Cloud Computing (UE18CS352)
Unit 3

Aronya Baksy

March 2021

1 Introduction: Disk Storage Fundamentals

- Disk latency has the following three components:
  1. **Seek Time**: The time needed for the controller to position the disk head to the correct cylinder of the disk
  2. **Rotational Latency**: The time needed for the first sector of the block to position itself under the disk head
  3. **Transfer Time**: Time needed for the disk controller to read/write all the sectors on the disk.
- RAID (Redundant Array of Independent Disks) is a storage virtualization technology that combines multiple physical disks into one or more logical volumes for increased redundancy and faster performance.
- The driving technologies behind RAID are **striping**,**mirroring** and **parity checking**.

1.1 Storage Architectures

- In **Directly Attached Storage** (DAS), the digital storage is directly attached to the network node that is accessing that storage.
- DAS is only accessible from the node to which the storage device is attached physically.
- **Network Attached Storage** (NAS) is a file-level storage device connected to a heterogeneous group of clients.
- A single NAS device containing physical storage devices (these may be arranged in RAID) serves all file requests from any client in the connected network.
- NAS removes the responsibility of file serving from other servers on the network. Data is transferred over Ethernet using TCP/IP protocol.
- **Storage Area Network** (SAN) is a network that provides access to block-level data storage.
- A SAN is built from a combination of servers and storage over a high speed, low latency interconnect that allows direct Fibre Channel connections from the client to the storage volume to provide the fastest possible performance.
- The SAN may also require a separate, private Ethernet network between the server and clients to keep the file request traffic out of the Fibre Channel network for even more performance.
- It allows for simultaneous shared access, but it is more expensive than NAS and SAN.
- Distinct protocols were developed for SANs, such as Fibre Channel, iSCSI, Infiniband.

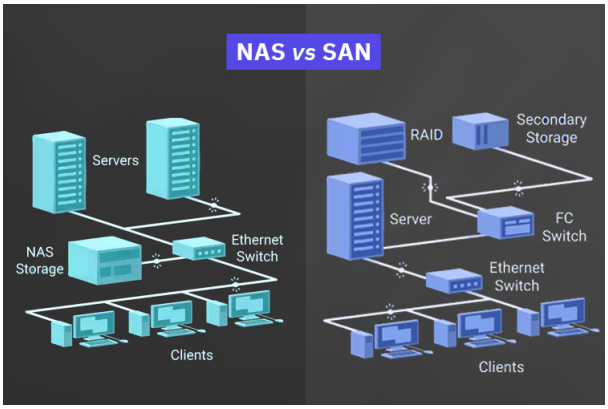

Figure 1: Storage Architectures

1.2 Logical Volume Management (LVM)

- LVM is a file-system virtualization layer
- LVM provides a method of allocating space on mass-storage devices that is more flexible than conventional partitioning schemes to store volumes.
- The components of LVM are:
  1. Extend volumes while a volume is active and has a full file system (shrinking volumes requires unmounting and suitable storage requirements)
  2. Collect multiple pysical drives into a volume group
- LVM consists of the following basic components layered on top of each other:
  - A *physical volume* corresponds to a physical disk that is detected by the OS (labelled often as `sda` or `sdb`) (NOTE: partitions of a single actual disk are detected as separate disks by the OS).
  - A *volume group* groups together one or more physical volumes
  - A *logical volume* is a logical partition of the volume group. Each logical volume runs a file system.
- The `/boot` partition cannot be included in LVM as GRUB (the GNU Bootloader that loads the bootstrap program from the master boot record) cannot read LVM metadata.

