

Section 12: Journaling, 2PC

CS 162

Nov 19, 2021

Contents

1	Vocabulary	2
2	Logs and Journaling	5
3	Two-Phase Commit	6

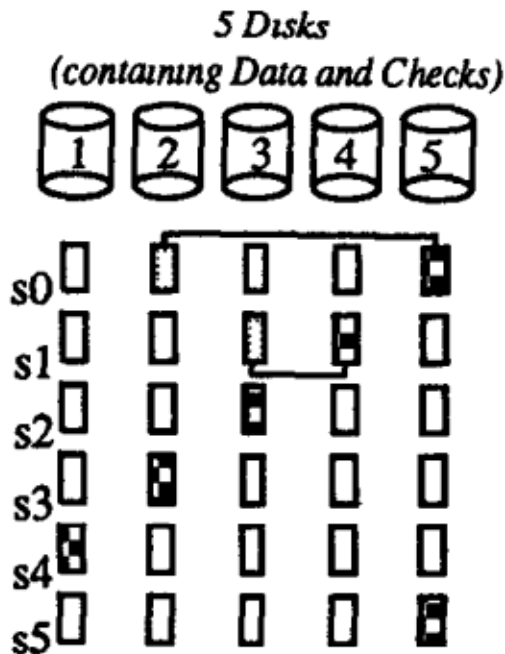
1 Vocabulary

- **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* - Concurrent 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.

- **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.
- **Failover** A fault tolerance procedure invoked when a component fails. Typically this involves switching over to, or promoting a secondary.
- **2PC** - Two Phase Commit (2PC) is an algorithm that coordinates transactions between one coordinator and many workers. Transactions that change the state of the worker are considered 2PC transactions and must be logged and tracked according to the 2PC algorithm. 2PC 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 coordinator,  
origin [MESSAGE] -> dest :  
---  
COORDINATOR [VOTE-REQUEST(ACTION)] -> WORKER  
WORKER [VOTE-ABORT/COMMIT] -> COORDINATOR  
COORDINATOR [GLOBAL-COMMIT/ABORT] -> WORKER  
WORKER [ACK] -> COORDINATOR
```

If at least one worker votes to abort, the coordinator sends a GLOBAL-ABORT. If all worker vote to commit, the coordinator sends GLOBAL-COMMIT. Whenever a coordinator receives a response from a worker, it may assume that the previous request has been recognized and committed to log and is therefore fault tolerant. (If the coordinator receives a VOTE, the coordinator can assume that the worker has logged the action it is voting on. If the coordinator 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.)

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, \dots, 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, \dots, 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 Two-Phase Commit

Quorum consensus is able to provide weak consistency. For this section, assume the DHT backs each node with multiple replicas. Further, assume that these machines use a two phase commit protocol to commit information in a strongly consistent manner.

1. 2PC requires a single coordinator and at least one worker. By default, all replicas can be workers. How should the coordinator be picked?

The general problem here is leader election - the bootstrap process of choosing and agreeing a leader in distributed systems.

Naively, we could pick a single machine to be the coordinator for all transactions and hard code that machine as the leader. If the leader crashes, then we must wait for that leader to restart before the system can make any forward progress.

Modern leader election algorithms do not rely on a hard coded leader but rather allow the workers to choose who they want as their leader. In the event that the leader fails (as detected by some form of health checking), the subset of the machines that are still connected would rerun the algorithm and find a new leader.

This provides better availability because a new leader can be choose on the fly, but it also introduces further complexities. For example what if there was a network partition and a single cluster of works split into two where each one elects their own leader. Then the partition heals how do you deal with the fact that there are now two leaders.

Additionally in any situation where we choose a single leader, we would be bottle necked by its upload bandwidth, since it would have to send the request to all replicas. To mitigate this problem, we could pick a leader at random for each request. This would spread resource usage for the dominant resource (upload bandwidth).

Note we typically don't consider consider GFS style chaining. It doesn't fit in the byzantine generals model, and may not be fast if we can't do a shortest hop tour (in the event of a failed machine).

2. Briefly describe the messages that the **coordinator** will send and receive in response to a PUT. Also describe when and what would be logged.

- RECEIVE (from client): A PUT query
- LOG (to journal): PREPARE query
- SEND (to workers): N PREPARE PUT messages
- RECEIVE (from workers): COMMIT or ABORT per worker
- LOG (to journal): GLOBAL-COMMIT or GLOBAL-ABORT
- SEND (to workers): GLOBAL-COMMIT if it receives COMMIT from all workers, otherwise GLOBAL-ABORT
- RECEIVE (from workers): ACK
- SEND (to client): SUCCESS if it sent GLOBAL-COMMIT, otherwise FAIL.

3. Briefly describe the messages that a **worker** will send and receive.

- RECEIVE (from coordinator): A PREPARE PUT
- LOG (to journal): PREPARE query
- SEND (to coordinator): COMMIT or ABORT
- RECEIVE (from coordinator): GLOBAL-COMMIT or GLOBAL-ABORT
- LOG (to journal): GLOBAL-COMMIT or GLOBAL-ABORT
- SEND (to coordinator): ACK

4. Under the current model, multiple PUT queries could be sent to separate coordinator machines. Propose set of worker routines which can handle this. In particular, consider the case in which 2 coordinators receive PUT queries for **the same key but different values**. The state of the system should remain consistent.

Upon receiving a PREPARE PUT request, `try_acquire` a lock on the key.

If acquire fails, ABORT, otherwise COMMIT.

Upon receiving a GLOBAL-COMMIT, set the value then release the lock. Release the lock without setting the value on a GLOBAL-ABORT.

The state of the system will always be consistent now. Note that a potentially unintuitive outcome of 2PC here is that both queries could fail here, leaving the key empty despite a PUT request.

One potential solution to this problem (which is out of scope) while maintaining strong consistency could include using PAXOS to determine an order of these queries, and accept only the first.

Note: Many popular DHTs in the real world take different approaches to handling this edge case. Most of these DHTs are eventually consistent.