Discussion 10: Distributed Systems

April 21, 2023

Contents

1 Reliability

Availability

Availability is the probability that the system can accept and process requests. This is measured in "nines" of probability (e.g. 99.9% is said to be "3-nines of availability").

Durability

Durability is the ability of a system to recover data despite faults (i.e. fault tolerance). It's important to note that durability does not necessarily imply availability.

When making a file system more durable, there are multiple levels which we need to concern ourselves with. For small defects in the hard drive, Reed-Solomon error correcting codes can be used in each disk block. When using a buffer cache or any other delayed write mechanism, it's important to make sure dirty data gets written back to the disk. To combat unexpected failures or power outages, the computer can be built with a special, battery-backed RAM called non-volatile RAM (NVRAM) for dirty blocks in the buffer cache.

To make sure the data survives in the long term, it needs to be replicated to maximize the independence of failures. **Redundant Array of Inexpensive Disks (RAID)** is a system that spreads data redundantly across multiple disks in order to tolerate individual disk failures. RAID 1 will **mirror** a disk onto another "shadow" disk. Evidently, this is a very expensive solution as each write to a disk actually incurs two physical writes. RAID 5 will stripe data across n multiple disks to allow for a single disk failure. Each stripe unit consists of $n-1$ blocks and one **parity block**, which is created by XOR-ing the $n-1$ blocks. To recover from a disk failure, the system simply needs to XOR the remaining blocks.

Reliability

Reliability the ability of a system or component to perform its required functions under stated conditions for a specified period of time. This means that the system is not only up (i.e. availability), but also performing its jobs correctly. Reliability includes the ideas of availability, durability, and security.

One approach taken by FAT and FFS (in combination with `fsck`) is **careful ordering and recovery**. For instance, creating a file in FFS may be broken down into the following steps

1. Allocate data block.
2. Write data block.
3. Allocate inode.
4. Write inode block.
5. Update free map.
6. Update directory entry.
7. Update modify time for directory entry.

To recover from a crash, `fsck` might take the following steps.

1. Scan inode table.
2. If any unlinked files (not in any directory), delete or put in lost and found directory.
3. Compare free block bitmap against inode trees.
4. Scan directories for missing update/access times.

It's important to note that this is not a foolproof method. While there are a few ways that failures can happen in spite of this method, the file system will be recoverable most of the times.

The other approach is to use **copy on write (COW)**. Instead of updating data in-place, new versions are written to a new location on disk, and the appropriate mappings and references to these data are subsequently updated. Since the mappings and references are updated last, this allows for easy recovery if the system crashes sometime in the middle of updating data since the old data and mapping will still be in tact. Furthermore, data is being only added, not modified, so batch updates and parallel writes can help improve performance.

1.1 Concept Check

1. What benefit with regards to read bandwidth might you see from using RAID 1?

   Read bandwidth can be doubled since there are two copies of the same data.

2. What is the minimum number of disks to use RAID 5?

   3. A disk is recovered by XOR-ing at least two other disks. It's important to note that the RAID level is not an indication of how many disks are needed.

3. RAID 4 had a dedicated disk with all the parity blocks. On the other hand, RAID 5 distributes the parity blocks across all disks in a round robin fashion. Why is this approach beneficial in terms of write bandwidth?

   When updating data, the parity block always needs to be written to. If all the parity blocks are in one disk like in RAID 4, this becomes a bottleneck as multiple writes to disk ultimately contend on one disk. As a result, it's better to distribute the parity blocks evenly such that more writes can happen without contending for the same disk.

4. How can COW help with write speeds?

   COW can transform random I/O into sequential I/O. For instance, take the example of appending a block to a file. In a traditional in-place system like FFS, the free space bitmap, file's inode, file's indirect block, and file's data block all need to be updated. On the other hand, COW could just find sequential unused blocks in disk and write the new bitmap, inode, indirect block, and data block.

2 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.

2.1 Concept Check

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

   On recovery, all the entries from a committed transaction are replayed. There is a possibility that the data corresponding to an entry was already modified prior to a crash. A non-idempotent operation would put us in an undesired state.

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?

   No. The buffer cache will still hold the blocks corresponding to `pintos.bean`.

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

   The commit entry makes each transaction atomic. These changes to the file system's on-disk structures are either completely applied or not applied at all. For instance, the creation of a file involves multiple steps (e.g. allocating data blocks, setting up the inode) 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 (i.e. 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.

3 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?