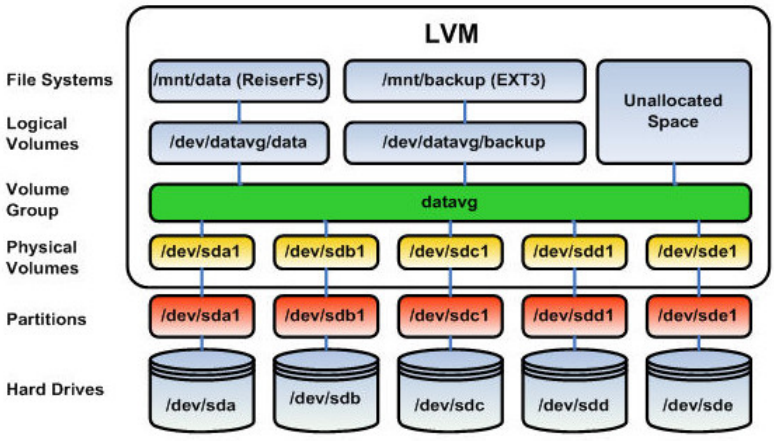

Figure 2: Logical Volume Management

2 Storage Virtualization

- Abstraction of physical storage devices into logical entities presented to the user, hiding the underlying hardware complexity and access functionality (either direct access or network access)
- Advantages of storage virtualization are:
  - Enables higher resource usage by aggregating multiple heterogeneous devices into pools
  - Easy centralized management, provisioning of storage as per application needs (performance and cost).

2.1 File-Level Virtualization

- An abstraction layer exists between client and server.
- This virtualization layer manages files, directories or file systems across multiple servers and allows administrators to present users with a single logical file system
- Normally implemented as a network file system that has
  - Standard protocol for file sharing
  - Multiple file servers enable access to files
- NFS, CIFS, and Web interfaces like HTTP/HTTPS are examples of this.

2.1.1 Distributed File System

- DFS is a type of network file system that is spread across multiple interconnected nodes.
- The objective of DFS is to enable file directory *replication* (for fault tolerance) and *location transparency* (using names to refer to resources rather than their actual location)
- Recently accessed disk blocks can be cached for better performance.
- Metadata managment is important for performance reasons. It can be either **centralized** or **distributed**

2.1.2 DFS with centralized metadata: Lustre

- All metadata operations by clients are directed to a single dedicated metadata server.
- Lock-based synchronization is used in every read or write operation from the clients.
- When workloads involve large files, such systems scale well. But the metadata server can become a SPOF or a performance bottleneck when loads increase.
- **Lustre** is a massively parallel, scalable distributed file system for Linux that uses DFS with centralized metadata.
- It is available under GNU General Public License, and used on many supercomputer grids that run Linux.
- The components of Lustre are:
  1. **Object Storage Server** (OSS), store file data on object storage targets (OSTs). A single OSS can serve 2-8 OSTs. The total capacity of a Lustre FS is the sum of capacities provided by the OSS across all the OST nodes.
  2. **Metadata target** (MDT) stores metadata on one or more metadata servers (MDS)
  3. **Luster clients** access data over a network using a POSIX-compliant interface.
- The file access is done in the following sequence:
  - Client performs a lookup on the MDS for a filename.
  - MDS either returns layout for the existing file, or creates the metadata for a new file.

  - The client passes this layout to a Logical Object Volume (LOV). The LOV maps the layout to objects and their actual locations on different OSTs
  - The client then locks the file range being operated on and executes one or more parallel reads/writes directly to the OSTs

2.1.3 DFS with distributed metadata : Gluster

- Metadata distributed among all the network nodes. Involves greater complexity as metadata has to be managed across multiple nodes
- **Gluster** is an open-source distributed file system with distributed metadata. It is optimized for high performance, and scales up to 1000s of clients and PB of data.
- Gluster mploys a modular architecture with a stackable user-space design.
- It aggregates multiple **storage bricks** on a network (over Infiniband RDMA or TCP/IP interconnects) and delivers as a network file system with a global name space
- The components of Gluster are:
  - **Server** delivers the combined disk space of all the physical storage servers as a single file system
  - **Client** implements highly available, massively parallel access to each storage node along with node failure handling
- A storage brick is a server (containing directly attached storage or connected to a SAN) on which a file system (like ext3 or ext4) is created
- A translator is a layer between a brick and the actual user. It acts as a file system interface and implements one single Gluster functionality
- I/O Scheduling Translators are responsible for load balancing,
- Automatic File Replication (AFR) translator keeps identical copies of a file/directory on all its subvolumes (used for replication)

2.2 Block-Level Virtualization

- Virtualizes multiple physical disks into a single virtual disk
- Data blocks are mapped to one or more physical disks sub-systems.

