Recap: last lecture

- Caching
  - We focus on basic eviction algorithms: finite cache, fixed-sized items/rows
  - No one-size-fits all policy works
  - We covered: FIFO, LRU, LFU, and some variants
  - OPT: not an “implementable” policy, but can be used as a benchmark to see how much room is left to improve
  - Trade offs between hit rate, complexity of policy, amount of metadata it needs
- Indexing
  - Finding individual rows / ranges of rows
  - Search key
  - Dense index → pointer to each unique search key
  - Sparse index → pointer to beginning of key range (assumes rows are sorted by search key)
  - Secondary index → pointer to non-sorted attribute
  - Outer/inner index → create hierarchy of indices, outer index is sparse
  - Trade offs in speed of lookup, metadata size, insertion/deletion performance

Recap: continued

- Bloom filters
  - Hash function → h(X) = Y, where Y is randomly distributed across a range of integers
  - Goal of Bloom filter: indicate whether an item exists or not in the database before having to actually fetch it from disk
  - Bloom filter structure: long array of 0's and 1's, k hash functions point to the positions in the array
    - If all '1's in the positions, then item may be in the database
    - If at least one '0', item for sure is not in the database
    - Insert item: mark '1' on all of its positions in the array
  - Try to minimize false positive rate
  - Trade off false positive rate, amount of memory bloom filter takes
- Networking layers: TCP/IP

Today

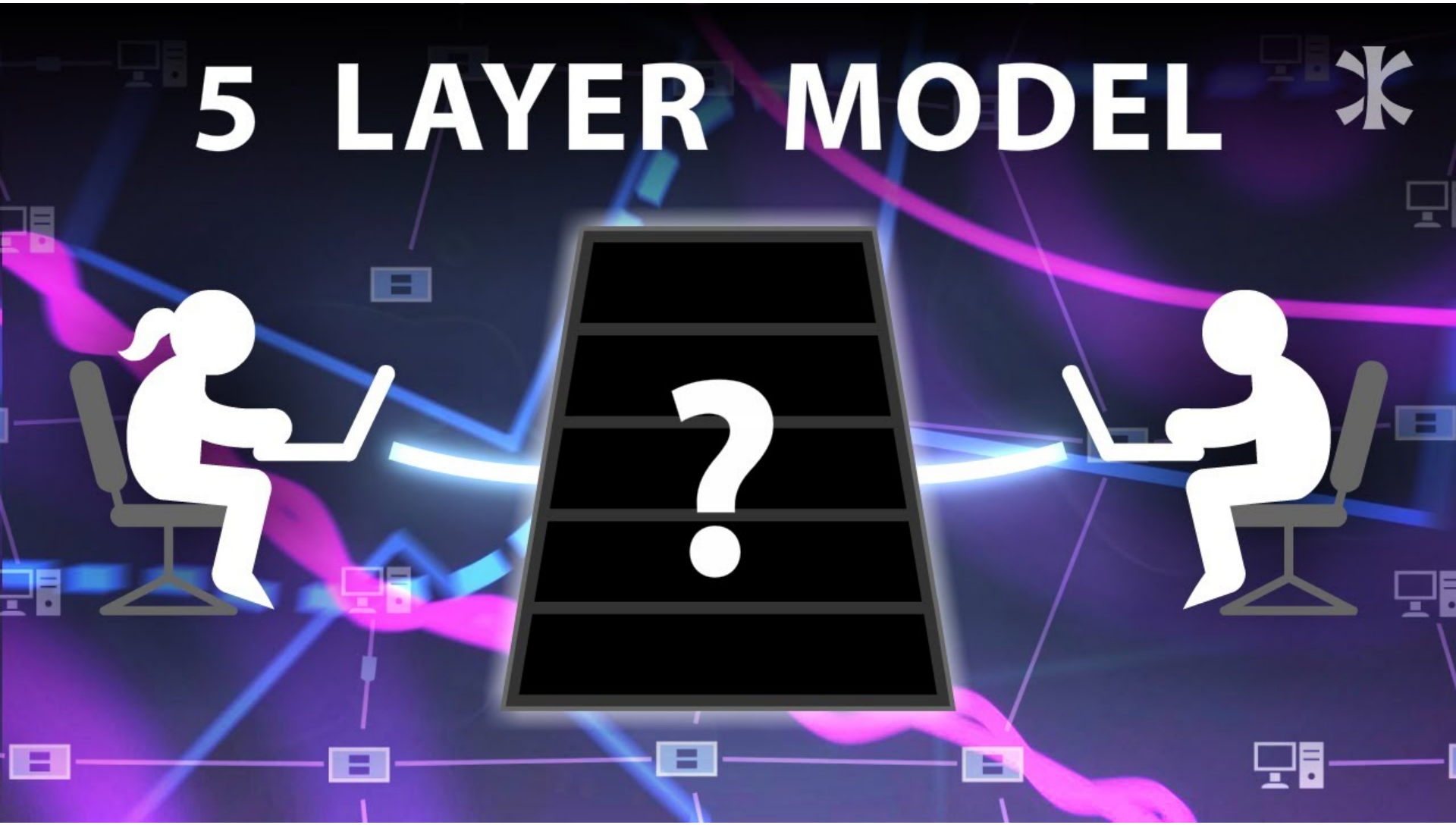
- Networking layers: TCP/IP
- What happens when all my data doesn't fit on a single server?
- Partitioning
  - How to partition data across multiple nodes
    - Round robin, hash, range
    - Trade offs between them
  - Skewed partitions
- Replication
- Distributed file systems
- Distributed databases and distributed transactions
- Two phase commit (2PC)
  - How to ensure ACID across multiple different servers
  - Under different failure scenarios
  - Blocking problem

