many (k,v) pairs it holds.

7. Would the Q6 DHT be at risk of hot spots? Either explain why hot spots shouldn't arise, or give two concrete examples in which hot spots could arise.

Yes, hot spots could be a risk. We are depending on the hash function to (1) hash the DHT server nodes fairly evenly into the key-value space, and (2) also assuming that requests will be pretty evenly spread.

The question didn't ask us to solve this problem, but many DHT solutions hash each server into the DHT at multiple locations as a way to even load out. One can also hash each key in K times by appending a suffix, like "photo17-1", "photo17-2", etc, and then having the lookup pick one of these versions at random and look that version up.

Now, continue to think about a service to support a Smart Highway (SH) system, but consider *other* aspects (not the sub-task from Q1-Q7). Identify some other task that requires a Paxos-based solution.

8. **Define this task, and explain why a CAP-style solution with weak consistency would not be adequate, and why Paxos does solve the problem.**

Consider the task of using vehicle tracks to decide whether two cars are at risk of getting closer to one-another than is allowable (I'm assuming that a vehicle track basically is a bounding box for a vehicle, and then a time-based rule for knowing the location of the boundary as a function of time, and that the closeness can be computed once you have two such boxes as a minimum straight-line distance between points on one, and points on the other). If the track data is incorrect, or any other information used for the computation is incorrect (so this is intended to include all sorts of parameters such as speed, direction, etc), then we might incorrectly decide that two vehicles are closer than permitted, or that they are safely spaced when in fact they are not safely spaced.

A CAP style solution might involve using stale vehicle track data for this task, and because cars can change their direction and speed, if we use old data in the current computation, we might reach an incorrect conclusion.

A solution based on Paxos would use a strongly ordered and consistent update (an atomic multicast) each time a track is revised with new data. Every process with a replica of that data would then have the new value.

9. Will your Paxos-based μ-service for Q8 come under heavy load? Justify your answer, thinking about how your system behaves as it scales to larger and larger uses. For example, you could think about whether the average interaction with the average car needs to trigger a Paxos update (if so, the answer is that yes, it would be very loaded).

Yes, a service for managing vehicle tracks could be very heavily loaded if we are doing a smart highway. Presumably, these tracks update frequently, and with lots of cars, we would be doing a huge rate of updates and safety checks.

But in fact even though the solution uses Paxos, it can still be sharded into small subsystems with just 2 or 3 nodes, each one of which is tracking a small set of vehicles – the basic approach is the same as with a DHT. So using Paxos doesn't mean that load all goes to a single small set of servers. We could have as many servers running our sharded Paxos solution as we would have used for a DHT with a CAP-based form of weak consistency.

Thus while the overall service might be heavily loaded, we should also realize that the load can potentially be spread just as widely for a Paxos-based sharded service as for a weakly consistent DHT. Individual servers shouldn't be at higher risk of becoming hot spots.

For the remaining questions, we are no longer thinking about a SH system. Suppose that we wish to keep an in-memory copy of a very large balanced tree (for example, an AVL tree structure). This tree has an official version in some form of back-end service, but the goal is to have the whole thing cached, to soak up a massive query (read-only) workload.

The AVL tree will cache hundreds of billions of nodes, and requires enough memory to fill the DRAM memory of 1000 computers. To support this model, your group at SkyCloud.com has decided that each node will have a unique node-id used as key, a string representing the node "name", and an associated list of edges pointing to other AVL nodes. In addition, each node would have an associated "value".

The main operations would be the standard AVL operations: look up a node (by its name, to learn the value), insert a node (with a new name and value pair), delete a node (again, by name), etc. Notice that none of these make direct access to the node-id.

10. **Would you recommend using a completely random hashing policy for this kind of data, one in which each key is mapped to some arbitrary value in a hashed space? Explain why this is a good idea, or why it is a bad idea, giving a clear example.**

[Remark, not part of the answer]. A few people wondered why a DHT might want to use numbers as its keys if the application is storing tuples that have a string field called "name" that the application wishes to think of as a key. The answer is that DHTs are a service in the existing cloud. You can use Amazon's Dynamo, for example, and you don't get to modify their source code. Dynamo is already there, and DHTs like it use numbers as keys.

So would Dynamo automatically hash your string names to keys? No, because (1) how would it know that your "name" field is a natural key? (2) We would still need a way to represent the pointers in the AVL tree. AVL trees permit operations other than just looking things up. You can search for ranges of names, you can do a pattern search, you can count nodes… there are many things an AVL tree could support. So Amazon Dynamo has no reason to assume that you even want the name to be the key. (3) You might want to use some sort of hashing policy of your own to map names to key values, so they don't try to force you to use some pre-specified one. [end of remark]

The issue that would worry me is that if we use totally random hashing, every node in the AVL tree will probably be on a completely different server than its parents or children. As a result, any operation that searches the tree, or does an insert or delete, would visit a distinct server, and a search path of length log(N) would probably touch log(N) distinct servers. This could be a slow process because each of those network requests is probably going to be a fairly slow operation.

This leaves me feeling that totally random hashing is a poor choice for this sort of AVL tree structure.

