Section 12: ACID, Fault Tolerance, and Distributed Data

November 13-15, 2019

Contents

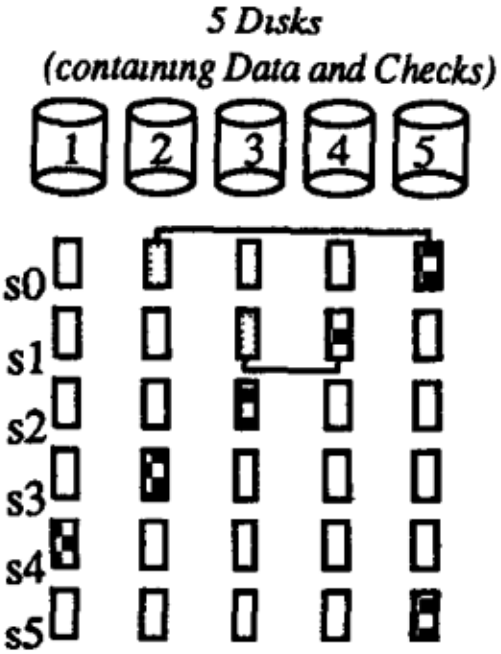
1 Vocabulary

- **Fault Tolerance** The ability to preserve certain properties of a system in the face of failure of a component, machine, or data center. Typical properties include consistencies, availability, and persistence.
- **Transaction** - A transaction is a unit of work within a database management system. Each transaction is treated as an indivisible unit which executes independently from other transactions. The ACID properties are usually used to describe reliable transactions.
- **ACID** - An acronym standing for the four key properties of a reliable transaction.
  - *Atomicity* - The transaction must either occur in its entirety, or not at all.
  - *Consistency* - Transactions must take data from one consistent state to another, and cannot compromise data integrity or leave data in an intermediate state.
  - *Isolation* - Concurent transactions should not interfere with each other; it should appear as if all transactions are serialized.
  - *Durability* - The effect of a committed transaction should persist despite crashes.
- **Idempotent** - An idempotent operation can be repeated without an effect after the first iteration.
- **Log** - An append only, sequential data structure.
- **Checkpoint** - Aka a snapshot. An operation which involves marshaling the system's state. A checkpoint should encapsulate all information about the state of the system without looking at previous updates.
- **Write Ahead Logging (WAL)** - A common design pattern for fault tolerance involves writing updates to a system's state to a log, followed by a commit message. When the system is started it loads an initial state (or snapshot), then applies the updates in the log which are followed by a commit message.
- **Serializable** - A property of transactions which requires that there exists an order in which multiple transactions can be run sequentially to produce the same result. Serializability implies isolation.
- **ARIES** - A logging/recovery algorithm which stands for: Algorithms for Recovery and Isolation Exploiting Semantics. ARIES is characterized by a 3 step algorithm: Analysis, Redo, then Undo. Upon recovery from failure, ARIES guarantees a system will remain in a consistent state.
- **Logging File System** - A logging file system (or journaling file system) is a file system in which all updates are performed via a transaction log ("journal") to ensure consistency in case the system crashes or loses power. Each file system transaction is written to an append-only redo log. Then, the transaction can be committed to disk. In the event of a crash, a file system recovery program can scan the journal and re-apply any transactions that may not have completed successfully. Each transaction must be idempotent, so the recovery program can safely re-apply them.
- **Metadata Logging** - A technique in which only metadata is written to the log rather than writing the entire update to the log. Modern file systems use this technique to avoid duplicating all file system updates.
- **EXT4** - A modern file system primarily used with Linux. It features an FFS style inode structure and metadata journaling.
- **Log Structured File System** - A file system backed entirely by a log.

- **Checksum** - A mathematical function which maps a (typically large) input to a fixed size output. Checksums are meant to detect changes to the underlying data and should change if changes occur to the underlying data. Common checksum algorithms include CRC32, MD5, SHA-1, and SHA-256.
- **Replication** - Replication or duplication is a common technique for preserving data in the face of disk failure or corruption.