TCP/IP
5-Layer Model

How Data Actually Travels Across the Internet

1 Physical | 2 Data Link | 3 Network | 4 Transport | 5 Application

Graduate Course in Computer Networking


5 LAYER MODEL
?

The 5-Layer Stack

5 — Application

HTTP · HTTPS · SMTP · FTP · SSH · DNS

4 — Transport

TCP · UDP · QUIC · Port numbers

3 — Network

IPv4 · IPv6 · BGP · Routers

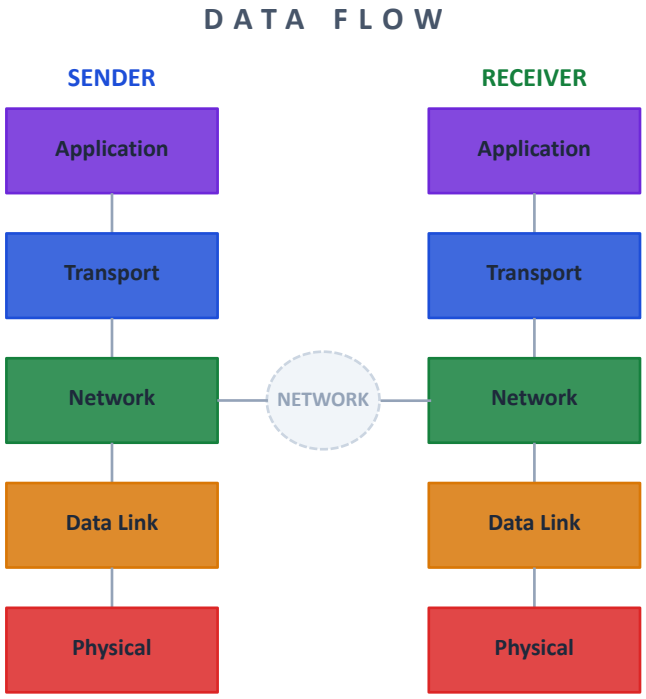
2 — Data Link

Ethernet frames · MAC · ARP · DHCP

1 — Physical

Fiber · Copper · Coax · WiFi · 5G · Satellite · Microwave

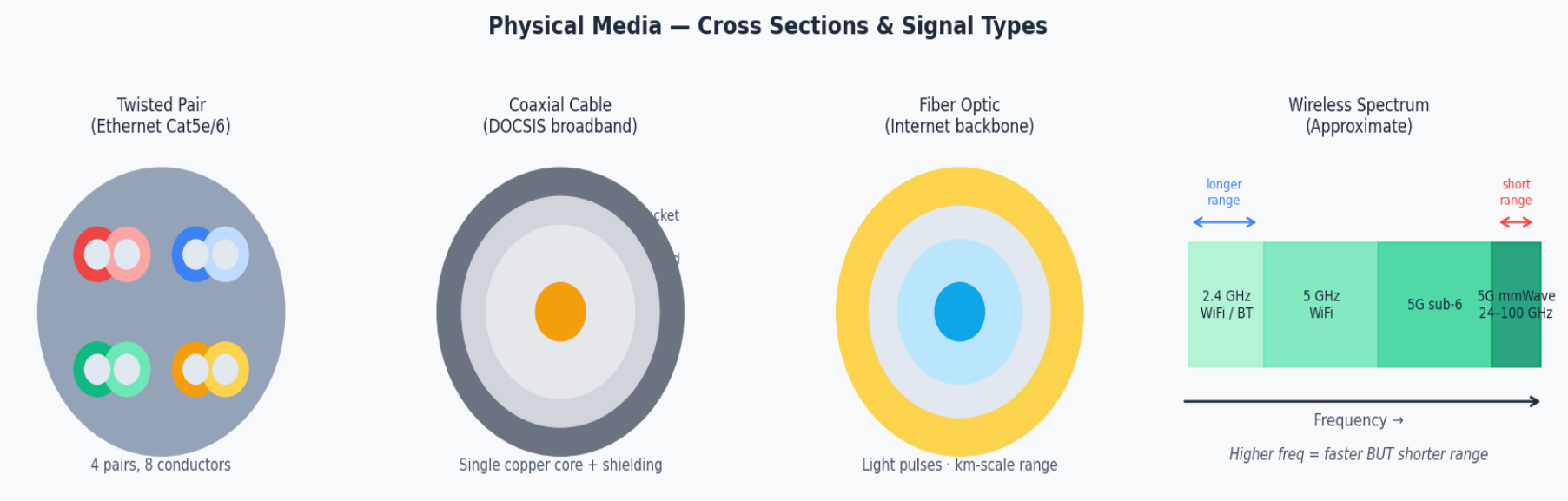
← Encapsulation (sending) · Decapsulation (receiving) →



Layer 1: Physical

L1 Physical

Converts bits to physical signals. The medium determines speed, range, and latency.



Ethernet spans L1 (signaling) + L2 (framing) — the same cable serves both layers

Microwave travels faster than fiber (air vs. glass) — HFT firms use this for trading

GEO satellite: ~600ms latency (physics). Starlink LEO: 20–40ms — viable for apps

KEY CONCEPT Layer 1 is protocol-agnostic — it just moves signals. A broken cable or weak signal kills everything above it regardless of how good your software is.

Layer 2: Data Link

L2 Data Link

Organizes bits into frames · handles local network delivery · MAC addresses identify hardware.

ETHERNET FRAME STRUCTURE

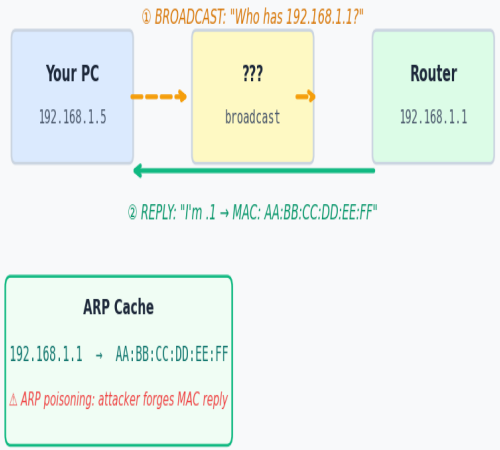
Preamble	Dst MAC	Src MAC	Type	Payload	FCS
8B	6B	6B	2B	46-1500B	4B

MAC Address

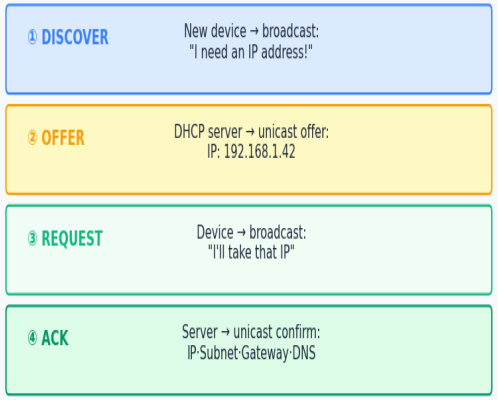
3C:22:FB:4A:9E:01

48-bit hardware identifier · burned in at manufacture
Local only — invisible beyond the router
Switches use MAC tables; routers do not

ARP – Address Resolution Protocol



DHCP – Dynamic Host Configuration



KEY CONCEPT MAC addresses solve local delivery. ARP bridges L2 (MAC) and L3 (IP). DHCP bootstraps new devices automatically. Both are critical infrastructure — and both have well-known attacks.