2.2.1 Host-based BLV

- Uses LVM (section 1.2) to support dynamic resizing of volumes, or combine fragments of unused disk space into a single volume, or create virtual disks (with size larger than physical disk)

2.2.2 Storage Device-level BLV

- Creates Virtual Volumes over the physical storage space of the specific storage subsystem.
- Using RAID techniques, logical units are created that span mulitple disks.
- Host independent and low latency as virtualization is built into the firmware and hardware of the storage device

2.2.3 Network-Level BLV

- Most commonly implemented, scalable form, implemented as part of the interconnect network between storage and hosts (e.g.: Fibre Channel SAN)
- **Switch-based**: the actual virtualization occurs in an intelligent switch in the network, and it works in conjunction with a metadata manager
- **Appliance-based**: I/O is routed through an appliance that manages the virtualization layer
- **In-band appliances** perform all I/O with zero direct interaction between client and storage.
- **Out-of-band appliances** manage only metadata (control paths) while the actual data flows directly between client and storage server (each client having an agent to manage this)

3 Object Storage Technologies

3.1 Amazon Simple Storage Service (S3)

- Highly reliable, available, scalable, fast cloud storage that supports storage and retrieval of large amounts of data using simple web services
- Interaction with S3 is done via the GUI (Amazon Console), the TUI (Amazon CLI) or language specific abstractions. A RESTful API is provided for basic HTTP operations
- Files are called **objects**. The **key** of an object is its identification (directory path + object name). All objects are stored in **buckets**.
- S3 objects are replicated across multiple global zones. Versioning enables further recovery from modification and deletion by accident.
- Security is maintained in S3 using:
  - **Access Control Lists**: Set permissions to allow other users to access an object
  - **Audit Logs**: Once enabled, stores the access log for an bucket. This enables one to identify the AWS account, IP Address, time of access and operations performed by the one who accessed.
- Data Security is maintained in S3 using:
  - **Replication**: across multiple devices, allows for upto 2 replica failures (cheaper option is Reduced Redundancy Storage which survives only 1 replica failure), but consistency across replicas is *not guaranteed*.
  - **Versioning**: If enabled, S3 stores the full history of each object. It allows for changes to be undone, including file deletions.
  - **Regions**: select location of S3 bucket for performance/legal reasons.
- S3 allows for large objects to be uploaded in parts. These parts can be uploaded in parallel for maximum network utilization

3.2 DynamoDB - NoSQL Service

- Cloud-based NoSQL database that is available with AWS. Consists of tables created and defined in advance (with some dynamic elements)
- Overall is schemaless.
- Supports only item-level consistency (similar to row-level consistency in RDBMS). If cross-item consistency is needed then don't use DynamoDB
- Joins are implemented only at the applicaiton side. DynamoDB *does not support joins* between tables.

- Table is collection of items, item is collection of attribute-value pairs. Primary key identifies items uniquely in a table.
- A partition is an allocation of storage for a table, backed by SSDs and automatically replicated across multiple Availability Zones within an AWS Region.
- Types of primary keys in DynamoDB:
  - **Partition Key**: The value of the partition key attribute is passed into a hash function to determine the physical partition on which that item will be stored
  - **Partition + Sort Keys**: All items with the same partition key hash value are stored together in sorted order by sort key value.
- Users can also create secondary keys in addition to primary keys for alternate queries.

3.3 Amazon Relational DB Service (RDS)

- Provides an abstraction of an RDBMS. Offers all majorly used RDBMS such as Amazon Aurora, PostgreSQL, MySQL, Oracle, MS SQL Server
- AWS performs all admin tasks related to maintenance, as well as periodic backups of the DB state and the ability to take snapshots.
- RDS provides encryption at rest and in transit, as well as APIs for applications.