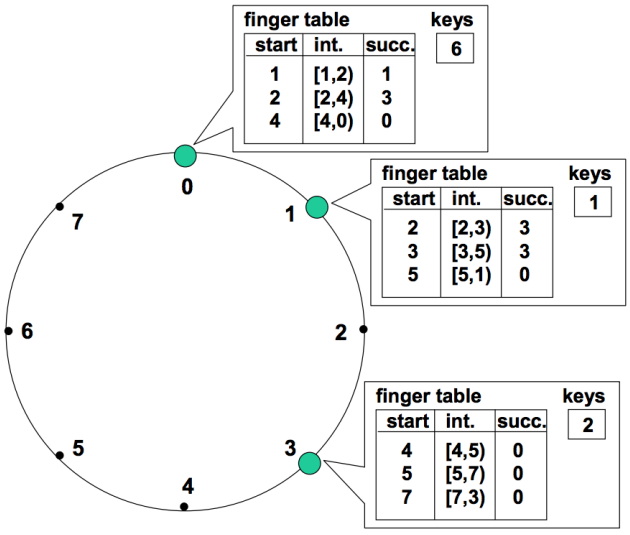
  If a disk fails, data can be read from the replica. If a sector is corrupted, it will be detected in the checksum. The data can then be read from another replica.
- **RAID** - A system consisting of a Redundant Array of Inexpensive Disks invented by Patterson, Gibson, and Katz.

  The fundamental thesis of RAID is that in most common use cases, it is cheaper and more effective to redundantly store data on cheap disks, than to use/engineer high performance/durable disks.
- **RAID I** - Full disk replication. With RAID I two identical copies of all data is stored. If disk heads are not fully synchronized, this can decrease write performance, but increase read performance.
- **RAID V+** - Striping with error correction. In RAID V, 4 sequential block writes are placed on separate disks, then a 5th parity block is written by XORing the data blocks on the same stripe. RAID VI uses the EVENODD scheme to encode error correction. In general, Reed Solomon coding can be used for an arbitrary number of error correcting disks.



Note: Due to the large size of disks in practice, RAID V is no longer used in practice, because it is too likely that a second disk will fail while the first is recovering. RAID VI is usually combined with other error recovery techniques in practice.

- **Eventual Consistency** - A weaker form of a consistency guarantee. If a system is eventually consistent, it will converge to a consistent state over time.
- **Network File System (NFS)** - A distributed file system written by Sun. NFS is based on a stateless RPC protocol. Buffers are **write behind**. Few strong consistency guarantees on parallel writes. NFS is **eventually consistent**.
- **Andrew File System (AFS)** - A distributed file system written at CMU. Full files are buffered locally upon `open`. Buffers are **write back** and only flushed on `close`. File contents follow "last write wins" semantics.
- **Distributed Hash Table (DHT)** - A distributed hash table, or a distributed key value store, is a system which follows the semantics of a regular key value store, but in which the data is distributed over multiple machines.
- **Recursive Query** - A DHT query strategy in which requests are made to a central directory, which acts as a proxy to reroute the request to the appropriate data server. Recursive queries tend to have lower latency, and provide for an easier consistency model, but don't scale as well.
- **Iterative Query** - A DHT query strategy in which a lookup occurs to resolve a node name. The client then directly connects to the node to continue the query. Iterative queries tend to have higher latencies, and are more difficult to design for consistency, but provide more scale. Many GFS based KV Stores follow this model.
- **Consistent Hashing** - A technique for assigning a K/V Pair to a node. With consistent hashing, a new node can be added to a DHT while only moving a fraction (K/N) of the total keys. With consistent hashing Nodes are placed in the key space. A node is responsible for all the keys less than it, but greater than its predecessor. When a new node joins, it copies its necessariy data from its successor.
- **Chord** - Chord is a distributed lookup protocol for efficiently resolving the node corresponding to a key in a DHT. Chord uses a finger table which contains pointers to exponentially further nodes provide $\log(N)$ lookup time. It periodically updates the fingertable to provide for eventual consistency.



2 Logs and Journaling

You create two new files, F_1 and F_2, right before your laptop's battery dies. You plug in and reboot your computer, and the operating system finds the following sequence of log entries in the file system's journal.

1. Find free blocks x_1, x_2, ..., x_n to store the contents of F_1, and update the free map to mark these blocks as used.
2. Allocate a new inode for the file F_1, pointing to its data blocks.
3. Add a directory entry to F_1's parent directory referring to this inode.
4. *Commit*
5. Find free blocks y_1, y_2, ..., y_n to store the contents of F_2, and update the free map to mark these blocks as used.
6. Allocate a new inode for the file F_2, pointing to its data blocks.

What are the possible states of files F_1 and F_2 *on disk* at boot time?

- File F_1 may be fully intact on disk, with data blocks, an inode referring to them, and an entry in its parent directory referring to this inode.
- There may also be no trace of F_1 on disk (outside of the journal), if its creation was recorded in the journal but not yet applied.
- F_1 may also be in an intermediate state, e.g., its data blocks may have been allocated in the free map, but there may be no inode for F_1, making the data blocks unreachable.
- F_2 is a simpler case. There is no *Commit* message in the log, so we know these operations have not yet been applied to the file system.