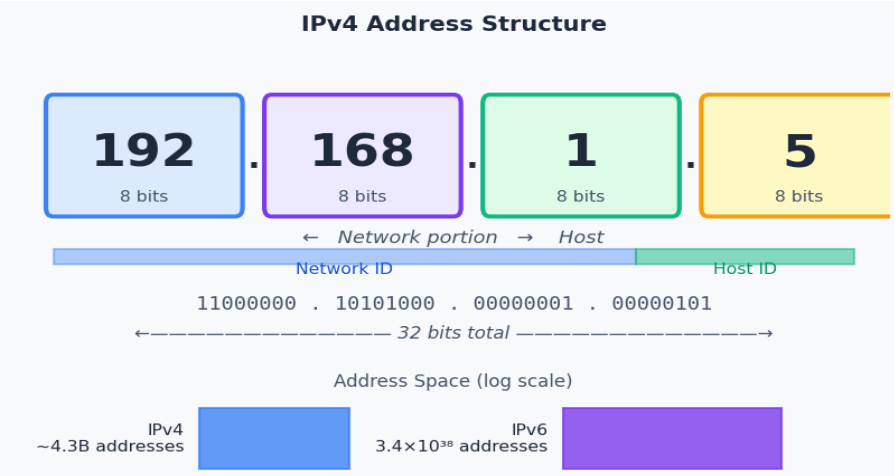
Layer 3: Network

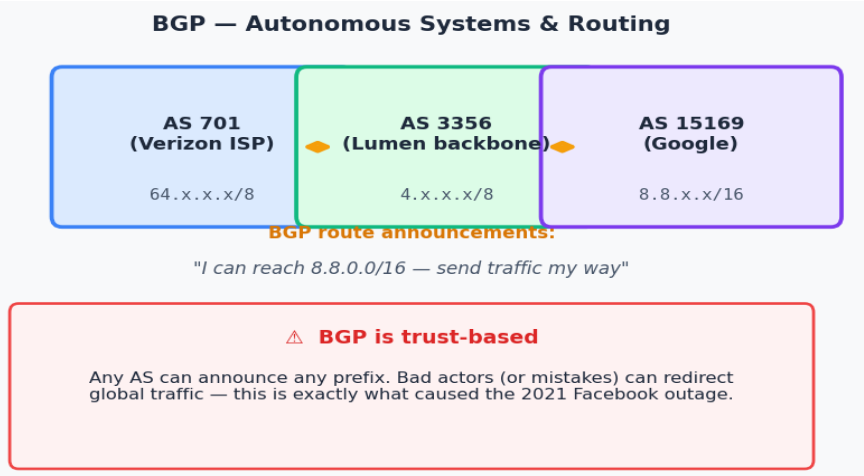
L3 Network

Routes packets across multiple networks using IP addresses. Makes the global internet possible.

IPv4 → 32-bit · ~4.3B addresses · Exhausted (NAT extends it)
IPv6 → 128-bit · 340 undecillion · Eliminates NAT · Dominant on mobile

```
Run  traceroute google.com  in a terminal to see every router hop in real time.
```





KEY CONCEPT IP addresses are logical and hierarchical — they encode location. No router has a complete map: each makes a local best-hop decision. BGP's trust model is its greatest vulnerability.

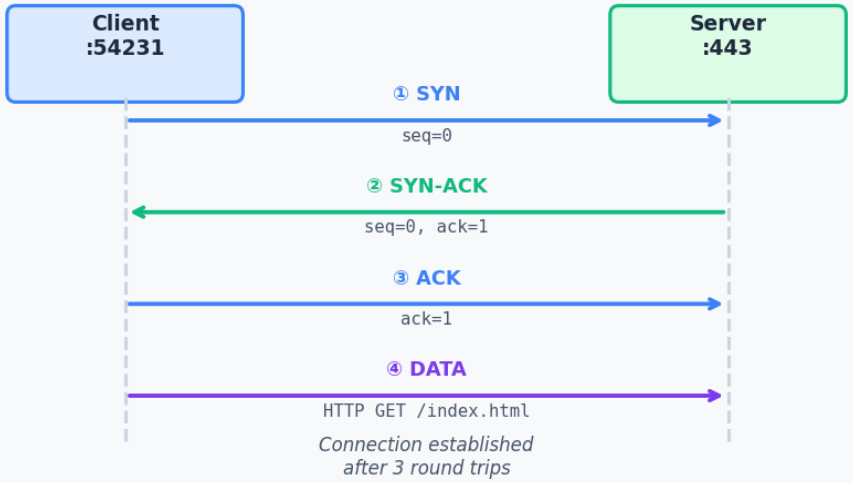
Layer 4: Transport

L4 Transport

End-to-end delivery between applications. Ports multiplex streams. TCP vs UDP is the central reliability/speed tradeoff.

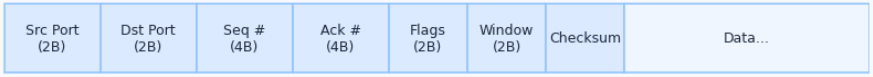
QUIC (RFC 9000 / HTTP/3) Runs over UDP but reimplements reliability. Integrates TLS 1.3 (0-RTT vs TCP+TLS's 3+ round trips). No head-of-line blocking. Survives WiFi→5G handoff without dropping connections. Used by Google, YouTube, Meta for a major fraction of traffic.

TCP — 3-Way Handshake



TCP vs UDP — Header Comparison

TCP Header (20-60 bytes)



UDP Header (8 bytes only!)

