**Lecture 6 Notes:**

**Write Ahead Logs:**

Machines can fail or restart anytime. If a program is in the middle of performing a data modification, what will happen when the machine it is running on loses power? When the machine restarts, the program might need to know the last thing it was doing. Based on its atomicity and durability needs, the program might need to decide to redo or undo or finish what it had started. How can the program know what it was doing before the system crash?  
  
**Definition:**  
  
To guarantee durability and data integrity, each modification to the system is first written to an append-only log on the disk. This log is known as Write-Ahead Log (WAL) or transaction log or commit log. Writing to the WAL guarantees that if the machine crashes, the system will be able to recover and reapply the operation if necessary.  
  
**Solution:**  
  
The key idea behind the WAL is that all modifications before they are applied to the system are first written to a log file on the disk. Each log entry should contain enough information to redo or undo the modification. The log can be read on every restart to recover the previous state by replaying all the log entries. Using WAL results in a significantly reduced number of disk writes, because only the log file needs to be flushed to disk to guarantee that a transaction is committed, rather than every data file changed by the transaction.

Each node, in a distributed environment, maintains its own log. WAL is always sequentially appended, which simplifies the handling of the log. Each log entry is given a unique identifier; this identifier helps in implementing certain other operations like **log segmentation**(discussed later) or **log purging**.

**Examples:**

* **Cassandra**: To ensure durability, whenever a node receives a write request, it immediately writes the data to a commit log which is a WAL. Cassandra, before writing data to a MemTable, first writes it to the commit log. This provides durability in the case of an unexpected shutdown. On startup, any mutations in the commit log will be applied to MemTables.
* **Kafka** implements a distributed Commit Log to persistently store all messages it receives.
* **Chubby**: For fault tolerance and in the event of a leader crash, all database transactions are stored in a transaction log which is a WAL.

**Segmented Logs:**  
  
A single log can become difficult to manage. As the file grows, it can also become a performance bottleneck, especially when it is read at the startup. Older logs need to be cleaned up periodically or, in some cases, merged. Doing these operations on a single large file is difficult to implement.

Break down the log into smaller segments for easier management.

**Solution**

A single log file is split into multiple parts, such that the log data is divided into equal-sized log segments. The system can roll the log based on a rolling policy - either a configurable period of time (e.g., every 4 hours) or a configurable maximum size (e.g., every 1GB).

**Examples**

* **Cassandra** uses the segmented log strategy to split its commit log into multiple smaller files instead of a single large file for easier operations. As we know, when a node receives a write operation, it immediately writes the data to a commit log. As the Commit Log grows in size and reaches its threshold in size, a new commit log is created. Hence, over time, several commit logs will exist, each of which is called a segment. Commit log segments reduce the number of seeks needed to write to disk. Commit log segments are truncated when Cassandra has flushed corresponding data to SSTables. A commit log segment can be **archived**, **deleted**, or **recycled** once all its data has been flushed to SSTables.
* **Kafka** uses log segmentation to implement storage for its partitions. As Kafka regularly needs to find messages on disk for purging, a single long file could be a performance bottleneck and error-prone. For easier management and better performance, the partition is split into segments.

## High-Water Mark: Definition

Keep track of the last log entry on the leader, which has been successfully replicated to a quorum of followers. The index of this entry in the log is known as the High-Water Mark index. The leader exposes data only up to the high-water mark index.

## Solution

For each data mutation, the leader first appends it to WAL and then sends it to all the followers. Upon receiving the request, the followers append it to their respective WAL and then send an acknowledgment to the leader. The leader keeps track of the indexes of the entries that have been successfully replicated on each follower. The high-water mark index is the highest index, which has been replicated on the quorum of the followers. The leader can propagate the high-water mark index to all followers as part of the regular Heartbeat message. The leader and followers ensure that the client can read data only up to the high-water mark index. This guarantees that even if the current leader fails and another leader is elected, the client will not see any data inconsistencies.

## Examples

**Kafka**: To deal with non-repeatable reads and ensure data consistency, Kafka brokers keep track of the high-water mark, which is the largest offset that all In-Sync-Replicas (ISRs) of a particular partition share. Consumers can see messages only until the high-water mark.