   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. 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 for F_2, so we know these operations have not yet been applied to the file system.

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?

   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.

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?

   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.

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?

   No. The operation for each log 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.

4 Distributed Systems

4.1 Concept Check

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

Abort decisions can be ignored and not logged with the idea of presumed abort. On recovery, if there is nothing logged, then we can simply assume that the decision made was to abort.

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

If a worker is blocked on this coordinator, then it may be holding resources that other transactions may need.

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.

Only If Sufficient
No. It is not sufficient to implement reliability in the network. The argument here is that a network element can misbehave (i.e. forwards a packet and then forget about it, thus not making sure if the packet was received on the other side). Thus the end hosts still need to implement reliability, so it is not sufficient to just have it in the network.

Only If Necessary
No. Reliability can be implemented fully in the end hosts, so it is not necessary to have to implement it in the network.

Only If Useful
Sometimes. Under circumstances like extremely lossy links, it may be beneficial to implement it in the network.

Lets say a packet crosses 5 links and each link has a 50% chance of losing the packet. Each link takes 1 ms to cross and there is an magic oracle tells the sender the packet was lost. The probability that a packet will successfully cross all 5 links in one go is $(1/2)^5 = 3.125\%$. This means the end hosts need to try 32 times before it expects to see the packet make it through, taking up to # of tries × max # of links per try $= 32 \times 5 = 160$ ms.

Likewise at each hop, if the router itself is responsible for making sure the packet made it to the next router, each router would know if the packet was dropped on the link to the next router. Thus each router only has to send the packet until it reaches the next router, which will be twice on average. So to send this packet, it will take on average # of tries per link # number of links $= 2 \times 5 = 10$ ms. This is a huge boost in performance, which makes it useful to implement reliability in the network under some cases.

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

UDP is used in applications that prioritize speed and low overhead, and either don't care about being "lossy" or implements their own protocol for reliability. For example, in streaming audio or video, it doesn't matter if some packets are lost or corrupted, as long as most of the packets are sent, the user will still be able to hear/see the data well enough. Another example would be any application where real time data is very important (e.g. real time news, weather, stock price tracking, etc), and we don't have time for anything beyond best effort delivery.

4.2 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?

For each phase, the worker needs to receive the message, log a result, and send back a message. Since each worker can operate in parallel, only the longest latency matters. Using the latencies given, W_1 has the longest round trip time latency with $400 + 10 + 400 = 810$ ms. Therefore, the two phases will require 1620 ms. However, we also need to add in the time that it takes for the coordinator to log the global decision, so the minimum amount of time is 1625 ms. The reason this is the minimum amount of time is because this assumes no failures.

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?

The transaction still commits since a commit decision was made by the coordinator. The preparation phase will take 810 ms as calculated before, and the commit decision needs to be logged which takes 5 ms. The first try of the commit phase will take 3000 ms (i.e. the timeout). The second try will only take $300 + 20 + 300 = 620$ ms because W_2 is the only worker that needs to commit; all other workers succeeded on the first try. In total, the latency of this transaction is $810 + 5 + 3000 + 620 = 4435$ ms.

4.3 Primes

Edward wants to build a RPC system for the following two procedures.

```
/* Returns the ith prime number (0-indexed). */
uint32_t ith_prime(uint32_t i);

/* Returns 1 if x and y are comprime, 0 otherwise. */
uint32_t is_coprime(uint32_t x, uint32_t y);
```

In particular, Edward is looking to setup the server side. You may assume the actual logic for both procedures have already been implemented.

1. Edward notices that the arguments required are either 4 or 8 bytes. As a result, he believes he can handle either case by attempting to read 8 bytes with the code below.

   ```
   /* Assume dest has enough space allocated. */
   void read_args(int sock_fd, char *dest) {
     int byte_len = 0;
     int read_bytes = 0;
     while ((read_bytes = read(sock_fd, dest, 8 - byte_len)) > 0) {
       byte_len += read_bytes;
     }
   }
   ```

   However when Edward implements it, he notices that for some inputs the server appears to be stuck. Why might this be happening and for which inputs could this happen?

   > The `read` is trying to return 8 bytes from the socket or indicate there is no more data. However, the socket will not return 0 unless the socket is closed. As long as the connection stays open, the `read` will block instead.

2. Edward realizes his previous solution was insufficient, so he decide to implement a slightly more complicated protocol.

   The client will perform the following.

   (a) Send an identifier for the function it wants as an integer (0 for `ith_prime`, 1 for `is_coprime`).