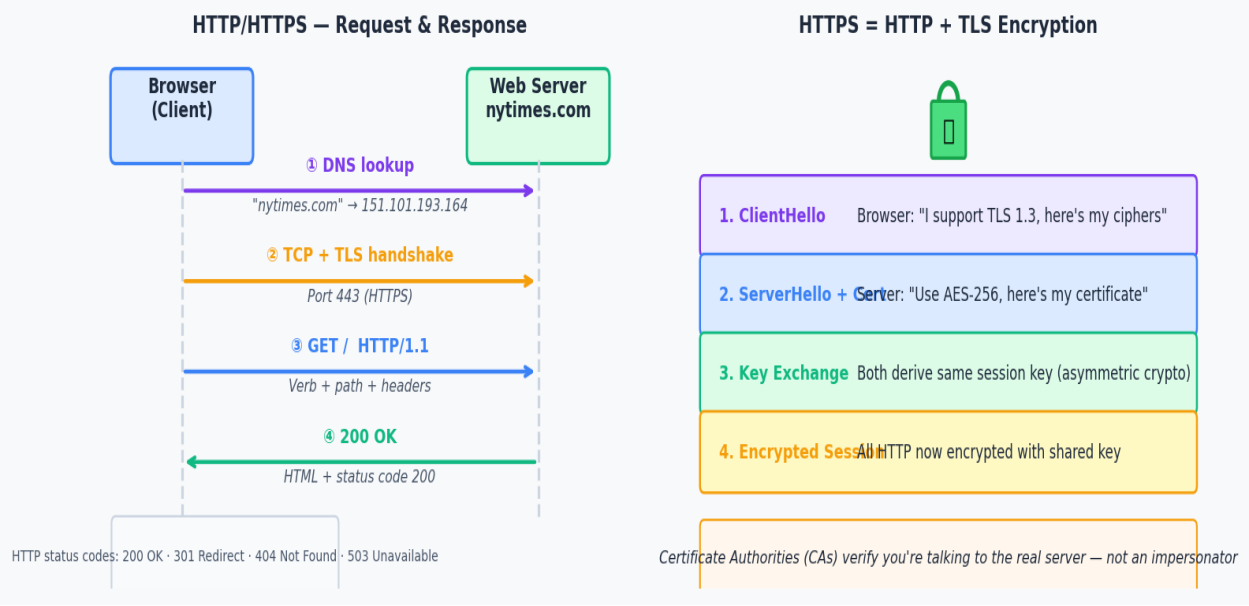
Feature	TCP	UDP
Connection	Required (handshake)	None
Reliability	Guaranteed	Best-effort
Ordering	Preserved	Not guaranteed
Speed	Slower	Faster
Use case	HTTP,SSH,SMTP,FTP	DNS,Zoom,games

KEY CONCEPT TCP trades latency for reliability. UDP trades reliability for speed. QUIC is the modern attempt to get TCP-level guarantees with UDP-level latency.

Layer 5: Application

L5 Application

Where developers work. Protocols define the language two applications speak over a network.



SMTP — Port 25 / 587

Mail transfer between servers. Plaintext by default (1980s!). STARTTLS adds encryption. Weak auth = endemic spam.

SSH — Port 22

Encrypted remote terminal. Replaced Telnet. Key-based auth. SSH tunnels forward arbitrary TCP over encrypted channel.

FTP/SFTP — Port 21 / 22

FTP (1971): plaintext incl. credentials. SFTP (over SSH): encrypted. FTP survives in legacy industrial systems.

DNS — Port 53 UDP

Hostname → IP. Hierarchical: root→TLD→authoritative. TTL caching. DoH/DoT encrypt queries. Critical infrastructure.

KEY CONCEPT Application protocols define message format, verbs, and session rules. Many were designed in the 1980s without security assumptions — which is why plaintext protocols and BGP hijacking remain live threats.

Summary: The 5-Layer Model

Every network problem has a layer. Identifying it is half the diagnosis.

5	**Application** *Software communicates*	`HTTP · HTTPS · SMTP · FTP · SSH · DNS · DoH`
4	**Transport** *App-to-app, reliability/speed*	`TCP · UDP · QUIC · Port numbers`
3	**Network** *Global routing, IP addressing*	`IPv4 · IPv6 · BGP · Routers`
2	**Data Link** *Local delivery, MAC addressing*	`Ethernet · 802.11 MAC · ARP · DHCP · Switches`
1	**Physical** *Bits → signals*	`Fiber · Copper · Coax · WiFi · 5G · Microwave · Satellite`

Mnemonic (bottom → top): **P**eople **D**on't **N**eed **T**o **A**sk (Physical → Data Link → Network → Transport → Application)

KEY CONCEPT The TCP/IP model is a mental framework for diagnosing failures: is this a physical issue, a routing issue, or an application protocol issue? Layer-based thinking is how professionals isolate problems fast.

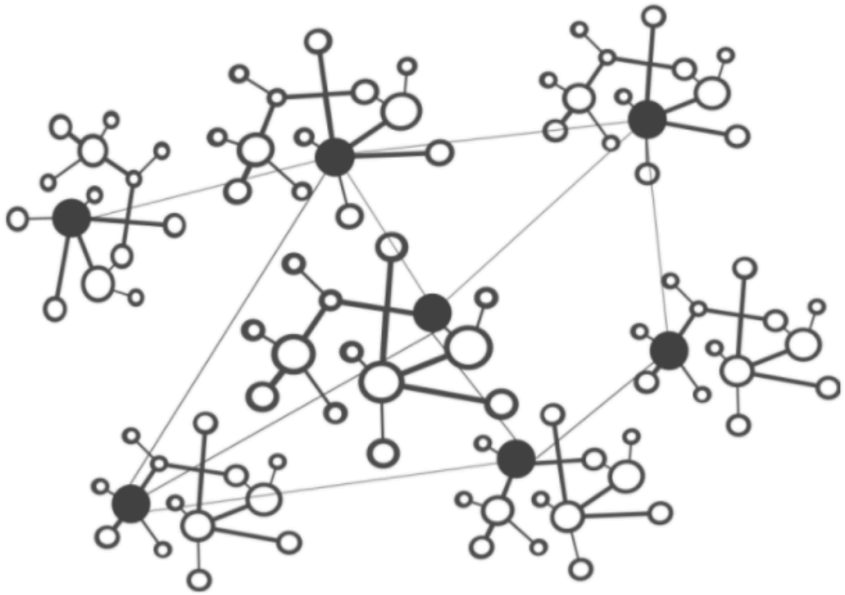
Understanding how Facebook disappeared from the Internet

2021-10-04

 Celso Martinho  Tom Strickx

5 min read

This post is also available in 简体中文, Français, Deutsch, Italiano, 日本語, 한국어, Português, Español, Русский and 繁體中文.



https://blog.cloudflare.com/october-2021-facebook-outage/

The Internet - A Network of Networks

"*Facebook can't be down, can it?*", we thought, for a second.

