Distributed Systems

Exam 3 Review – Spring 2020

Paul Krzyzanowski

Rutgers University

Spring 2020

Note

Your questions and your answers are most likely in a different order than listed here.

The Google search architecture favored the use of:

a. **Low-cost computers.**
b. The fastest available processors.
c. Highly-reliable computers.
d. Fault-tolerant disks.

The Google cluster architecture that was developed for Google search (among other services) focused on the price-performance ratio, factoring in the cost the computer, power efficiency, and cost of operation.

They favored placing reliability into the software rather than the hardware.

Sharding an index for a Google search:

a. **Converts a big search into many small searches.**
b. Makes search fault tolerant by creating replicas.
c. Makes a single search span across multiple data centers.
d. Separates index servers from document servers.

Sharding an index breaks a huge (10s or 100s of TB) index into lots of smaller indices. Searching for words can be done quickly and in parallel among these small indices. Results can then be merged.

- Index & document files are replicated but that's not the point of sharding.
- Files are replicated across data centers but an individual search stays local.
- Index & document servers are separate files but that's not the point of sharding.

If a *reduce worker* fails in a MapReduce job:

a. **Only that failed reduce worker must restart.**
b. All reduce workers must restart.
c. All map workers and the failed reduce worker must restart.
d. The entire job must restart from its last checkpoint.

A reduce worker contacts all of the master workers for the (key,value) data it needs for its partition. There is no need to recreate that data.

- Since reduce workers don't need to communicate with each other, there is no need to restart any other reduce worker.
- The map workers are done. No need to restart them.
- There is no checkpointing in MapReduce.

A *reduce* function on a reduce worker in MapReduce can start running:

a. **When every map worker has finished.**
b. When at least one map worker has generated a key for that reduce function.
c. At the same time as all the map workers.
d. When at least one map worker finishes.

Map workers produce intermediate files that are partitioned for all the reduce workers. Those files must be complete before the master starts any reduce tasks. Each reduce worker contacts all the master workers for the (key,value) data it needs for its partition.

- A reduce worker needs ALL of the (key, value) data for its partition and we don't know whether *any* map worker still has another (key, value) pair to generate for that reduce worker until it is finished.
- A reduce function cannot be invoked until we have all the values for a key.
- Other map workers may generate (key, value) data for the reduce worker.

What is the importance of choosing a good row key in Bigtable?

a. **Take advantage of the locality of adjacent rows.**
b. Avoid use of duplicate keys.
c. Enable rapid search for an arbitrary row based on its key.
d. Make queries user friendly for easier debugging.

Bigtable keeps its data sorted by the row key. Rows are stored next to each other in a tablet – except occasionally when we have to jump to the next tablet.

- Bigtable shouldn't support duplicate keys, just like a database won't support duplicate primary keys for a table.
- Bigtable uses a balanced tree to find a range of rows. Key choice will not change how rapid that search is.
- Making queries user friendly doesn't really make sense.

Spanner enables lock-free reads:

a. **By having a transaction read data that is no newer than some chosen time.**
b. Only in the case that there are no currently running transactions that have write locks.
c. Under the condition that there are no other read locks or write locks for the needed data.
d. In cases where a parent transaction already grabbed all the necessary locks.

Spanner supports ACID transactions but also *snapshot reads.* These do not require locks because all the data read is ≤ some specified point in time. Modifications to that data will result in newer versions but spanner stores multiple versions.

.

If the network connection between two data centers that run spanner is broken:

a. **Spanner transactions may have to wait until the connection is restored.**
b. Transactions in one of the data centers may access stale data.
c. Any outstanding transactions will be aborted.
d. Transactions will still be permitted to commit even if they cannot access some data.

Spanner behaves as though it violates the CAP Theorem. Instead of an eventual consistency model, which became popular to offer high availability, it appears to offer high availability AND consistency.

