Section 13: Distributed Systems

CS162

April 26th, 2019

Contents

1 Vocabulary

- **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 first 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.
- **TPC/2PC** - Two Phase Commit is an algorithm that coordinates transactions between one coordinator (Master) and many slaves. Transactions that change the state of the slave are considered TPC transactions and must be logged and tracked according to the TPC algorithm. TPC ensures atomicity and durability by ensuring that a write happens across ALL replicas or NONE of them. The replication factor indicates how many different workers a particular entry is copied among. The sequence of message passing is as follows:

  ```
  for every worker replica and an ACTION from the master,
  origin [MESSAGE] -> dest :
  ---
  MASTER [VOTE-REQUEST(ACTION)] -> WORKER
  WORKER [VOTE-ABORT/COMMIT] -> MASTER
  MASTER [GLOBAL-COMMIT/ABORT] -> WORKER
  WORKER [ACK] -> MASTER
  ```

  If at least one worker votes to abort, the master sends a GLOBAL-ABORT. If all worker vote to commit, the master sends GLOBAL-COMMIT. Whenever a master receives a response from a slave, it may assume that the previous request has been recognized and committed to log and is therefore fault tolerant. (If the master receives a VOTE, the master can assume that the worker has logged the action it is voting on. If the master receives an ACK for a GLOBAL-COMMIT, it can assume that action has been executed, saved, and logged such that it will remain consistent even if the worker dies and rebuilds.)
- **RPC** - Remote procedure calls (RPCs) are simply cross-machine procedure calls. These are usually implemented through the use of stubs on the client that abstract away the details of the call. From the client, calling an RPC is no different from calling any other procedure. The stub handles the details behind marshalling the arguments to send over the network, and interpreting the response of the server.
- **General's Paradox** - The idea that there is no way to guarantee that two entities do something simultaneously if they can only send messages to each other over an unreliable network. There is no way to be sure that the last message gets through, so one entity can never be sure that the other entity will act at a specific time.

2 Problems

2.1 Distributed Systems

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

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

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

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

Advatanges: 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 $i = \text{hash}(K) \mod N$, 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**.

2.2 A Simple 2PC

Suppose you had a remote storage system composed of a client (you), a single master server, and a single worker server. All units are separated from each other and communicate using RPC. There is no caching or local memory; all requests are eventually serviced using the backing store (disk) of the worker. The worker guards itself against failure by committing entries to a non-volatile log that never gets deleted in the event of a crash. The system only understands PUT(VALUE) and DEL(VALUE) commands, where VALUE is an arbitrary string. Calls to DEL on values that dont exist cause the worker to VOTE-ABORT.

Suppose you issue the following sequence of commands. Recall that the correct sequence of message passing is CLIENT - MASTER - worker - MASTER - CLIENT. Calls to DEL on values that don't exist cause VOTE-ABORT.

- PUT(I LOVE)

- PUT(OPERATING SYSTEMS)

- DEL(I LOVE)

- DEL(I LOVE)

- PUT(GOBEARS)

What is the sequence of messages sent and received by the MASTER server? List communications with the worker only. Your answer should be a list of the form:

SEND: PUT(XXX)

RECIEVE: VOTE-XXX

SEND: DEL(XXX)

...

```
SEND: PUT(I LOVE)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: PUT(OPERATING SYSTEMS)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: DEL(I LOVE)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
SEND: DEL(I LOVE)
RECEIVE: VOTE-ABORT
SEND: GLOBAL-ABORT
RECEIVE: ACK
SEND: PUT(GOBEARS)
RECEIVE: VOTE-COMMIT
SEND: GLOBAL-COMMIT
RECEIVE: ACK
```

What is the sequence of messages committed to the log of the worker?