Storage Parallelism

- Reduce the time required to retrieve data from disk by partitioning the relations on *multiple disks*, on *multiple **nodes*** (computers)
  - Our description focuses on parallelism across nodes
  - Same techniques can be used across disks on a node
- **Horizontal partitioning** – tuples of a relation are divided among many nodes, e.g., r1(A,B,C,D) is placed on node 1, r2(A,B,C,D) on node 2
- **Vertical partitioning** – columns are divided among nodes, e.g., table(A,B,C,D) is split into table1(A,B) on node 1, table2(A,C,D) on node 2
- We will assume we are using horizontal partitioning by default in the class

Storage Parallelism

- Partitioning techniques (number of nodes = n):

  Round-robin:

  Send the i^{th} tuple inserted in the relation to node i mod n

  Hash partitioning:

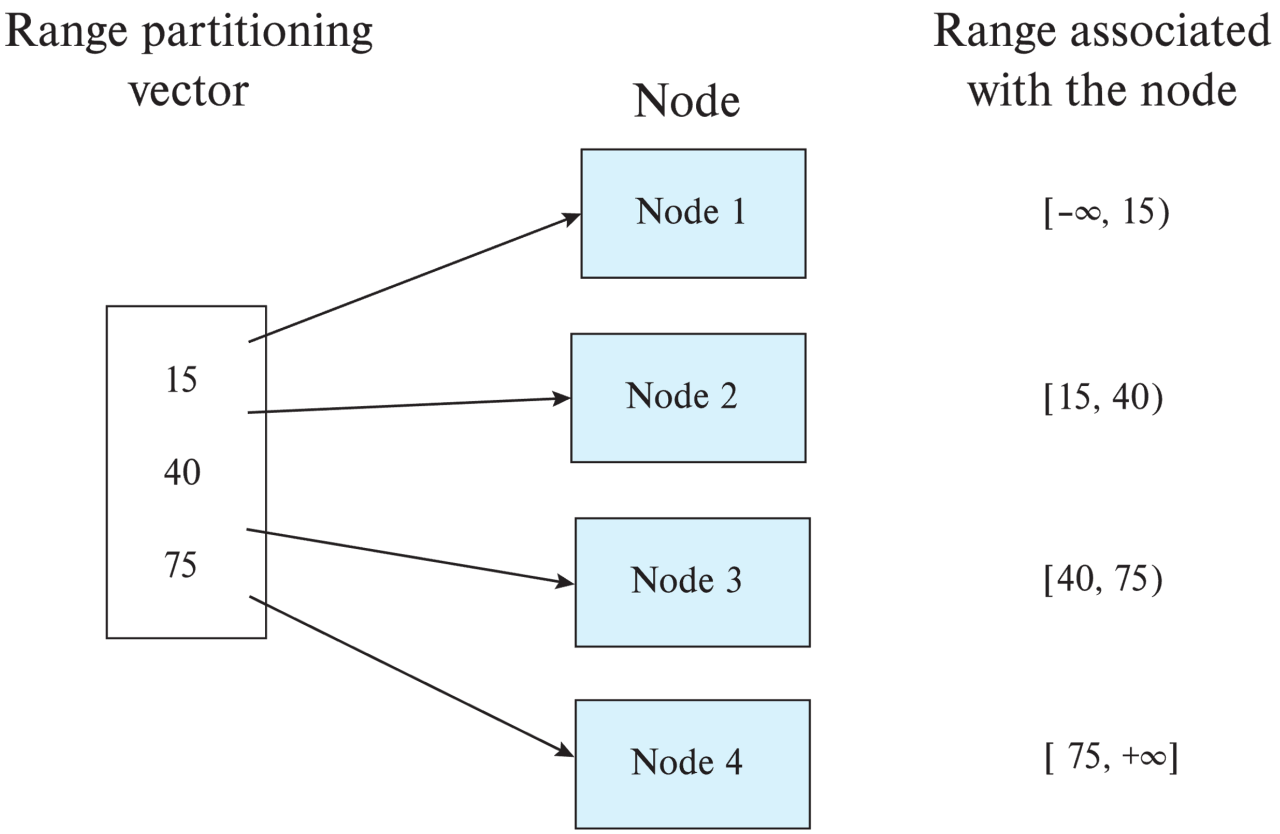
  – Choose one or more attributes as the partitioning attributes.

  – Choose hash function h with range $0 \ldots n - 1$

    - Remember what a hash function is?

  – Let i denote result of hash function h applied to the partitioning attribute value of a tuple. Send tuple to node i

Range Partitioning



Storage Parallelism (Cont.)

Partitioning techniques (cont.):

- **Range partitioning**:
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector $[v_0, v_1, ..., v_{n-2}]$ is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \le v_{i+1}$ go to node $i + 1$. Tuples with $v < v_0$ go to node 0 and tuples with $v \ge v_{n-2}$ go to node n-1.

  E.g., with a partitioning vector [5,11]
  - a tuple with partitioning attribute value of 2 will go to node 0,
  - a tuple with value 8 will go to node 1
  - a tuple with value 20 will go to node 2

Comparison of Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
  1. Scanning the entire relation.
  2. Locating a tuple associatively – **point queries**.
     - E.g., $r.A = 25$.
  3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
     - E.g., $10 \leq r.A < 25$.
- Reminder: we want the accesses to be balanced across the different nodes/disks
- Do above evaluation for each of
  - Round robin
  - Hash partitioning
  - Range partitioning

Comparison of Partitioning Techniques

Round robin:

- Best suited for sequential scan of entire relation and point lookups
  - Point lookups require querying all nodes since placement is based on insertion order, not key values
  - All nodes have almost an equal number of tuples; retrieval work is thus well balanced between nodes
- Another advantage: very simple to partition, little state (insertion counter)
- Disadvantage: all queries involving multiple tuples must be processed at all nodes, not good for range queries

Hash partitioning:

- Similar to round robin
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between nodes
  - Slightly less evenly distributed than round robin (uniform random is not the same as round robin)
- Advantages and disadvantages similar to round robin except
  - Point lookups are trivial: you can immediately find the node you are looking for
  - No insertion counter needed

Comparison of Partitioning Techniques

Range partitioning:

- Provides data clustering by partitioning attribute value
  - Good for limited-size range queries
  - Also good for transactions, assuming transactions access similar key range (we'll talk about this later) → this makes it a popular choice for ACID distributed databases
- For range queries on partitioning attribute, one to a few nodes may need to be accessed
  - Remaining nodes are available for other queries
  - Good if result tuples are from one to a few blocks.
  - But if many blocks are to be fetched, they are still fetched from one to a few nodes, and potential parallelism in disk access is wasted
    - Example of **execution skew**
- Disadvantages:
  - Requires a centralized service (also called database proxy, acts like a registry or index) to figure out which database stores which range
  - Skew is more likely to occur

Handling Small Relations

- Partitioning not useful for small relations which fit into a single disk block or a small number of disk blocks
  - Instead, assign the relation to a single node, or
  - Replicate relation at all nodes
- For medium sized relations, choose how many nodes to partition across based on size of relation
- Large relations typically partitioned across all available nodes

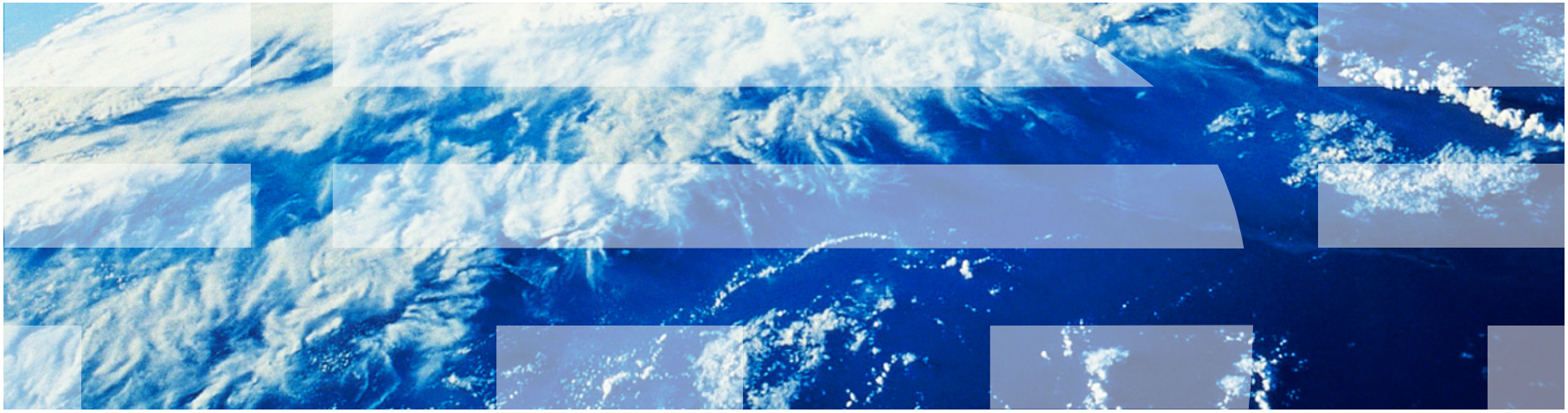
Types of Skew

- **Data-distribution skew:** some nodes have many tuples, while others may have fewer tuples. Could occur due to
  - **Attribute-value skew**.
    - Some partitioning-attribute values appear in many tuples (e.g., “Price”)
    - All the tuples with the same value for the partitioning attribute end up in the same partition
    - Can occur with range-partitioning and hash-partitioning
  - **Partition skew**.
    - Imbalance, even without attribute –value skew
    - Badly chosen range-partition vector may assign too many tuples to some partitions and too few to others
    - Less likely with hash-partitioning
- Data-distribution skew can be avoided with range-partitioning by creating balanced range-partitioning vectors
  - May need to be balanced periodically (can be expensive)
- **Execution skew:** access pattern may be skewed, some tuples may be more “popular”
  - E.g., CUIDs of active students are accessed more frequently by queries than CUIDs of students who’ve graduated
  - This can happen across all three partitioning methods

Routing of Queries

- For range partitioning, partition table typically stored at a **master** node
- Two alternative designs:
  1. Queries are sent first to **master**, which forwards them to appropriate node
     - Disadvantage: need to communicate with master for each query
  2. Master tells **client** (the node asking the query) which nodes stores which key range → clients directly communicate with data nodes
     - Advantage: do not need to talk to master for each query
     - Disadvantages: what happens if a node dies and client has old information? Scalability
  - Examples of storage systems that use a master node: Hadoop File System, Google File System, BigTable (precursor to BigQuery), HBase (open source version of BigTable), …
- **Consistent hashing** is an alternative fully-distributed scheme, without a master, uses a form of hash partitioning
  - Without any master nodes, works in a completely peer-to-peer fashion
  - Advantage: no single point of failure (master), scalability
  - Disadvantage: typically requires more communication/coordination among nodes
  - Examples of systems that use consistent hashing: Cassandra, DynamoDB

Replication



Replication

- Goal: **availability** despite failures
- Data replicated at 2, often 3 nodes
  - Why 3?
- Unit of replication typically a partition (tablet) → may be smaller than a full table
- Requests for data at failed node automatically routed to a replica
- Partition table with each tablet replicated at two nodes

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0,Node1
2013-01-01	Tablet1	Node0,Node2
2014-01-01	Tablet2	Node2,Node0
2015-01-01	Tablet3	Node2,Node1
2016-01-01	Tablet4	Node0,Node1
2017-01-01	Tablet5	Node1,Node0
2018-01-01	Tablet6	Node1,Node2
MaxDate	Tablet7	Node1,Node2

Basics: Data Replication

- Location of replicas
  - **Replication within a data center**
    - Handles machine failures
    - Reduces latency if copy available locally on a machine
    - Replication within/across racks
  - **Replication across data centers**
    - Handles data center failures (power, fire, earthquake, ..), and network partitioning of an entire data center
    - Provides lower latency for end users if copy is available on nearby data center