However, you cannot violate the CAP Theorem, and Spanner provides consistency *as long as there are no partitions.* Should a network partition occur, Spanner chooses consistency over availability, so transactions may have to wait for the partition to be repaired. Google addresses this largely by having multiple redundant links between servers and between datacenters.

A Pregel job is complete when:

a. **Every vertex votes to halt and no vertex sends a message to another vertex.**
b. Every vertex votes to halt.
c. No vertex receives any more incoming messages to process.
d. A majority (over 50%) of vertices agree to halt.

In Pregel, a vertex may vote to halt when it is done. However, another vertex sent it a message, the framework will keep running and the vertex will process its input. If it chooses to do so, it can vote to halt again. We can have a situation where every vertex voted to halt but some vertex also sent a message to another. That will cancel the halt.

- Every vertex voting to halt is not sufficient.
- Not receiving incoming messages doesn't guarantee the vertex will vote to halt.
- Halting isn't done via a majority vote.

For Hive to create the proper code for a MapReduce job, it gets information about the structure of the data from the:

a. **Metastore.**
b. Driver.
c. Hadoop Distributed File System (HDFS).
d. Compiler.

The *metastore* is a small database that describes the structure of data so Hive can generate code to extract it and parse it properly.

- The driver receives HiveQL queries and manages the session that coordinates the progress of the execution of the query.
- HDFS is typically the file system that stores the data HiveQL will work on.
- The compiler converts the HiveQL query into a series of steps that will be performed by MapReduce.

A resilient distributed dataset (RDD) in Spark:

a. **Cannot be modified.**
b. Can have transformations but not actions applied to it.
c. Can have actions but not transformations applied to it.
d. Is persistent and must be written to a file rather than cached in memory.

RDDs are immutable. This means they cannot be modified. If you want to make changes, you do so by applying a transformation, which creates a new RDD.

- RDDs can have both transformations and actions applied to them. An action is a finalizing operation prior to returning data to the user. Transformations create other RDDs.
- RDDs are not necessarily persistent. They are generally stored in memory and, if missing, can be regenerated from the transformation that was used to create them.

Spark's lazy evaluation means that the Spark framework will:

a. **Start from the final action and figure out what transformation is needed to generate data for that action.**
b. Shard the input data so multiple workers can process it in parallel.
c. The results of one transformation will be completed before the next one starts.
d. Break a complex problem into a series of transformations and actions.

Spark's lazy evaluation means it works backwards, starting from the action. Each RDD that is missing triggers execution of the transformation that creates it.

- RDDs are sharded but that's not what lazy evaluation does.
- A transformation may be able to start before another is complete, but this has nothing to do with lazy evaluation.
- Breaking a problem into a series of transformations and actions is a task for the programmer (or a query compiler) and not a facet of lazy evaluation.

A technique shared in both Adaptive Bitrate Coding (ABR) and Spark Streaming is:

a. **Break a continuous stream of data into chunks.**
b. Use a variable-size buffer to handle the mismatch between input and output data rates.
c. Use feedback from the client consuming the data to change how the data is processed.
d. Partition the data so that it can be processed by multiple systems in parallel.

Both ABR and Spark Streaming break streaming data into chunks that represent a window of time. In the case of Spark Streaming, the data in that chunk can be processed as an RDD. In the case of ABR, we have a chunk of video data that can be encoded in a variety of bitrates to accommodate different connections to the end user.

If a caching server gets heavily loaded, a Content Delivery Network (CDN) is likely to:

a. **Return alternate addresses to new DNS (Domain Name System) server queries.**
b. Forward the request to a different server.
c. Pass the request directly through to the origin.
d. Drop the request and hope the client retries when the server is less busy.

CNDs (such as Akamai) use a load balancing DNS. In addition to geographic distribution to find the closed data center, the DNS server can perform *load shedding*, where it will not return IP addresses of heavily-loaded servers.