## HeartBeat: Background

In a distributed environment, work/data is distributed among servers. To efficiently route requests in such a setup, servers need to know what other servers are part of the system. Furthermore, servers should know if other servers are alive and working. In a decentralized system, whenever a request arrives at a server, the server should have enough information to decide which server is responsible for entertaining that request. This makes the timely detection of server failure an important task, which also enables the system to take corrective actions and move the data/work to another healthy server and stop the environment from further deterioration.

## Definition

Each server periodically sends a heartbeat message to a central monitoring server or other servers in the system to show that it is still alive and functioning.

## Solution

Heartbeating is one of the mechanisms for detecting failures in a distributed system. If there is a central server, all servers periodically send a heartbeat message to it. If there is no central server, all servers randomly choose a set of servers and send them a heartbeat message every few seconds. This way, if no heartbeat message is received from a server for a while, the system can suspect that the server might have crashed. If there is no heartbeat within a configured timeout period, the system can conclude that the server is not alive anymore and stop sending requests to it and start working on its replacement.

## Examples

* **GFS**: The leader periodically communicates with each ChunkServer in HeartBeat messages to give instructions and collect state.
* **HDFS**: The NameNode keeps track of DataNodes through a **heartbeat** mechanism. Each DataNode sends periodic heartbeat messages (every few seconds) to the NameNode. If a DataNode dies, then the heartbeats to the NameNode are stopped. The NameNode detects that a DataNode has died if the number of missed heartbeat messages reaches a certain threshold. The NameNode then marks the DataNode as dead and will no longer forward any I/O requests to that DataNode.

**Gossip Protocol:**  
  
In a large distributed environment where we do not have any central node that keeps track of all nodes to know if a node is down or not, how does a node know every other node’s current state? The simplest way to do this is to have every node maintain a heartbeat with every other node. Then, when a node goes down, it will stop sending out heartbeats, and everyone else will find out immediately. But, this means O(N^2)*O*(*N*2) messages get sent every tick (N*N* being the total number of nodes), which is a ridiculously high amount and will consume a lot of network bandwidth, and thus, not feasible in any sizable cluster. So, is there any other option for monitoring the state of the cluster?

**Definition:**  
  
Each node keeps track of state information about other nodes in the cluster and gossip (i.e., share) this information to one other random node every second. This way, eventually, each node gets to know about the state of every other node in the cluster.

**Solution:**

Gossip protocol is a peer-to-peer communication mechanism in which nodes periodically exchange state information about themselves and about other nodes they know about. Each node initiates a gossip round every second to exchange state information about themselves and other nodes with one other random node. This means that any state change will eventually propagate through the system, and all nodes quickly learn about all other nodes in a cluster.

## Examples

**Dynamo & Cassandra** use gossip protocol which allows each node to keep track of state information about the other nodes in the cluster, like which nodes are reachable, what key ranges they are responsible for, etc.

**Phi Accrual Failure Detection:**

**Background:**  
  
In distributed systems, accurately detecting failures is a hard problem to solve, as we cannot say with 100% surety if a system is genuinely down or is just very slow in responding due to heavy load, network congestion, etc. Conventional failure detection mechanisms like Heartbeating outputs a boolean value telling us if the system is alive or not; there is no middle ground. Heartbeating uses a fixed timeout, and if there is no heartbeat from a server, the system, after the timeout assumes that the server has crashed. Here, the **value of the timeout is critical**. If we keep the timeout short, the system will detect failures quickly but with many false positives due to slow machines or faulty network. On the other hand, if we keep the timeout long, the false positives will be reduced, but the system will not perform efficiently for being slow in detecting failures.

**Definition:**  
  
Use adaptive failure detection algorithm as described by Phi Accrual Failure Detector. Accrual means accumulation or the act of accumulating over time. This algorithm uses historical heartbeat information to make the threshold adaptive. Instead of telling if the server is alive or not, a generic Accrual Failure Detector outputs the suspicion level about a server. A higher suspicion level means there are higher chances that the server is down.

**Solution:**  
  
With Phi Accrual Failure Detector, if a node does not respond, its suspicion level is increased and could be declared dead later. As a node’s suspicion level increases, the system can gradually stop sending new requests to it. Phi Accrual Failure Detector makes a distributed system efficient as it takes into account fluctuations in the network environment and other intermittent server issues before declaring a system completely dead.