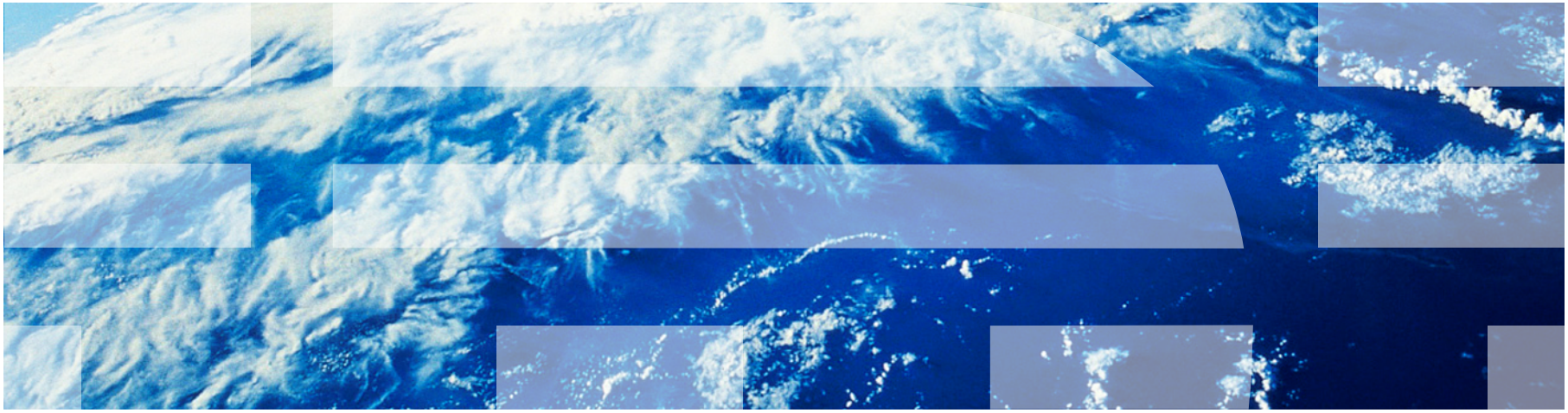
Updates and Consistency of Replicas

- Replicas must be kept consistent on update
  - Despite failures resulting in different replicas having different values (temporarily), reads must get the latest value
  - Special concurrency control and atomic commit mechanisms to ensure consistency
- **Master replica** (**primary copy**) scheme
  - All updates are sent to master, and then replicated to other nodes
  - Reads are performed at master
  - But what if master fails? Who takes over? How do other nodes know who is the new master?

Protocols to Update Replicas

- *Two-phase commit*
  - Coming up!
  - Assumes all replicas are available
- *Consensus protocols*
  - Protocol followed by a set of replicas to agree on what updates to perform in what order
  - **Can work even without a designated master**

Distributed File Systems

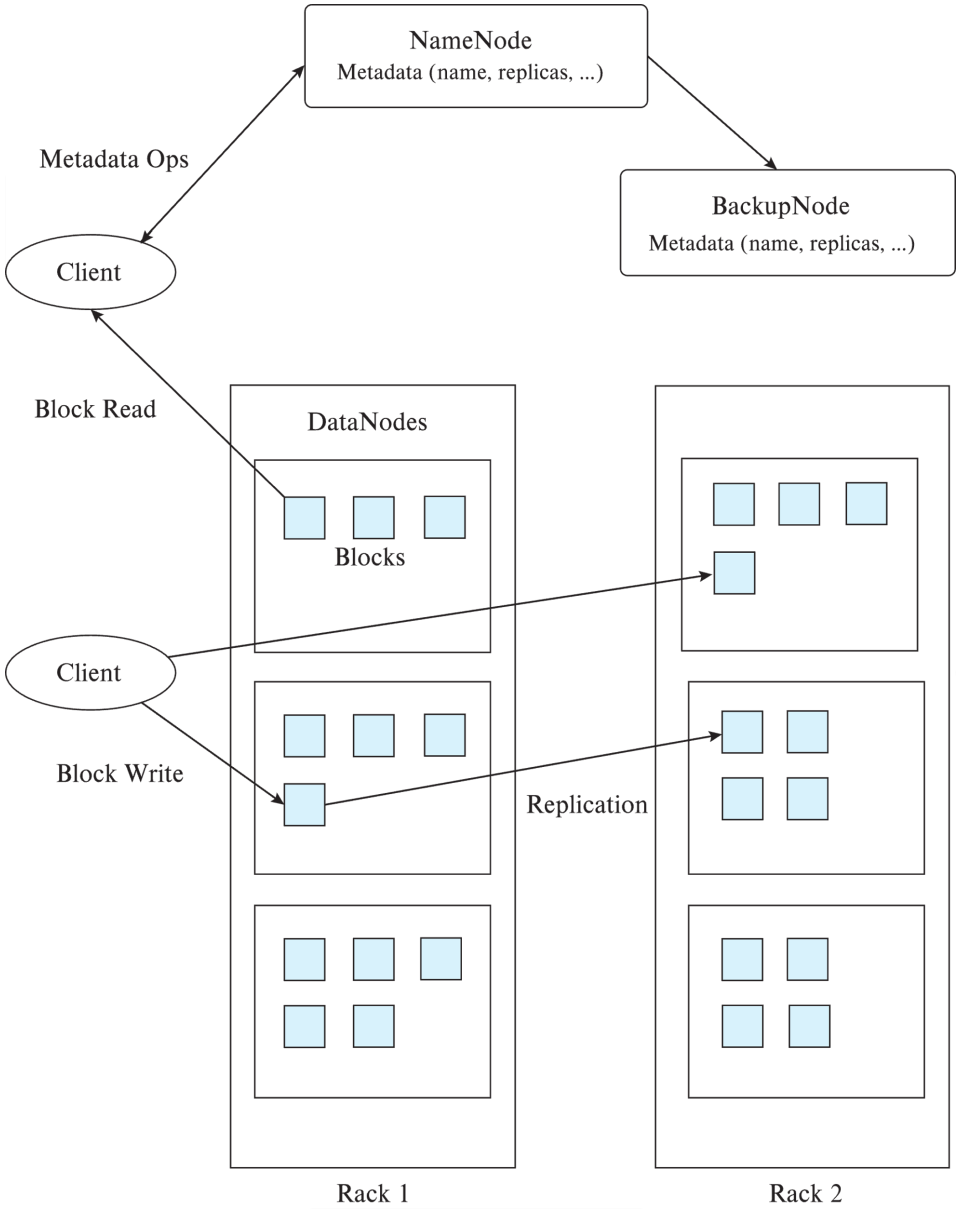


Distributed File Systems

- **Hadoop File System (HDFS)**
- Google File System (GFS)
- And older ones like CODA
- And more recent ones such as Google Colossus
- Basic architecture:
  - Master: responsible for metadata
  - Chunk servers: responsible for reading and writing large chunks of data
  - Chunks replicated on 3 machines, master responsible for managing replicas
  - Replication in GFS/HDFS is within a single data center

Hadoop File System (HDFS)

- Client: sends filename to NameNode
- NameNode (the master node)
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
  - Returns list of BlockIDs along with locations of their replicas
- DataNode:
  - Maps a Block ID to a physical location on disk
  - Sends data back to client



