Name: _____ Netid: _____

Quiz 1 – Solution Set

Q1 problem setting: You are in charge of content caching for an internet news feed. Your system runs on the Azure cloud and implements a DHT cache, with one node per shard (no replication). If a failure causes some node in the DHT to crash, a replacement is automatically launched and will jump into the empty slot. It will start up with an empty ("cold") cache. *If we scale this form of cache up by adding more nodes, we would have more shards but they would still have one node per shard.*

The way your service works is very simple: any time the newspaper has a new article, web page servers check the DHT first for a copy of any photos or video in the article. If they get a hit in DHT cache, they don't need to fetch it from the newspaper database. On the other hand, if the DHT cache lacks the photo, the server will fetch the photo from the database, but then stores a copy into the DHT. This way other servers won't need to pester the database.

[1] As the newspaper becomes more and more popular, you increase the number of DHT server nodes in proportion to the peak number of concurrent customers. Would you expect this to provide linear scalability? Explain, and be sure to state any assumptions that your answer depends on.

On average, yes, this would give better scalability – but the performance will depend on how evenly load is spread. If one specific article is very popular, this scheme will map it to one particular shard, hosted on one machine. That server will get hit by all the requests for this one article.

[2] Prince Harry and Duchess Meghan have a new baby, and they release an official photo. Your DHT service malfunctions: instead of showing the official photo, the news article shows a blank box with an X in it and a message "DHT service timeout error." You notice that one of the DHT server nodes has crashed. Worse, each time it self-repairs, the new server crashes too. Explain what is causing this problem, and why your answer to part [1] makes sense given this obvious scaling issue!

This is just what I would expect with a hot-spot scenario. If that one server can handle, say, 10,000 requests per second, but the customers are trying to fetch the photo at a rate of 100,000 per second, then (at least) 9 out of 10 requests will timeout.

[3] Suppose that you are permitted to modify the implementation of the DHT cache server, but cannot modify the application (the newspaper web implementation, or the photo collection service). How could you change or reconfigure it to fix the caching issue identified in [2]?

The best option is probably to add more nodes per shard. Some DHTs allow one server to join multiple shards, and that strategy could work really well here. If each DHT server is "primary" for one shard but has a "secondary" membership in some other shard, our clients can be load-balanced over the two. Or we could generalize this for k-level redundancy. The number of servers won't change but the load will be way more evenly spread.

[4] Now, unlike for part [3], suppose that you *cannot modify the DHT cache in any way*. How could you modify the newspaper photo management service (and if needed, the first-tier web servers that generate web pages) to avoid the issue identified in [2]?

An easy fix is to just upload a bunch of versions of the article and photo, like "xxxx.v1", etc. Then randomly hand out URLs with different version numbers. This can spread the users out within the DHT because with different file names, the odds are that the versions will be placed on different servers. So this spreads the load too, even with just one server per shard.

[5] Exactly at the same time that the DHT cache was having problems due to the Harry and Meagan photo, lots of other photos were displaying without problems. On the other hand, there are some photos that had problems too. For example, a photo of Professor Birman teaching cloud computing that happened to be in the Cornell alumni news ("Professor still teaching after nearly 40 years") also times out, just like the Harry and Meagan one. Explain both observations.

With a DHT, many objects might hash to the same server. So if that server becomes overwhelmed, all the requests start to timeout in a correlated way. Other servers (hosting different shards, hence different data) might not be overloaded at all. This accounts for all the behavior we see.

Q2: DHT hashing. In class we looked at key-value stores (DHTs) that map from key to shard this way: **shard-number = Hash(key) % number-of-shards**. Here, **Hash** is a hashing function supplied by the programming language, implemented by a method such as SHA-64 or SHA-256.

[1] Define the term "shard" that was just used above.

If a database has d items in it, and we create s subsets of size roughly d/s and then store the subsets on different machines, we say that the servers hosting a given subset are a "shard" of the overall database. The same idea can work for a computation split over lots of machines, files in a scaled out file server, etc. The total data or computation is hosted in the whole service, but any given server has just some portion of it.

[2] A shard often uses state-machine replication for fault-tolerance. What guarantees do state machine replication methods provide?

State machine replication says that if we have a deterministic piece of code and start a set of replicas of it in the same initial state, then present them with the same updates in the same order, they will evolve through a sequence of identical states.

[3] Suppose that an application needs to store a photo into the DHT. The developer proposes to hash the photo itself (the jpg image object) and use the hash as a key. Would this be a good idea? Why?

No, not a very good idea. If we need to fetch the photo by some sort of name (like in a database query), we wouldn't know its key and will have to maintain a secondary index that can be searched to learn the key. It usually makes more sense to just agree that the k'th photo Ken uploaded would have an easily-known name like photos/Ken/photo-k. This way if an application wants to show all of Ken's photos, it knows their names just by design. Otherwise, it would have to know what is in the photo in order to generate the name of the photo, which is a kind of chicken-and-egg problem.

[4] Suppose that we insert 75,000 objects into the DHT, each with its own unique key. If there are S servers (assume S < 20) and each shard has just one server (so there are also S shards), would we

expect each shard to have exactly 75000/S objects on it? Would it point to a bug if some shard had twice as many objects on it as the average shard, and some other shard had half as many? Explain.