Say the following entries are also found at the end of the log:

7. Add a directory entry to F_2's parent directory referring to F_2's inode.
8. *Commit*

How does this change the possible states of file F_2 on disk at boot time?

The situation for F_2 is now the same as F_1: the file and its metadata could be fully intact, there could be no trace of F_2 on disk, or any intermediate between these two states.

Say the log contained only entries (5) through (8) shown above. What are the possible states of file F_1 on disk at the time of the reboot?

We can now assume that F_1 is fully intact on disk. The log entries for its creation are only removed from the journal when the operation has been fully applied on disk.

What is the purpose of the *Commit* entries in the log?

- The *Commit* entry makes the creation of each file *atomic*. These changes to the file system's on-disk structures are either completely applied or not applied at all.
- The creation of a file involves multiple steps (allocating data blocks, setting up the inode, etc.) that are not inherently atomic, nor is the action of recording these actions in the journal, but we want to treat these steps as a single logical transaction.
- Appending the final *Commit* entry to the log (a single write to disk) *is* assumed to be an atomic operation and serves as the "tipping point" that guarantees the transaction is eventually applied.

When recovering from a system crash and applying the updates recorded in the journal, does the OS need to check if these updates were partially applied before the failure?

No. The operation for each log entry (e.g., updating an inode or a directory entry) is assumed to be *idempotent*. This greatly simplifies the recovery process, as it is safe to simply replay each committed transaction in the log, whether or not it was previously applied.

3 Distributed File Systems

Distributed file systems must provide the same access API as standard file systems. That access API quickly becomes non standard in the face of concurrent operations.

Compare the performance of AFS, NFS, EXT4, and EXT4 with the streaming api. Assume processes run on separate machines wherever it is applicable.

1. **open**

   In AFS, open is an expensive operation in terms of performance as it requires transferring the entire file over the network.

   In NFS, little data is transferred, but the client now begins to poll the server for changes to the file.

   Both open and fopen finish quickly for ext4.

2. **write**

   In AFS, this operation occurs faster than NFS, and requires no network activity. It operates entirely on the local copy of the file.

   In NFS, this operation is slow as it requires an RPC.

   With ext4 write is relatively slow compared to fwrite as it involves a syscall and i/o (fwrite might flush its buffer which involves syscall and i/o, but typically it will not).

   With the streaming api, this operation is relatively fast as it's likely userspace buffered.

3. **read**

   In AFS not network I/O will occur. It operates entirely on a local copy of the file.

   In NFS, and the streaming API expensive I/O is not likely to occur, but can occcur if the data being read is not cached.

   For ext4, this operation still requires a syscall and potentially an I/O operation.

4. **close**

   In AFS, this is an expensive operation. It involves transferring the entire file over the network.

   In NFS, little data is transferred since all buffers are already flushed, and the RPC is stateless.

   For ext4, close finishes relatively quickly. Very little I/O will occur.

   For the streaming api, the entire userspace buffer must be flushed to disk, which is potentially large.

Now compare the behavior of AFS, NFS, EXT4, and EXT4 with the streaming api (with a large buffer). Assume processes run on separate machines wherever it is applicable and that buffers are only flushed when filled and upon close.

1. Process A and Process B write to a file simultaneously, then Process A closes the file, then Process B closes the file.

   In AFS, only Process B's writes are reflected in the final file.

   In NFS, it is undefined what the contents of the file are.

   For ext4, it is undefined what the contents of the file are (writes aren't atomic).

   For the streaming api (assuming the buffer isn't flushed on write), only Process B's writes are

reflected in the final file.

2. Process A increments an integer (using read and write), 1 second later, Process B increments the integer too. Both processes close the file.

   In AFS, NFS, and the streaming api, the number is incremented once.

   In ext4, the number is incremented twice.

3. Process A increments an integer (using read and write), 1 minute later, Process B increments the integer too. Both processes close the file.

   In AFS and the streaming api, the number is incremented once.

   In ext4 and NFS the number is incremented twice.

   Note that these answers are different because NFS tends to poll on a timescale of seconds. Therefore it is likely that in the one second case NFS will not have pulled the updated, whereas after a minute, we expect the update to be reflected locally.

4. Process A writes a large amount fo data, then Process B writes a large amount of data. Then Process B closes the file, then Process A closes the file.

   In AFS only Process A's data is reflected in the file.

   In NFS, only Process B's data is reflected in the file.