4 Partitioning

- Breaking down large DBs into smaller units that are stored on different machines. Each row belongs to exactly one partition
- Supports operations that touch mulitple partitions at the same time.
- Motivation is **scalability** in terms of load balancing and query throughput, as well as fault tolerance (when combined with replication)
- Small queries can be independently processed by one partition. Large queries can be parallelized between multiple partitions.
- When some partitions have more data than others, they are said to be **skewed**. A partition with disproportionately high load is called a **hot spot**

4.1 Partitioning Strategies

4.1.1 Randomized Partitioning

- Distribute the data quite evenly across the nodes
- Disadvantage: When trying to read a particular item, no way of knowing which node it is on, so all nodes need to be queried in parallel.

4.1.2 Partitioning by Key Range

- Assign range of key values to a given partition. If partition boundaries are known then determining which partition a given key is in is very simple
- Ranges may not be equal width, as data distribution is not uniform
- Each partition can have keys in sorted order
- Disadvantage: certain access patterns can lead to hot spots (e.g.: storing sensor data, if the key is timestamp then all writes go to one single partition which the current day's partition)

4.1.3 Partitioning by Hash of Key

- Using a suitable hash function for keys, each partition has a range of hash values assigned to it (rather than a range of keys), and every key whose hash falls within a partition's range will be stored in that partition.
- A good hash function takes skewed data and makes it uniformly distributed
- Simple hash partitioning do not allow efficient range queries. This is solved using composite keys.
- **Consistent hashing** is a way of evenly distributing load across an internet-wide system of servers such as a content delivery network
- It uses randomly chosen partition boundaries to avoid the need for central control or distributed consensus

4.2 Secondary Indexes

- Do not map neatly to partitions, but useful for increasing performance of queries made on a particular key.

4.2.1 Document-based Secondary Indexing

- Also called **local secondary indexing**
- Each partition maintains its own secondary index, covering only the documents in that partition.
- Reading involves reading from each and every partition and separately combining the results. This approach is called **scatter-gather**, and it makes read queries expensive
- Even if the partitions are queried in parallel, scatter/gather is prone to tail latency amplification

4.2.2 Term-based Secondary Indexing

- A single **global secondary index** covers data from all partitions.
- The index is stored on multiple nodes, partitioned by the term (for range scans) directly, or a hash of the term (for load balancing)
- Reads are more efficient as a query is made only to the partition where the term resides
- Writes are less efficient as a write affects multiple partitions of the index. This requires a distributed transaction across all partitions affected by a write
- In practice, updates to global secondary indexes are often asynchronous

4.3 Rebalancing Partitions

- The process of moving load from one node in the cluster to another is called rebalancing.
- Requirements of rebalancing:
  - After rebalancing, loads should be shared fairly between all cluster nodes
  - During rebalancing the system should still accept read/write requests
  - Minimize the amount of data moved around to reduce network and I/O overheads
- The following are rebalancing strategies:

4.3.1 Hash mod n

- hash(key) % n returns a number between 0 and n-1, corresponding to a single partition
- Simple, but drawback is that any change in N leads to rehashing of large number of keys which makes the rebalancing very expensive

4.3.2 Fixed number of partitions

- Move only entire partitions. Assignment of keys to partitions does not change, but only assignment of partitions to nodes changes.
- Create many more partitions than there are nodes and assign several partitions to each node
- If a node is added to the cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again
- So many fixed-partition databases choose not to implement partition split and merge
- Choosing the right number of partitions is difficult if the size of the dataset is variable

4.3.3 Dynamic Partitioning

- Fixed number of partitions can become imbalanced as data is inserted and removed from the database
- In dynamic partitioning, the number of partitions adapts to the total data volume
- In dynamic partitioning, the partitions split if they grow beyond an upper bound. If the partition shrinks below a lower bound, it can be merged with an adjacent partition
- Can be used with both key-range partitioned and hash partitioned data

4.3.4 Proportional to number of nodes

- The size of each partition grows proportionally to the dataset size while the number of nodes remains unchanged, but when number of nodes increase, the partitions become smaller again
- Keeps partition sizes stable
- When a new node joins the cluster, it randomly chooses a fixed number of existing partitions to split, and then takes over half of each of those split partitions.

