Discussion 10: Distributed Systems

April 19, 2024

Contents

1 Transactions

A more general way to handle reliability is through the use of **transactions**, indivisible units which execute independently from other transactions. Transactions typically follow the **ACID properties**.

Atomicity
A transaction must occur in its entirety or not at all.

Consistency
A transaction takes system from one consistent state (i.e. meets all integrity and correctness constraints) to another.

Isolation
Each transaction must *appear* to execute on its own.

Durability
A committed transaction's changes must persist through crashes.

The idea of a transaction is similar to that of a critical section with the newly added constraint of durability. Each entry in a transaction needs to be **idempotent**, which have the same effect when executed once or many times (i.e. $f(f(x)) = f(x)$).

Transactions are recorded on **logs/journals** which are stored on disk for persistence. Writes to a Log are assumed to be atomic. Logs will typically use a circular buffer as its data structure, which maintains a head and tail pointer.

Transactional file systems use the idea of transactions and logs to make the file system more reliable. There are two main types of transactional file systems: journaling and log structured.

Journaling file systems use **write-ahead logging (WAL)** where log entries are written to disk *before* the data gets modified. During the preparation phase, all planned updates are appended to the log. Each log entry is tagged with the transaction id. During the commit phase, a commit record is appended to the log, indicating the transaction must happen at some point. Next, the write back will take place *asynchronously* after the commit phase where the transaction's changes will be applied to persistent storage. In the background, garbage collection will take place to clean up fully written back transactions. When recovering from a crash, only the committed transactions are replayed to restore the state of the file system.

Log structured file systems (LFS) use the log as the storage. The log becomes one contiguous sequence of blocks that wrap around the whole disk.

1.1 Concept Check

1. Why does each entry in a transaction need to be idempotent?

2. Consider a file system with a buffer cache. Your program creates a file called `pintos.bean`. While the changes have been logged with a commit entry, the tail of the log still points to before the start of the log entry corresponding to creating `pintos.bean`. Assuming no further disk reads or writes have happened, if your program wants to read `pintos.bean`, will it need to scan through the logs?

3. What is the purpose of a commit entry in a log?

2 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 $x_1, x_2, \ldots, x_n$ 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 $y_1, y_2, \ldots, y_n$ 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.

You may assume a single write to disk is an atomic operation.

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

2. 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?

3. 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?

4. 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?

3 Distributed Systems

3.1 Concept Check

1. The vanilla implementation of 2PC logs all decisions. How could 2PC be optimized to reduce logging?

2. 2PC exhibits blocking behavior where a worker can be stalled until the coordinator recovers. Why is this undesirable?

3. An interpretation of the End to End Principle argues that functionality should only be placed in the network if certain conditions are met.

   Only If Sufficient

   Don't implement a function in the network unless it can be completely implemented at this level.

   Only If Necessary

   Don't implement anything in the network that can be implemented correctly by the hosts.

   Only If Useful

   If hosts can implement functionality correctly, implement it in the network only as a performance enhancement.

   Consider the example of the reliable packet transfer: making all efforts to ensure that a packet sent is not lost or corrupted and is indeed received by the other end. Using each of the three criteria, argue if reliability should be implemented in the network.

4. Why would you ever want to use UDP over TCP?

3.2 Two Phase Commit

Consider a system with one coordinator (C) and three workers (W_1, W_2, W_3). The following latencies are given for each worker.

Worker	Send/Receive (each direction)	Log
W_1	400 ms	10 ms
W_2	300 ms	20 ms
W_3	200 ms	30 ms

You may assume all other latencies not given are negligible. C has a timeout of 3 s, log latency of 5 ms, and can communicate with all workers in parallel.

1. What is the minimum amount of time needed for 2PC to complete successfully?

2. Consider that all three workers vote to commit during the preparation phase. The coordinator broadcasts a commit decision to all the workers. However, W_2 crashes and does not recover until immediately after the coordinator's timeout phase. Does this transaction commit or abort? What is the latency of this transaction, assuming no further failures?