   In ext4, only Process B's data is reflected in the file.

   In the streaming api, the file is mostly Process B's data, but ends with Process A's data.

4 Distributed Hash Tables

1. Consider a distributed key-value store using a directory-based architecture.

   Assume that keys are 256 bytes, values are 128 MiB, and each machine in the cluster has a 8 GiB/s network connection and the client has a unlimited amount of bandwidth. Finally assume the RTT between the directory and data machines is 2ms.

   If the RTT between the client and directory/data nodes is 64ms:

   (a) How long would it take to execute a single GET request using a recursive query?

   There would be 66 ms of latency + $\frac{2^{27}}{2^{33}} = 0.0156$ seconds of transfer time. The total time would be 82ms.

   (b) How long would it take to execute 2048 GET requests using recursive queries?

   Assuming we are able to carefully implement pipeline parallelism, we would still have 66 ms of latency. The transfer time would now be $\frac{2^{11} \times 2^{27}}{2^{33}} = 2^5 = 32$ seconds.

   (c) How long would it take to execute a single GET request using an iterative query?

   There would be 128 ms of latency and $\frac{2^{27}}{2^{33}} = 0.0156$ seconds of transfer time so the total time would be 143ms.

   (d) How long would it take to execute 2048 GET requests using an iterative query?

Assuming we can take advantage of pipeline parallelism for resolving nodes, it would take 64 ms + $\frac{2^{11} \times 2^8}{2^{33}}$ seconds to resolve all the keys.

We assume each data request can be executed in parallel, so it would take 64 ms + $\frac{2^{27}}{2^{33}} = 0.0156$ seconds to transfer all the data.

This is a total of 143 ms. Notice that in the recursive query case, the directory server was a much larger bottleneck.

Note it's reasonable to assume that we can execute our requests in parallel because we can use a cryptographic hash function to effectively map a key to a uniform random address in our hash table (assuming a non adversarial client). We are also asssuming that there are a large number of nodes, so no single node is limited by bandwidth.

Here's a proof that our objects should be conveniently distributed across our nodes: https://inst.eecs.berkeley.edu/ cs70/sp17/static/notes/n15.pdf

Pay special attention to the assumptions made, and whether or not they make sense in practice!

Also note that the peak bandwidth in this scenario exceeds 1TiB/s, which would require careful client design.

(e) Now imagine our client is located in the same datacenter, and the RTT between all components is the same (this is a common assumption when modeling datacenter topology).
Briefly describe how your results would change.

In broad terms, the RTT latency would now become far smaller than the actual transfer time, removing the previous bottleneck on latency for small transfers.

From a performance perspective, it is almost strictly better to use iterative querying under these assumptions about latency.

Note that there could still be other reasons to use a recursive query strategy even under these conditions. For example, one could try to simplify their design, ensure ordering/-timestamp their outputs, aggregate the data, etc, but perhaps this provides insight into why iterative querying is a popular design for DHT's within datacenters (such as GFS and the many systems based on it).

(f) What are some advantages and disadvantages to using a recursive query system?

Advantages: Faster, easier to maintain consistency.
Disadvantages: Scalability bottleneck at the directory/master server.

(g) What are some advantages and disadvantages to using an iterative query system?

Advantages: More scalable.
Disadvantages: Slower, harder to maintain consistency.

2. **Quorum consensus:** Consider a fault-tolerant distributed key-value store where each piece of data is replicated N times. If we optimistically return from a put() call as soon as we have received acknowledgements from W replicas, how many replicas must we wait for a response from in a get() query in order to guarantee consistency?

We must wait for at least $R > N - W$ responses. If we have any fewer than this number, there is a possibility that none of our responses contain the latest value for the key we are requesting.

3. In a distributed key-value store, we need some way of hashing our keys in order to roughly evenly distribute them across our servers. A simple way to do this is to assign key K to server i such that $i = \text{hash}(K) \mod N$, where N is the number of servers we have. However, this scheme runs into an issue when N changes — for example, when expanding our cluster or when machines go down. We would have to re-shuffle all the objects in our system to new servers, flooding all of our servers with a massive amount of requests and causing disastrous slowdown. Propose a hashing scheme (just an idea is fine) that minimizes this problem.

   We can treat the possible hash space as a circle, where every possible hash maps to some point on the circle. We then roughly evenly distribute our servers across this circle, and have each hash be stored on the next closest server on the circle. Then, when we add or remove servers, we need only move a portion of the objects on one server adjacent to the server we just added or removed. This technique is commonly known as **consistent hashing**.