   (b) Send all bytes for all the arguments.

   The server will then perform the following.

   (a) Read identifier.

   (b) Use identifier to allocate memory and set read size.

   (c) Read arguments.

   ```
   /* Converts NETLONG from network byte order to host byte order. */
   uint32_t ntohl(uint32_t netlong);

   /* Converts NETLONG from host byte order to network byte order. */
   uint32_t htonl(uint32_t hostlong);

   void receive_rpc (int sock_fd) {
     /* Read in procedure identifier. */
     uint32_t id;
     int bytes_read = 0, cur_read = 0;
     while (_______________) {
       bytes_read += cur_read;
     }
     id = _______________;

     /* Get sizes and allocate space for arguments and return values. */
     char *args, *rets;
     size_t arg_bytes, ret_bytes;
     get_sizes(id, &args, &arg_bytes, &rets, &ret_bytes);

     /* Read in arguments. */
     bytes_read = 0;
     while (_______________) {
   ```

```
    bytes_read += cur_read;
  }

  /* Call appropriate server stub stub function based on id. */
  call_server_stub(id, args, arg_bytes, rets, ret_bytes);

  /* Write return values. */
  int bytes_written = 0, cur_written = 0;
  while (_______________) {
      bytes_written += cur_written;
  }

  /* Clean up socket. */
  _______________;
}
```

```
void receive_rpc (int sock_fd) {
  /* Read in procedure identifier. */
  uint32_t id;
  int bytes_read = 0, cur_read = 0;
  while ((cur_read = read(sock_fd, ((char *) &id) + bytes_read,
                         sizeof(uint32_t) - bytes_read)) > 0) {
    bytes_read += cur_read;
  }
  id = ntohl(id);

  /* Get sizes and allocate space for arguments and return values. */
  char *args, *rets;
  size_t arg_bytes, ret_bytes;
  get_sizes(id, &args, &arg_bytes, &rets, &ret_bytes);

  /* Read in arguments. */
  bytes_read = 0;
  while ((cur_read = read(sock_fd, &args[bytes_read],
                         arg_bytes - bytes_read)) > 0) {
    bytes_read += cur_read;
  }

  /* Call appropriate server stub stub function based on id. */
  call_server_stub(id, args, arg_bytes, rets, ret_bytes);

  /* Write return values. */
  int bytes_written = 0, cur_written = 0;
  while ((cur_written = write(sock_fd, &rets[bytes_written],
                          ret_bytes - bytes_written)) > 0) {
      bytes_written += cur_written;
  }

  /* Clean up socket. */
  close(socket_fd);
}
```

3. Finally, Edward want to implement the call stubs, which are function wrappers between each individual function we support and the generic RPC library. The following are the *client* stubs for our procedures.

```
uint32_t ith_prime_cstub(struct addrinfo *addrs, uint32_t i) {
  uint32_t prime_val;
  i = htonl(i);
  call_rpc(addrs, RPC_PRIME, (char *)&i, 1, (char *) &prime_val, 1);
  return ntohl(prime_val);
}

uint32_t is_coprime_cstub(struct addrinfo *addrs, uint32_t x, uint32_t y) {
  uint32_t coprime_val;
  uint32_t args[2] = {htonl(x), htonl(y)};
  call_rpc(addrs, RPC_COPRIME, (char *) args, 2, (char *) &coprime_val, 1);
  return ntohl(coprime_val);
}
```

Implement the server stubs for our procedures.

```
void ith_prime_sstub (char *args, char *rets) {
  ______________________________;
  ______________________________;
  ______________________________;
  ______________________________;
}
void is_coprime_sstub (char *args, char *rets) {
  ______________________________;
  ______________________________;
  ______________________________;
  ______________________________;
  ______________________________;
}
```

```
void ith_prime_sstub (char *args, char *rets) {
  uint32_t* arg_ptr = (uint32_t *) args;
  uint32_t i = ntohl(args_ptr[0]);
  uint32_t result = htonl(ith_prime(i));
  memcpy(rets, &result, sizeof(uint32_t));
}
void is_coprime_sstub (char *args, char *rets) {
  uint32_t* arg_ptr = (uint32_t *) args;
  uint32_t x = ntohl(args_ptr[0]);
  uint32_t y = ntohl(args_ptr[1]);
  uint32_t result = htonl(is_coprime(x, y));
  memcpy(rets, &result, sizeof(uint32_t));
}
```