- Requests are not routed from server-to-server.
- User requests are only passed to the origin if they require dynamic content.
- Requests are not dropped.

Under what condition can you download useful content from a leecher in BitTorrent?

a. **If it has any block of content that you don't have.**
b. If a leecher started prior to your download, even if it is still downloading content.
c. Never; content should be downloaded from seeders.
d. f you need to re-download an earlier block that was discovered to be corrupt.

There are three answers that work. You needed to figure out the *best* one. The whole point of bittorrent is that each leecher downloads a random collection of blocks. Hence, even if a leecher started to run *after* your leecher, it may very well have downloaded blocks you can use.

- A leecher starting before yours is not a necessary condition and there is always a chance it will not have content you can use (its downloads may be slower than yours and it has no blocks you need).
- You may need to re-download a block if it is corrupt, but you will realize that as soon as a block download has completed rather than go check earlier blocks.

An advantage of using query flooding in a peer-to-peer network is:

a. **It does not rely on a centralized database.**
b. It scales well, providing increased performance as more peers are added.
c. Messages contain a hop count to avoid forwarding loops.
d. Queries have a variable latency, so you get rapid responses from neighbors.

Flooding generally isn't desirable. It bothers servers needlessly and takes time because of query forwarding. One advantage it does have is the lack of a central server. This makes it difficult to shut down as a peer-to-peer service.

- Flooding scales poorly; performance usually drops with more peers.
- You want to avoid forwarding loops but this isn't an advantage to flooding.
- Variable latency isn't an advantage either.

An advantage of clustered file systems is:

a. **File data is consistent when accessed from multiple servers.**
b. They spread data among multiple systems for high scalability.
c. Metadata is separated from file data.
d. They eliminate the need for a distributed lock manager.

A *clustered file system* is a special type of file system that is installed locally on each computer and understands that it is making block read and block write requests to a shared disk. Normal file systems can read and write blocks without any concern that some other system will modify them since the operating system owns the disk. With network attached storage, you send high-level RPCs (read bytes from a file, delete a file, etc.) that the file system on the server operates on. With cluster file systems, the file system on your local disk must use a *distributed lock manager* at certain times to grab locks for certain blocks to ensure that another operating system doesn't modify the same data at the same time.

- Spreading data among multiple computers is a key method of addressing scalability. However, the definition of a clustered file system does not have anything to do with setting up a cluster of computers.
- This may be done in some implementations of a cluster file system but isn't expected.
- Cluster file systems require a DLM to avoid conflicting modifications of the same block.

Fencing can be useful in a cluster because:

a. **We cannot distinguish a failed system from communication delays.**
b. We can improve the security of a system by isolating security-critical components.
c. It enables multi-directional failover.
d. It avoids the need for cascading failover.

A big problem with asynchronous networks (such as the Internet) is that we cannot reliably distinguish non-responding systems from delayed messages. We saw that this was addressed in virtual synchrony by taking suspect systems out of the group. Fencing is a way of isolating a system from the cluster.

- Fencing not a form of firewall to isolate security-critical parts from the rest of the cluster.
- Fencing has nothing to do with failover.

For Sophia to send a secret message to Emma, she would:

a. **Encrypt it with Emma's public key.**
b. Encrypt it with Sophia's private key.
c. Encrypt it it with Emma's private key.
d. Encrypt it with Sophia's public key.

For Sophia to send a secret message to Emma, she needs to encrypt it in a way that only Emma would be able to decrypt. The thing that Emma has that nobody else has is Emma's private key. Hence, Sophia has to encrypt the message with Emma's public key.

Which is not an encryption algorithm?

a. **DH Diffie-Hellman.**
b. AES (Advanced Encryption Standard).
c. RSA (Rivest–Shamir–Adleman).
d. ECC (Elliptic Curve Cryptography).

Diffie-Hellman is a key exchange algorithm, not an encryption algorithm.