4.4 Request Routing

- In case of a dataset partitioned among multiple nodes, which node should read/write requests from a client go to? Request routing solves this issue
- Approaches to routing are:
  - Client contacts one node at random. If that node contains the request partition then it serves the client, else it forwards the request to the appropriate node (this requires all nodes to be aware of partition -> node assignments)
  - Client contacts a routing tier, which is aware of all the node assignments. It forwards the request to the appropriate node. The routing tier only acts as a *partition-aware load balancer*
  - Client directly contacts the appropriate node on which the requested partition lies, requiring each client to know about partitioning and assignment to nodes.

4.4.1 ZooKeeper

- A distributed metadata management system for clusters.
- ZooKeeper maintains an authoritative mapping between partititons and nodes, and each node registers itself with the ZooKeeper service.
- Other actors, such as the routing tier or the partitioning-aware client, can subscribe to this information in ZooKeeper
- When partitioning changes or node removal/addition occurs, ZooKeeper notifies the routing tier

5 Replication

- Keeping multiple copies of a single partition on different nodes connected by a network
- Motivation for replication:
  - Reduce latency by reducing distance to user
  - Increase availability by allowing fault tolerance
  - Increase read throughput by allowing more parallel reads (scalable)

5.1 Single Leader Replication

- Among all replicas, elect one leader and keep all other partitions as followers.
- All write requests from clients are directed to the leader, but read requests can be served by leader or followers
- When a leader gets a read request, it first updates its own write log. This write log is transmitted to all the followers, and the followers apply the changes in the same order as the leader.
- In **synchronous replication**, the leader waits for all followers to confirm that they have received a write request, and only then sends success message to the user.
- In **asynchronous replication**, the leader sends the success message to the user without waiting for followers to acknowledge the receipt of the write.
- In **semi-synchronous replication**, the leader waits for exactly one follower to confirm that it has received a write request, and only then sends success message to the user.
- Sychronous replication sacrifices availability for consistency, vice versa for asynchronous

5.1.1 Node Failure

- Follower failure is handled using **catch-up recovery**. Follower stores the edit logs on its disk
- If a failed follower is restarted, then it can ask the leader for all log entries between the time it crashed and the current time. Upon receiving this, the follower replayed these log entries to get the updated data
- In case of leader failure, one of the old followers has to be elected to the position of leader. Clients are to be reconfigured to send their write queries to this new leader
- Leader failover takes place manually (by theactions of a system admin) or automatically. The steps in leader failover are:
  - Identify leader failure
  - Elect a new leader
  - Reconfigure system to use the new leader

5.1.2 Implementation

- **Statement Replication**: The leader logs every write request that it executes and sends that statement log to its followers (fails for non-deterministic functions like `rand()` and `now()`)
- **Write-Ahead Log Shipping**: The leader writes the log (an append-only byte stream) to disk and sends it across the network to its followers. When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.
- **Logical Log Replication**: Uses different log formats for replication and for the storage engine. A logical log (aka the replication log) is a sequence of records describing writes to database tables at the row level
- **Trigger-Based Replication**: A trigger on the leader table logs the change to another table where an external process can read it. The external process applies the replication to another system

5.2 Replication lag

- The delay between a write happening on the leader and the same being reflected on a follower is known as the replication lag.
- Read-After-Write consistency is a guarantee for a *single user*, in that if the same user reads the data at any time interval after reading it, the user will get the updated data.
- Solutions:
  - Read critical data from leader, rest from follower (negates scaling advantage)
  - Prevent queries on any follower that is lagging significantly behind the leader
  - Client remembers the timestamp of their most recent write, and ensure that the node serving that user is updated atleast till that timestamp
  - Monotonic reads: each user read from the same replica always
  - Consistent prefix reads - if a sequence of writes happen in a certain order, then anyone reading those writes should see them appear in the same order