Pseudo-random hash functions aren't perfect, so this might not surprise me and wouldn't be a sign of a bug. In fact it would surprise me to see an absolutely perfect split, since that doesn't sound perfectly random. In addition, there could be application-specific reasons that we might deliberately collocate objects in the same shard for better performance, for example by carefully selecting a sequence of object keys designed to hash to that shard for each of the distinct objects.

[5] Suppose we have sub-objects, like "Ken Birman/photo", "Ken Birman/age", "Ken Birman/job". These names are used as keys. Would you expect to find all of the sub-objects on the same shard, or spread over multiple shards? Explain briefly.

Multiple shards: DHTs hash the entire pathname. These objects have different names so they will be hashed to different pseudo-random numbers, and then the modulus function will probably map them to distinct shards.

Q3: Jim Gray concern. In a paper we discussed, Jim Gray and some colleagues at Microsoft studied a model of a database with T concurrent transactions and S servers, and discovered that in their model, overheads were growing as a polynomial (specifically, S^3T^5). You just took a new job at "Skunkworks.com", and your team is responsible for the main database system. Nobody at your new job is aware of the Jim Gray paper or analysis, so the team has never seen this formula.

[1] Your boss says that the system currently has a capacity of T = 10,000 transactions per second and runs on a server with S=2 nodes. Load is growing and by the end of the year is predicted to reach 25,000 transactions per second. The team wants to upgrade to a 5-node system (S=5), and your boss asks you to double check that the new configuration will be more than fast enough for T = 25,000 transactions. Is Jim Gray's analysis useful here? Why or why not? Will the upgrade help? *Keep in mind that your boss and the team had never heard of Jim's result. They could be doing a smart thing, or they could be doing a dumb thing… they have never seen Jim's equation.*

*Jim teaches us that what the boss is trying to do will probably cause a slowdown. Once the server is under enough load so that overheads dominate, they tend to rise as N^3T^5. This says that doubling the number of servers and also doubling the number of transactions the service is asked to handle will most likely cause a slowdown of about 8*32 which is 256. For example, if the service was completing 10,000 transactions per second before scaling up, now it will only be doing something like 40 per second. The boss will freak out!*

[2] Your boss believes that the current load limit of T = 10,000 is actually due to "hot spots" in which many transactions happen to access the identical item at the identical time. The team has come up with an idea for eliminating half the hot spots (they can't eliminate all of them, so transactions will still be accessing some shared data). Will this hot-spot-reducing idea increase the capacity of the server? Explain your reason for saying yes or no.

Yes, this could help a little: Jim's analysis focused on transactions that conflict by accessing the same

data and needing the same read and write locks. Spread the transactions out to have less contention and the T^5 part of the formula will definitely have a smaller effective value of T. But the phenomenon itself would still be present. So this might help, and could perhaps even get us past the original limit of 10,000 per second. But because half the hot spots remain, we ultimately will see a Jim Gray slowdown even so.

[3] Your team has decided to take Jim Gray's advice and shard the database. Explain what this actually means. For example, suppose that the original single database had 10M database items in it, and suppose that the sharded system will have 5 shards. Where do the 10M items end up stored?

They get spread over the 5 shards, with 200,000 items per shard. Now we need to modify the logic of the application because Jim's approach won't allow one transaction to touch more than one shard – but this seems to be the topic of part [4], so see below.

[4] Your boss asks you to port some of the existing transactions to the new sharded system. You notice that several transactions to atomic transactional (SQL) reads or updates to data that would now live on multiple shards. What issues will arise when you port this logic to the sharded solution? *Hint: your answer should talk about performance issues but also correctness, for example if some of those transactions do reads and updates to multiple data items.*

The basic idea is that you take the transaction and just break it into multiple transactions that run in parallel, or if there is a read-then-write, maybe run in some sequence. But that can be difficult and also a failure or any kind of contention with other concurrent transactions would cause the split transaction to run non-atomically. So there is a good chance the users could notice that behavior changed at this point. The so-called BASE methodology adopts Jim's approach and recommends just modifying the API the users see so that the new behavior seems like a feature, not a bug. Then still with BASE, you add a cleanup that periodically looks for such issues and somehow reconciles them.

Q4: Paxos Protocol. Leslie Lamport's original Paxos protocol defines two constants, QW and QR.

[1] Define the constants QW and QR, including any restrictions they must satisfy.

QW is the number of acceptors (log servers) that a write needs to update. QR is the number of logs that need to be merged to fill in any gaps. The rules are that QW+QW > N, and QW+QR > N, for N the total number of servers.

[2] In very short English sentences, one sentence each, define the Paxos "client", "leader", "acceptor" and "learner" roles.

A client submits requests which can be updates (writes) or queries (reads). A leader is in charge of doing a write and runs the Paxos protocol. Acceptors log requests and respond ("ack" or "reject") to the leaders, following rules spelled out by the protocol. A learner is a kind of leader but used for read queries – it merges logs on behalf of a client requesting to read the current Paxos logged state.