11. Suppose that SkyCloud.com has a huge amount of data about actual patterns of access to the tree, and from this learns that the read accesses are a lot like the ones for Facebook photos: heavily skewed, so that the great majority of the read operations access a small subset of the nodes. But assume that these accesses *search* for the node. The query doesn't know the key of the node, and has to start from the root and examine (key-value pairs) until it reaches the node it is seeking.

An AVL node lookup of this kind normally takes O(log N) time. For us this means O(log N) nodes will be visited. Invent a way to reduce the cost to O(1) for the most popular AVL nodes. Explain how your solution would work. Hint: you can modify the DHT sharding policy or hashing policy, or both.

Our popular AVL nodes could be anywhere in the AVL tree, so the challenge here is to ensure that the very first access tends to find them. As stated, the AVL node id's have nothing to do with the AVL keys, which are strings.

Because the data is very heavily skewed, we know that there are only a small number of very popular AVL nodes. For a purpose such as this, we could simply cache pointers to the popular nodes (cache the keys for them, and also the pointers to the actual shard containing the corresponding node).

For example, suppose that every node in our system has a shortcuts table of size cK, where c is a constant such as 2 or 3, and where K is large enough to include all the popular nodes. Then each time we access a node, we could update the shortcuts table to remember the key and the server at which we actually found it, using a policy such as LRU. Since the popular nodes are accessed very frequently, they will tend to stick in the cache. Later, when we see the same key, we can quickly find the cache entry, then jump directly to the proper server where the node actually lives.

Other possible solutions come to mind, such as heavy replication of the popular items (a technique used in Gun Sirer's Beehive DHT). Those would be fine too. But I think that a simple cache would be easiest to implement.

12. For your solution in Q11, suppose that nodes have an additional operation, update-value, which doesn't insert or delete the node, but does change the associated value. Does your solution to Q11 support updates? If so, explain why. If not, explain how you would add this extra capability. Would updates also cost O(1), or would they have a higher cost, like O(log(N))? Is locking required? Be detailed!

For my cache-based lookup operation, the AVL tree itself is still held in the same sharded DHT structure as we would have used without the shortcuts. As a result, since the AVL structure was already supporting update, it still does so. We can update tree nodes by simply having the shard where the key lives replace the old AVL node with a new copy that has a new version of the value field.

These updates would not invalidate the short-cut cache, so we still can do lookups of popular values in time O(1): those nodes quickly stick in the cache, after which a lookup can jump directly to the target shard.

One impact of updates is that perhaps the popularity of AVL nodes would change over time. But my cache has size c*K so it has ample extra space for changing popularity. If some AVL node X used to be ultra-popular, it will be in cache, but now when Y becomes popular and X cools off, X will drift towards the bottom in the LRU ranking and eventually will be evicted.

13. [Skipped, in case anyone is superstitious]

14. A different DHT question. We will use a DHT to store (key,value) pairs where the key is a string and the value is an integer. A task must scan from key A to key B (in alphabetic order), and sum up the corresponding values. Would you prefer for the DHT key hashing function to be highly random, or would you wish for it to preserve the ordering on keys, so the node "owning" A is the first in a sequential list of DHT nodes, with the last one in the list owning B? Explain briefly.

To solve this problem, I would periodically run a job to take the AVL nodes, left to right, and "bin" them into my shards, also viewed as a left to right list. If there are N AVL nodes, very large, and S shards, much smaller, we would simply put N/S nodes into each shard. Each shard ends up containing all the AVL nodes with keys in some range, such as X to Y. This does force us to use an index of size S to find the proper shard for a particular AVL node, but the lookup is easy and can be done quickly. Thus my answer is "yes, an order-preserving policy is best."

Now, all we have to do is to look up the shard where A lives, and the shard where B lives, and all the AVL nodes with keys from A to B are on shards in that linear sequence: **Shard(A)**… **Shard(B).** We can just scan those shards.

15. With respect to your answer to Q14, discuss the expected performance in terms of delay seen by the query caller.

If there are K shards in the range from **Shard(A)** to **Shard(B),** we need to visit each of them, and the problem described this as a sequential "scan", so the delay would probably be something like K times the delay for visiting one shard.

16. With respect to your answer to Q14, discuss efficiency for the overall DHT μ-service.

There are often tradeoffs between the most efficient policy for the client of the DHT (the query caller) and the DHT itself. Our solution to Q14 allows the individual client to access DHT shards directly, which minimizes costs for the client, but each client request will involve a message to send the request, and then a message in which the server sends back its reply. So with Q queries we would have 2Q messages. In the case of Q15 where we scan K shards, if K is an average, we end up with 2*Q*K messages.

It is possible to imagine ways of batching these requests, so that the DHT itself would have B queries per message that shows up. Batching would be more efficient: messages get larger (B times larger), but the number of messages sent and received drops (by a factor of B). The queries had to be sent in any case, so there is no loss of efficiency in terms of bytes sent per query, or per reply. However, because sending or receiving a message is costly (the service probably receives an interrupt per message, and may have other overheads too), we gain efficiency (one interrupt now covers B requests, or B replies). But batching would slow down the response in terms of client delay: the larger we make B, the longer it might take to fill a message with B queries, and hence the longer the delay before we execute the query.