5.3 Multi-Leader Replication

- Allow more scalability in writes by allowing multiple leaders. Each leader simultaneously acts as a follower to the other leaders.
- Conflict Avoidance:
  - Ensure that all writes for a particular record go through the same leader
  - Give each write an unique ID and pick the write with the highest ID (throw the others away)
  - Custom conflict resolution logic in the application code that may be executed on write/reads
- In a single leader config, .
- In a multi leader config, the writes can go to the nearest leader only and replicated asynchronously to all the other leaders (better perceived performance)
- In a single leader config, the failure of a leader means there is downtime involved in failover.
- In a multi-leader config, each datacenter can continue operating independently of the others, and replication catches up when the failed datacenter is back online.
- In a single-leader config, the public internet is used for synchronous updates between leader and follower, hence is sensitive to problems in this network
- A multi-leader config with asynchronous replication tolerates network problems better as a temporary network problems do not prevent writes being processed

5.4 Leaderless Replication

- No single dedicated leader, all replicas of a partition are the same from the client point of view.
- In some implementations, the client sends writes to multiple nodes at the same time
- In others, a single co-ordinator node does this on behalf of the client, but it does not enforce a particular order of writes (like a leader in a single-leader set up does)
- If writes are sent to multiple nodes, but some nodes out of these fail and hence cannot complete the write. If the nodes that failed come back online, then any data on them is now out of date (stale)
- To solve this issue, each data item has a version number associated with it. The client reading from multiple replicas checks the version number of the data and selects the most recent one.
- When the client reads values with different version numbers, the client writes the most recent version of the data to all the nodes with less recent versions. This is called **read repair**

- A background process (rather than the client itself) monitors all data values and their versions across all nodes, and periodically writes the latest value of the data to all the replicas. This is called an **anti-entropy process**.
- Let there be n nodes. Let r nodes be queried for each read, and w nodes confirm for each write. If

$$w + r > n$$

  then an up-to-date copy of the data is guaranteed while reading, as at least one of the r nodes being read from must be up to date.
- Reads and writes that obey the above rule are called **quorum reads and writes**.

5.4.1 Monitoring

- Monitoring in leaderless systems is difficult as writes do not happen in any particular order
- In single-leader systems, the writes are in a fixed order maintained on the edit log of the leader. The out-of-date follower can compare its position (timestamp) with that of the leader and make the necessary changes.

5.4.2 Multi-datacenter Operation

- Leaderless replication is suitable for multi-datacenter operation, since it is designed to tolerate conflicting concurrent writes, network interruptions and latency spikes.
- The number of replicas of a single partition n is across all datacenters. Number of replicas in a single datacenter can be configured
- All writes are sent to all replicas, but only a quorum of nodes within the local datacenter is sufficient for the client to detect a success.
- The higher-latency writes to other datacenters are often configured to happen asynchronously

5.4.3 Detecting concurrent writes

- Several clients writing to the same key concurrently means that conflicts will occur (even if quorum is followed)
- Events may arrive in a different order at different nodes, due to variable network delays and partial failures
- **Last Write Wins**: each replica stores the value with the highest version number only (discarding the rest of the data)
- Given two events A and B, A is said to **happen before** B if B knows about A, or depends on A or builds on A.
- Definition of concurrency is dependent on this happens-before relationship. Server can determine whether two operations are concurrent by looking at the version numbers

5.4.4 Algorithm for detecting concurrent operations

- A client must read a key before writing to it.
- When a client reads a key, the replica sends the values that have not been overwritten, as well as the latest version number
- When a client writes a key, it must include the version number from the prior read, and it must merge together all values that it received in the prior read.
- When the server receives a write with a particular version number, it can overwrite all values with that version number or below, but still maintains the values with higher version numbers.
- The collection of all version numbers for an item across all its replicas is called the *version vector*

- Version vectors are sent from the database replicas to clients when values are read, and need to be sent back to the database when a value is subsequently written
- The version vector allows the database to distinguish between overwrites and concurrent writes, and ensures that it is safe to read from one replica and write to another.