[3] What possible values for QW and QR are permitted in a Paxos system with 5 acceptors that *needs to remain available even with* 1 failed node (1 acceptor might not be accessible)? N*ote: reads and writes must remain active and complete successfully even if only 4 of the 5 acceptors can respond.*

With N=5, we could allow (QW=4,QR=2), (QW=3,QR=3). We can't support QW=5 or QR=5, and QW=2 violates QW+QW>N, so these two are the only options.

[4] Now consider a sharded service like a DHT and assume that it uses Paxos inside the shards, to implement state machine replication. Suppose that the shard size is 3, but that the service as a whole has 300 members (so, it has 100 shards). In your answer to [1] you talked about the number of members. Would QW and QR need to be defined relative to 300, or relative to 3? Explain.

If we use Paxos on a per-shard basis, N would be the shard size (3). This is because each shard has its own private set of logs, and its own private Paxos state. If we used N=300 it would be as if our service wasn't sharded at all.

[5] Suppose that Paxos was deployed once per shard in our system for [4], but also deployed one extra time for the whole service (spanning the full set of shards). The idea is to use state machine replication for the shards, but also to have configuration data that the whole application uses, and the "whole service" Paxos would be used to track changes to the configuration data.

a. Would the whole-service Paxos use the same log as the per-shard Paxos, so that a given slot in some shard's log could be either a shard-data update or a configuration update? Explain.

The whole-service Paxos would run the identical Paxos protocol, but with its own separate logs, and its own values of QW and QR. So in effect we would have two side-by-side modules using the Paxos protocol, but configured differently and using distinct data (logs), so they would never interfere in any sense.

b. When using Paxos this way, would the total-order guarantee of Paxos span all updates (shards and also the configuration updates), or would configuration updates not be ordered relative to shard data updates?

The guarantee would only apply within the Paxos group, so we would have total ordering for configuration updates, and total ordering for shard writes, but not "between" shard writes and configuration updates.

Q5: IoT Data and the Meta System. Suppose we have IoT sensors that have some known error bound for their value, δ, and also some known error bound on time, ε. Thus if a sensor reports value x at time t, this really needs to be interpreted as $x \pm \delta$ at time $t \pm \varepsilon$.

[1] Suppose that we have redundancy: some value is tracked by 3 sensors. 1 might be faulty. Meta teaches us that the correct sensor values will overlap. Thus if 1 sensor gives a very divergent reading, we can conclude that the other 2 must be correct. Explain why this might let us get a more accurate estimate of x and t than the official accuracy of $x \pm \delta$ at time $t \pm \varepsilon$.

If the very divergent sensor can be ignored, we know our true sensor reading must be in the overlap region for the two remaining sensors. This will still be a bounding box, but in general would be a smaller area than the area for a single sensor (a picture makes this obvious, and we had one on the slide set for that lecture). Since accuracy is basically "area within which the value resides" this shows that accuracy would often be better – not always, but often.

[2] Still in a case where we might have 1 faulty sensor out of 3, suppose that all 3 sensor readings overlap (meaning: any pair of 2 has at least some shared value and time range). What can we conclude about the actual ("true") value of the underlying data?

Now we can't figure out a-priori which sensor is faulty. Any pair could be correct and the third could be wrong, and in fact all three could be correct as well. So the best we can do is to take the union of the three overlap regions: we know the correct value is somewhere in this union.

[3] Suppose that we need to cool a chemical reaction if the temperature *might* have reached some threshold value *max*. In case [2] where we have 3 sensors *and they all overlap*, what rule should we use to be sure that we never make a mistake and let the chemical reaction overheat? *Hint: before answering, make a picture of 3 sensor readings that all overlap, and think about what it means if you suspect that perhaps 1 of the 3 is in fact a faulty sensor. Keep in mind that the batch of vaccine will be spoiled if the reactor temperature actually does exceed <u>max</u>. We don't want the faulty sensor to be able to trick us into allowing that to happen!*

In this case, we apply the logic from (1) and (2) and arrive at some region within which the value resides. If any point within this region exceeds the threshold, cool the chemical reactor. This is because any point in this region, by definition, could actually be the correct current reading.

[4] In a dairy setting, suppose that we measure the body temperature of a cow using a sensor inside the cow's stomach (this doesn't bother the cow). Now assume that the cow drinks a gallon of cold water and the sensor temporarily shows the water temperature, until it warms up. Thus in this case the sensor is not reporting what we would like it to report – the value is a "faulty" value for the cow's body temperature. Would the 2 out of 3 redundancy feature allow us to figure out the actual cow body temperature (e.g. if we put 3 sensors in the cow's stomach instead of just 1)? Explain.

Here, there is no fault – the thermometer sees the actual temperature. But the temperature briefly isn't reflective of the cow's body temperature. So we need to monitor when she drinks water and inhibit our body-temperature tracking for a suitable amount of time, like 20 minutes, to give the water time to be absorbed and for her stomach to return to the true temperature of her body as a whole. If we can't know when she drinks water, we could probably train an ML model to recognize these dips and automatically apply this rule based on learned patterns.