**Example:**  
  
**Cassandra** uses the Phi Accrual Failure Detector algorithm to determine the state of the nodes in the cluster.

**Split Brain:**  
  
In a distributed environment with a central (or leader) server, if the central server dies, the system must quickly find a substitute, otherwise, the system can quickly deteriorate.

One of the problems is that we cannot truly know if the leader has stopped for good or has experienced an intermittent failure like a stop-the-world GC pause or a temporary network disruption. Nevertheless, the cluster has to move on and pick a new leader. If the original leader had an intermittent failure, we now find ourselves with a so-called **zombie leader**. A zombie leader can be defined as a leader node that had been deemed dead by the system and has since come back online. Another node has taken its place, but the zombie leader might not know that yet. The system now has two active leaders that could be issuing conflicting commands. How can a system detect such a scenario, so that all nodes in the system can ignore requests from the old leader and the old leader itself can detect that it is no longer the leader?

**Definition:**  
  
The common scenario in which a distributed system has two or more active leaders is called split-brain.

Split-brain is solved through the use of **Generation Clock**, which is simply a monotonically increasing number to indicate a server’s generation.

**Solution:**  
  
Every time a new leader is elected, the generation number gets incremented. This means if the old leader had a generation number of ‘1’, the new one will have ‘2’. This generation number is included in every request that is sent from the leader to other nodes. This way, nodes can now easily differentiate the real leader by simply trusting the leader with the highest number. The generation number should be persisted on disk, so that it remains available after a server reboot. One way is to store it with every entry in the Write-ahead Log.

**Examples:**  
  
**Kafka**: To handle Split-brain (where we could have multiple active controller brokers), Kafka uses ‘**Epoch number**,’ which is simply a monotonically increasing number to indicate a server’s generation.

**HDFS**: ZooKeeper is used to ensure that only one NameNode is active at any time. An epoch number is maintained as part of every transaction ID to reflect the NameNode generation.

**Cassandra** uses generation number to distinguish a node’s state before and after a restart. Each node stores a generation number which is incremented every time a node restarts. This generation number is included in gossip messages exchanged between nodes and is used to distinguish the current state of a node from the state before a restart. The generation number remains the same while the node is alive and is incremented each time the node restarts. The node receiving the gossip message can compare the generation number it knows and the generation number in the gossip message. If the generation number in the gossip message is higher, it knows that the node was restarted.

**Fencing:**  
  
**Background:**  
  
In a leader-follower setup, when a leader fails, it is impossible to be sure that the leader has stopped working. For example, a slow network or a network partition can trigger a new leader election, even though the previous leader is still running and thinks it is still the active leader. Now, in this situation, if the system elects a new leader, how do we make sure that the old leader is not running and possibly issuing conflicting commands?

**Definition:**

Put a ‘Fence’ around the previous leader to prevent it from doing any damage or causing corruption.

## Solution

Fencing is the idea of putting a fence around a previously active leader so that it cannot access cluster resources and hence stop serving any read/write request. The following two techniques are used:

* **Resource fencing**: Under this scheme, the system blocks the previously active leader from accessing resources needed to perform essential tasks. For example, revoking its access to the shared storage directory (typically by using a vendor-specific Network File System (NFS) command), or disabling its network port via a remote management command.
* **Node fencing**: Under this scheme, the system stops the previously active leader from accessing all resources. A common way of doing this is to power off or reset the node. This is a very effective method of keeping it from accessing anything at all. This technique is also called STONIT or “Shoot The Other Node In The Head.”

## Examples

**HDFS** uses fencing to stop the previously active NameNode from accessing cluster resources, thereby stopping it from servicing requests.

## CheckSum: Background

In a distributed system, while moving data between components, it is possible that the data fetched from a node may arrive corrupted. This corruption can occur because of faults in a storage device, network, software, etc. How can a distributed system ensure data integrity, so that the client receives an error instead of corrupt data?

## Definition

Calculate a checksum and store it with data.

To calculate a checksum, a cryptographic hash function like MD5, SHA-1, SHA-256, or SHA-512 is used. The hash function takes the input data and produces a string (containing letters and numbers) of fixed length; this string is called the checksum.

## Solution

When a system is storing some data, it computes a checksum of the data, and stores the checksum with the data. When a client retrieves data, it verifies that the data it received from the server matches the checksum stored. If not, then the client can opt to retrieve that data from another replica.

## Examples

**HDFS** and **Chubby** store the checksum of each file with the data.