Section 13: Distributed Systems

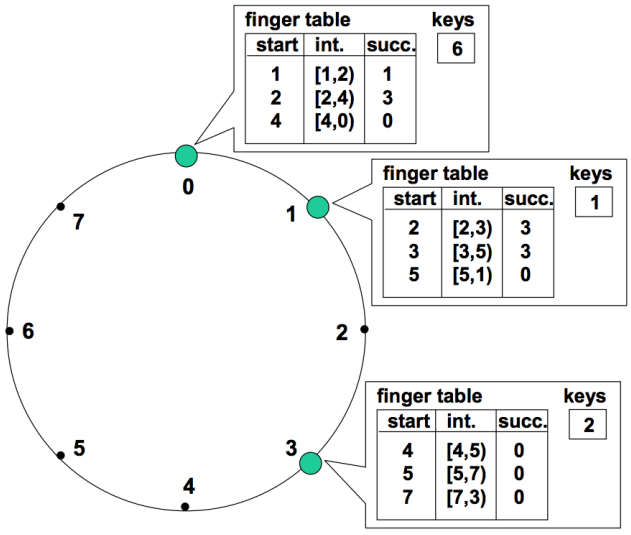
CS 162

November 27, 2020

Contents

1 Vocabulary

- **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`.
- **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.



- **Replication** A strategy for fault tolerance. With replication, one or more **replicas** are responsible for trying to maintain the same state.

2 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 of 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.

3 Distributed Key-Value Stores

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

Keys are 256 bytes, values are 128 MiB, each machine in the cluster has a 8 GiB/s network connection, and the client has a unlimited amount of bandwidth. The RTT between the directory and data machines is 2ms and the RTT between the client and directory/data nodes is 64ms.

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

There would be 66 ms of latency + $2 \times \frac{2^{27}}{2^{33}} = 0.0312$ seconds of transfer time. The total time would be 97.2ms.

ii) 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 $2 \times \frac{2^{11} \times 2^{27}}{2^{33}} = 2^5 = 64$ seconds.

iii) 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.

iv) 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 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.

v) 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).

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

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

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

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

b) **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.

c) 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 $i = hash(K)N$, 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**.

d) Consider a distributed key value store, in which for each KV pair, that pair is stored on a single node machine, and we use iterative querying (this is essentially what we looked at in the previous DHT problem).

(a) Describe some limitations of this system. In particular, focus on bandwidth and durability.

Bandwidth: This system could still be bottlenecked by bandwidth. If there is a popular key in the store, all reads/writes to that key will be directed to the same machine, and thus be limited by the bandwidth of a single machine.
Durability: Writes to this system are only durable, so long as the machine backing the node does not fail. If this machine is backed by RAID then it may have fault tolerance with respect to a disk failing, but if a flood damages the entire machine, data written to that node will be lost.

(b) Propose some strategies for overcoming these limitations.

One strategy for overcoming these limitations is replication.

If multiple replicas contain the same data, the directory server, or a load balancer, could direct GET requests to any machine which contains the data, thus the bandwidth of the node is now scaled by the number of replicas.

Note that this replication now introduces new challenges, such as consistency and consensus.