162 Dis 13 Notes

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 slaves a particular entry is copied among. The sequence of message passing is as follows:

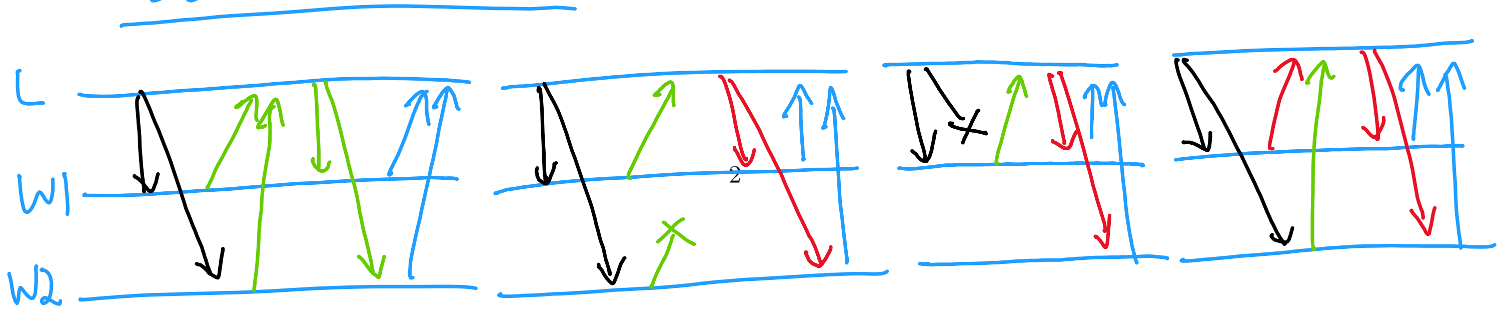
```
for every slave replica and an ACTION from the master,
origin [MESSAGE] -> dest :
---
MASTER [VOTE-REQUEST(ACTION)] -> SLAVE
SLAVE [VOTE-ABORT/COMMIT] -> MASTER
MASTER [GLOBAL-COMMIT/ABORT] -> SLAVE
SLAVE [ACK] -> MASTER
```

  If at least one slave votes to abort, the master sends a GLOBAL-ABORT. If all slaves 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 slave 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 slave 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.

Possible Executions



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?

Same

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

Same

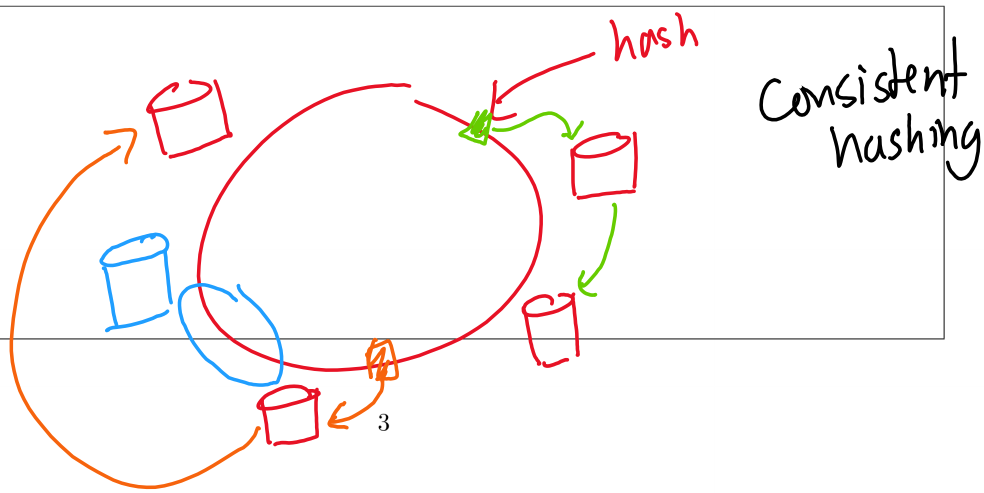
b) **Quorum consensus:** Consider a fault-tolerant distributed key-value store where each piece of data is replicated N times. If we optimistically return from a put() call as soon as we have received acknowledgements from W replicas, how many replicas must we wait for a response from in a get() query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

at least one has data

2 options

1) "agree" in that no one disagrees → 1 correct & R-1 no responses

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.



2.2 A Simple 2PC

Suppose you had a remote storage system composed of a client (you), a single master server, and a single slave 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 slave. The slave 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 slave to VOTE-ABORT.

Suppose you issue the following sequence of commands. Recall that the correct sequence of message passing is CLIENT - MASTER - SLAVE - MASTER - CLIENT. Calls to PUT on values that already 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 slave only. Your answer should be a list of the form:

SEND: PUT(XXX)

RECIEVE: VOTE-XXX

SEND: DEL(XXX)

...

Send command
receive all vote commits
Send global commit
receive ack
⋮

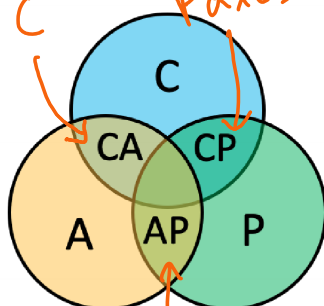
What is the sequence of messages committed to the log of the slave?

PUT (I love)
Commit
PUT (OS)
commit
DEL (I love)
Commit
DEL (I love)
Abort
PUT (Gobears)
commit

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:



2PC — Paxos — Gossip (weak C)

CP: don't have ⇒ error; have ⇒ correct (from network fail (P))

AP: don't have ⇒ send old val; have ⇒ correct

CA: always send back right data, never fail in connection

Consistency: every read must be correct, and so can either return the most recent write or an error
Availability: every read must not return an error, but does 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.

- split ties b/w the two

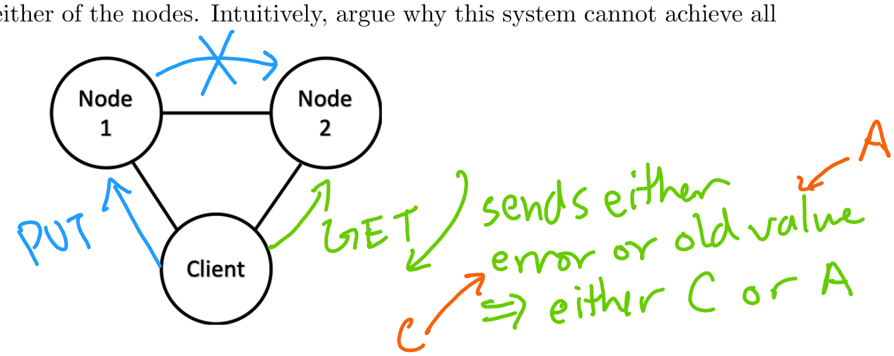
System does not shut down in this case

Many systems are architecture to have two of the three properties, and some even have just one or none. 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.

- might not be right, but it's trying.
  - going for partial credit on a midterm

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.

Assume P → need both C & A
PUT Node 1, Get Node 2

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

Definitely not P (no response → abort)
⇒ assume not P – all works perfactly
C – consistency is the point of 2PC – everyone agrees
A – all actions succeed ⇒ always get a value

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

- Linearizable consistency: all operations are executed atomically in the real time order in which they are created by the clients.
- 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?

definitely can do AP - get vals, return what you have, even if PUT were lost

eventual consistency ⇒ YES CAP

↳ Gossiping - send neighbors your value & hope that spreads the correct answer