Section 11: Fault Tolerance

April 15, 2020

Contents

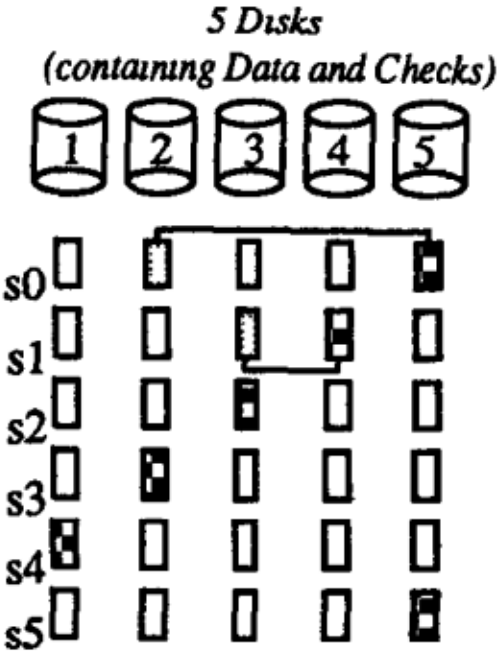
1 Vocabulary

- **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* - Concurent 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.

- **Checksum** - A mathematical function which maps a (typically large) input to a fixed size output. Checksums are meant to detect changes to the underlying data and should change if changes occur to the underlying data. Common checksum algorithms include CRC32, MD5, SHA-1, and SHA-256.
- **Replication** - Replication or duplication is a common technique for preserving data in the face of disk failure or corruption.

  If a disk fails, data can be read from the replica. If a sector is corrupted, it will be detected in the checksum. The data can then be read from another replica.
- **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.
- **2PC** - Two Phase Commit (2PC) is an algorithm that coordinates transactions between one coordinator (Master) and many slaves. Transactions that change the state of the slave 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 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.)

- **Endianness** - The order in which the bytes are stored for integers that are larger than a byte. The two variants are big-endian where byte the most significant byte is at the lowest address and the least significant byte is at the highest address and little-endian where the least significant byte is at the lowest address and the most significant byte is at the highest address. The network is defined to be big-endian whereas most of your machines are likely little-endian.
- `uint32_t htonl(uint32_t hostlong)` - Function to abstract away endianness by converting from the host endianness to the network endianness.
- `uint32_t ntohl(uint32_t netlong)` - Function to abstract away endianness by converting from the network endianness to the host endianness.
- **RPC** - Remote procedures calls are a technique for distributed computation through a client server model. This process effectively calls a procedure on a possibly remote server from a client by wrapping communication over the network with wrapper stub functions. This six steps are:
  1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
  2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshaling.
  3. The client's local operating system sends the message from the client machine to the server machine.
  4. The local operating system on the server machine passes the incoming packets to the server stub.
  5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshaling.
  6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction

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, ..., 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, ..., 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?

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?

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?

What is the purpose of the *Commit* entries in the log?

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 Two-Phase Commit

In order to explore 2 phase commit in we will use a replicated Key-Value Store as an example. In this KV Store, PUT operations must be atomic and strongly consistent across all workers.

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

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

3. 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.

4 RPC

To explore the process for performing RPC we will consider implement the server end for two procedures. We want to implement the server side of an RPC version of the following code

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

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

Assume the server has already implemented `ith_prime` and `is_coprime` locally.

4.1 Selecting Functions

As a first step we receive data from the client. How do we decide which procedure we are executing? Provide a sample header file addition that could be used to indicate this.

4.2 Reading Arguments

When examining the arguments for your two functions you notice that the arguments require either 8 or 4 bytes, so you believe you can handle either case by attempting to read 8 bytes using 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 (fd, dest, 8 - byte_len)) > 0) {
        byte_len += read_bytes
    }
}
```

However when you implement it you notice that for some inputs your server appears to be stuck? Why might this be happening and for which inputs could this happen?

4.3 Handling RPC

Realizing your previous solution was insufficient you decide to implement a slightly more complicated protocol. You settle on the following steps for the client:

1. The client sends an identifier of the function it wants as an integer (0 for `ith_prime`, 1 for `is_coprime`).
2. The client sends all the bytes for all the arguments.

The server then takes the following steps:

1. The server reads the identifier.
2. The server uses the identifier to allocate memory and set the read size.
3. The server reads the remaining arguments.

Complete the following function to implement the server side of handling data. You may find `ntohl` useful.

```
// Function to implement the server side of the protocol.
// Returns whether or not it was successful and closes the socket when finished.
// Assume get_sizes loads all our sizes based on id and call_server_stub selects
// our host function
void receive_rpc (int sock_fd) {
    uint32_t id;
    char *args;
    size_t arg_bytes;
    char *rets;
    size_t ret_bytes;
    int bytes_read = 0;
    int bytes_written = 0;
    int curr_read = 0;
    int curr_write = 0;
    while ((________________________________________________________________
                    ____________________________________________________) > 0) {
        bytes_read += curr_read;
    }
    ____________________________________________________________________________;
    // Allocates sizes based on our id (not a real library function)
    get_sizes (id, &args, &arg_bytes, &rets, &ret_bytes);
    bytes_read = 0;
    while ((________________________________________________________________
                    ____________________________________________________) > 0) {
        bytes_read += curr_read;
    }
    // Calls the appropriate server stub based on id (not a real library function)
    call_server_stub (id, args, arg_bytes, rets, ret_bytes)
    while ((________________________________________________________________
                    ____________________________________________________) > 0) {
        bytes_written += curr_write;
    }
    ____________________________________________________________________________;
}
```

4.4 Call Stubs

Finally we want to implement the call stubs, which are function wrappers between each individual function we support and the generic RPC library. For example these are the client stubs for our functions

After the raw data is processed we need to perform the actual computation and return the result to the client. To do this we introduce a stub procedure which unpacks our arguments, calls the procedure to execute on the server, and finally packs the return results to give to the transport layer.

```
// addrs is used for setting up our socket connection
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);
}
```

`call_rpc` is our generic rpc handler that our stubs provide an abstraction around. Notice that we marshal arguments with `htonl()` and unmarshal them with `ntohl()`.

Implement `ith_prime_sstub` and `is_coprime_sstub` which should unmarshal the arguments, call the implementation functions, and finally marshal the return data.

```
// Both functions take in args in network order and should place
// return value(s) in rets in network order
void ith_prime_sstub (char *args, char *rets) {
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
}
void is_coprime_sstub (char *args, char *rets) {
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
}
```

4.5 Handling failure

What are some ways a remote procedure call can fail? How can we deal with these failures?