- AES is the most popular symmetric encryption algorithm.
- RSA is the most widely-used public key encryption algorithm.
- ECC is a newer & generally faster public key encryption algorithm.

Systems often store password hashes to:

a. **Enable validating a user without storing the user's password.**
b. Enable passwords to be sent securely over an insecure network.
c. Make sure that the stored password has not been corrupted.
d. Make it difficult to perform a dictionary attack.

Hashes are one-way functions. Given a hash, you cannot find an inverse function that would give you the original message. This makes them popular for storing passwords. A user can supply a password and the system can check if

hash(password) == stored_hash

but anyone who steals a password file will not be able to find passwords except by trying all permutations.

- The hashing of a password is done at the server. They don't secure the network link. An attacker could just as easily sniff & send a password hash.
- They are not used as an integrity check for stored passwords.
- A dictionary attack is one where you try well-known passwords and dictionary words to break in. Hashing doesn't make that more difficult.

A digital signature enhances a message authentication code because it:

a. **Can only be created by a single party.**
b. Uses a hash as an integrity check.
c. Is encrypted so that only trusted parties can validate it.
d. Cannot be reversed to reveal the message.

A digital signature is encrypted with a user's private key, ensuring that anyone who does not have that key will not be able to create the message. At the same time, anyone with access to the public key can verify the message. With message authentication codes, we need a shared key. Either party can create a valid MAC.

- MACs use hash functions too.
- Anyone can generally validate a digital signature since public keys are public. MACs, on the other hand, require a shared secret key.
- Hashes cannot be reversed. Hence, neither MACs nor digital signatures can be reversed.

If an attacker steals your X.509 certificate, they can:

a. **Authenticate you.**
b. Impersonate you.
c. Extract your private key.
d. Ask the certification authority (CA) to revoke your identity.

The term “attacker steals” was used for deception. There is nothing secret about an X.509 certificate. Every time you connect to a web site with https, you download someone’s X.509 certificate. It stores a public key & information that identifies the server. The public key can be used to authenticate the owner (send a nonce & ask the owner to encrypt it with their private key).

- To impersonate you, the attacker will need to replace your public key with their own so they will have the corresponding private key. That means they will need to be able to recreate the signature on the certificate, which requires knowing the private key of the certification authority (CA).
- Your private key is not in your certificate.
- Having a copy of someone’s certificate does not give you any rights in having the CA send any certificate revocation messages … or anything else.

MapReduce

You are given a data set of students. It contains the following fields: Name, Age, School, Grade

Explain, in pseudocode, how you would use MapReduce to list the average age of students below the ninth grade in each school:

```
map:
    if (grade < 9) write(school, age)
reduce(school, ages)
    write(school, sum(ages)/count(ages))
```

The *map* function passes through only the data of interest: the school and age of any student below the 9th grade.

The school is the key so shat all we can group the ages per school. The *reduce* function is called once for each school (the key) and given a list of ages (values). It simply computes the average:

```
reduce(school, ages)
    write(school, sum(ages)/count(ages))
```

Alice (*A*) wants to create a fixed-size message integrity check for a message *M* that only Bob (*B*) can verify. What would she need to send in addition to the message *M*. The message is not secret.

Assume all public keys are shared.
Use only hashes and public key cryptography and the notation

H(X) to hash a message *X*
ER(X) to encrypt *X* with *R*'s public key
Er(X) to encrypt a message *X* with a user *R*'s private key.

Multiple messages may be separated by commas*: T, U, V*.

A fixed-length integrity check would be a hash. To ensure that *only* Bob can verify the hash, Alice will encrypt it with his public key: EB(H(M))

Bob can validate the message by decrypting it with his private key and comparing the results to a hash of the message.

If Alice is concerned about an intruder modifying the code and the message, then Alice would need to encrypt her signature of the message: EB(Ea(M))

Bob validates that by decrypting the message with his private key, decrypting the result with Alice's public key and comparing the results to the hash of the message.

The end.