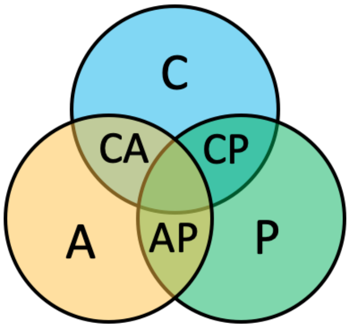
```
PUT(I LOVE)
COMMIT
PUT(OPERATING SYSTEMS)
COMMIT
DEL(I LOVE)
COMMIT
```

```
DEL(I LOVE)
ABORT
PUT(GOBEARS)
COMMIT
* notice that there are no read commands (i.e. GET) in this toy example,
but there would be no need to commit reads anyways as they do not modify
the state of the server.
```

2.3 CAP Theorem

This question is not in scope, but may be an interesting concept to think about.

One of the most famous and relevant distributed systems concepts today is the CAP theorem, first conjectured by Professor Eric Brewer (Berkeley) in 1998. It states that it is impossible for a distributed system to simultaneously have all three of the following properties:



Consistency: every read must be correct, and so can either return the most recent write or an error

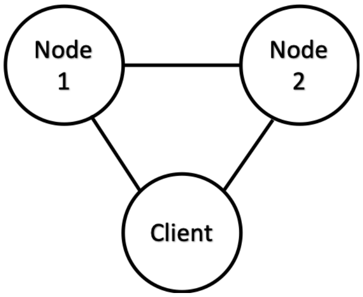
Availability: every request must not return an error, but read requests do not necessarily have to return the most recent write

Partition Tolerance: the system must be able to continue to operate despite an arbitrary number of messages between nodes dropped or delayed, such as in the event of a complete network partition.

Many systems are architected to have two of the three properties, and some even have just one. For example, a system that is Consistent and Available (CA) must always return the most recent write for every read, without fail.

Because of the nature of distributed systems, network partitions are considered an inevitable fact of life. Thus many distributed applications are forced to be either CP or AP, forfeiting either C or A in favor of partition tolerance.

1. The proof for the CAP theorem, while conceptually simple, was not formulated until 2002 by Gilbert and Lynch (MIT). Imagine you had a distributed database with two nodes, and a client that can read or write to either of the nodes.



Intuitively, argue why this system cannot achieve all three CAP properties.

For the system to have partition tolerance, then it must exhibit consistency and availability even if the link between Node 1 and Node 2 fails. With the link removed, the client can make a write request for some key K to Node 1. Then, it makes a read request for K at Node 2. Because the link between Node 1 and Node 2 is gone, Node 1 has no way to communicating to Node 2 about the value of K. Thus Node 2 must either return an error (because it knows somehow that it may not have the most up to date value) or it can return an older version of K. This violates either availability in the former case and consistency in the latter case. By contradiction, this system cannot be CAP.

2. Key value stores sometimes use Two Phase Commit to update the value of its keys. Which CAP properties does the 2PC protocol exhibit?

In the event of a network partition, such as if some subset of nodes cannot be reached or have failed, 2PC simply blocks until timeout and then sends out global abort messages. Therefor it cannot produce meaningful work in the event of a partition, and thus does not hold the P property. It does however hold C because consistency is the whole point of having 2PC. For availability, we look at 2PC under normal circumstances (since we already ruled that it does not have P). 2PC requests are simply forwarded to the workers, which by the fact that there are no failures, will return correctly and therefore giving Two Phase Commit A.

3. The consistency defined for the CAP theorem refers to *strong* consistency, of which there are two variants:

   - Linearizable consistency: all operations appear to have executed atomically in an order that is consistent with the global real-time ordering of operations.
   - Sequential consistency: operations can be reordered, but this new order must be consistent across all nodes.

   There are also variants of *weak* consistency. A famous one is eventual consistency:

   - Eventual consistency: if no new updates are made to a key, then eventually all nodes will agree on the last value for that key.

   Is it possible to achieve all three CAP properties if C referred to weak consistency instead of strong consistency?

The reason why an AP system cannot also be strongly consistent is that after an update for a particular key, one partition will always have a stale value for that key since the new value cannot be propagated to that partition. However if we can assume that after some time the network partition will be resolved, then it is possible for an AP system to have eventual consistency. Cassandra is a KV store that is an example of an AP system with eventual consistency.