6 Consistency Models

- Most distributed systems only guarantee **eventual consistency**
- In eventual consistency, data read at any point may not be consistent across nodes, but if there are no writes for some unspecified interval then all the nodes can catch up to the consistent state
- This is a weak guarantee, as it does not give any guarantees about actual time of consistency.

6.1 Linearizability

- The illusion that there is only one copy of a data item across a distributed system. (implies that all data must be up to date at all times, no staleness in caching)
- Ensures that applications running on the distributed system do not need to worry about replication.
- Main point of linearizability: After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.
- **Compare-and-Set** is an operation on the database:
  - The CAS operation takes in 3 arguments: a memory location to read from (called X), an old value (v_{old}) and a new value (v_{new})
  - If $X == v_{old}$ then set $X := v_{new}$
  - If $X \neq v_{old}$ then return an error, don't change the value in X
- Test for linearizable behaviour: record the timings of all requests and responses and check whether a valid sequential ordering can be constructed from them.
- In synchronous mode, single leader replication is linearizable.
- Consensus algorithms implement measures to avoid stale replicas, and implement safe linearizable storage. (e.g.: ZooKeeper)
- Multi-leader and leaderless replication are not linearizable (leaderless probably not)

6.2 CAP Theorem

- **Consistency** requires that all reads after a write return the same value, which is the latest value
- **Availability** requires that any node (that is not in a failure state) is ready to process requests.
- **Partition Tolerance** requires that a system is tolerant to any network or node failures by rerouting the communications.
- The CAP Theorem states that a distributed system can satisfy at most two of these 2 constraints at a time
- Consistent and Partition Tolerant systems: If a network outage causes one node to be unavailable, then such a system can still use a majority consensus and deliver consistent results (e.g.: MongoDB, Redis, BigTable)
- Available and Partition Tolerant systems: If a network outage disconnects two nodes, then they can still independently process results but there is no consistency guarantee between the data on the 2 nodes. (e.g.: Cassandra, Riak, CouchDB)
- Consistent and Available systems: Ones that cannot handle any network failures (e.g.: RDBMS such as SQL Server, MySQL)

- The modern CAP goal is to maximize combinations of consistency and availability that make sense for the specific application, while incorporating plans for unavailability and recovery of failed partitions.

6.3 Two Phase Commit

- Every node has a transaction **manager** that:
  - Maintains transaction log for recovery
  - Co-ordinates concurrent transactions at the node
- Every node has a transaction **co-ordinator** that:
  - Starts the execution of a transaction at the site, distributes the sub-transactions to other sites
  - Co-ordinates the termination of the transaction (either a successful commit on all nodes or abort on all nodes)
- A concurrency control system must:
  - be resilient to node/communication link failure
  - allow parallelism for greater throughput
  - Optimize cost and communication delay
  - Place constraints on atomic actions
- Either commit at all sites, or abort at all sites. The 2PC mechanism is designed to implement this.

6.3.1 Phase 1

- Coordinator places the record **Prepare T** on its log. The message is then sent to all the sites.
- Each site that receives the message decides whether to commit the componenet of transaction T or to abort it.
- A site that wants to commit enters the pre-commit stage (in this state the site can no longer abort the transaction)
- The site takes the necessary actions to ensure that its component of T will not be aborted, then writes the log message **Ready T**.
- Once the log is stored on disk at the site, the site sends the **Ready T** message back to the coordinator
- A site that doesn't want to commit sends the message **Don't Commit T** back to the coordinator

6.3.2 Phase 2

- If Coordinator gets **Ready T** from all the sites, it logs the message **Commit T** and sends it to all the sites
- If the coordinator has received **don't commit T** from one or more sites, it logs **Abort T** at its site and then sends **abort T** messages to all sites involved in T
- If a site receives a **commit T** message, it commits the component of T at that site, logging **Commit T** as it does
- If a site receives the message **Abort T**, it aborts T and writes the log record **Abort T**