Hadoop Distributed File System

Hadoop Distributed File System (HDFS)

- Modeled after Google File System (GFS)
- Single Namespace (e.g., single directory structure) for entire cluster
- Data model
  - Write-once-read-many access model
  - Client can only append to existing files
    - Not ACID!
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple (e.g., 3) DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode
    - NameNode is not on the critical path of reads and writes

Limitations of HDFS

- Central master becomes bottleneck
  - Keep directory information in memory to avoid expensive storage reads/writes
  - Memory size limits number of files
  - What happens if it fails?
- File system directory overheads per file
  - Not appropriate for storing very large number of objects
- Does not support in-place updates
  - File systems cache blocks locally
  - Ideal for write-once and append only data
  - Can be used as underlying storage for a data storage system
    - E.g., **BigQuery/BigTable** uses GFS/Colossus underneath, **Hbase/Spark** uses HDFS underneath

Distributed File Systems vs. Databases

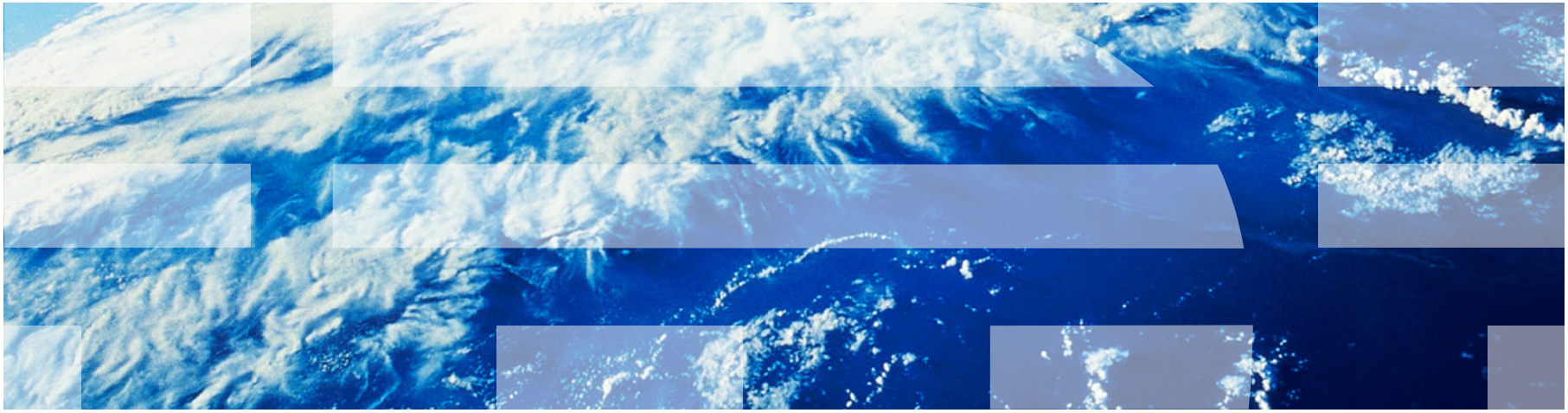
Distributed data storage implementations:

- May have limited support for relational model (no schema, or flexible schema)
- But usually do provide flexible schema and other features
  - Objects e.g. using Parquet, ORC, JSON
  - Multiple versions of data items
- Often provide no support or limited support for transactions
  - But some do!
- Provide only lowest layer (similar to the file system layer)
- Often have KV store on top of them, and relational DB on top of the KV store

Geographically Distributed Storage

- Many storage systems today support geographical distribution of storage
  - Motivations: Fault tolerance, latency (close to user), governmental regulations
- Latency of replication across geographically distributed data centers much higher than within data center
  - Some key-value stores support **synchronous replication**
    - Must wait for replicas to be updated before committing an update
  - Others support **asynchronous replication**
    - update is committed in one data center, but sent subsequently (in a fault-tolerant way) to remote data centers
    - Must deal with small risk of data loss if data center fails.

Distributed Databases and Transactions



Approach 1: Shared-nothing

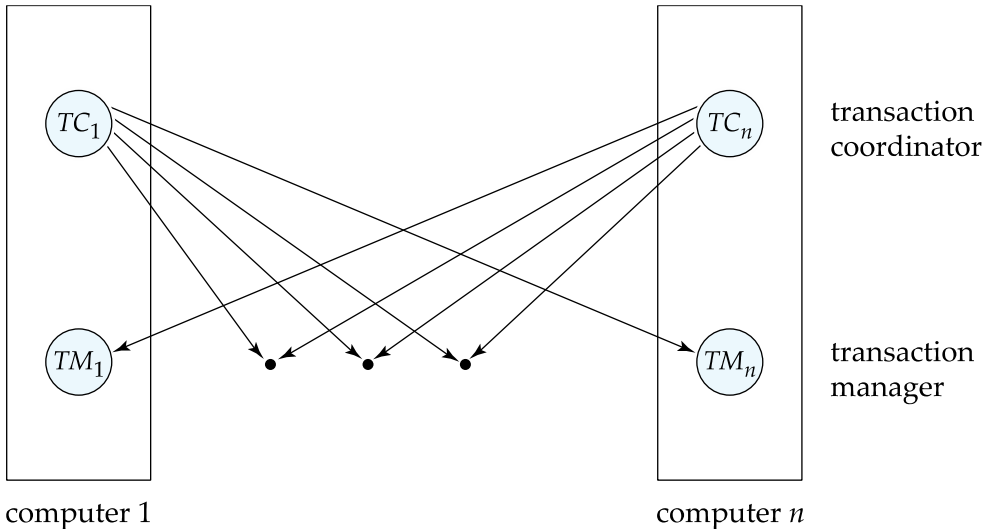
- Divide data amongst many single-server databases (MySQL/MyRocks/ PostgreSQL)
- Manage parallel access in the application
  - Partition tables map keys to nodes
  - Application decides where to route storage or lookup requests
- Scales well for both reads and writes: used widely in data center settings
- Limitations
  - Not transparent
    - application needs to be shard-aware
    - AND application needs to deal with replication
  - (Not a true parallel database, since parallel queries and transactions spanning nodes are not supported)

Approach 2: Distributed Transactions

- **Local transactions**
  - Access/update data at only one database
- **Global transactions**
  - Access/update data at more than one database
- Key issue: how to ensure ACID properties for transactions in a system with global transactions spanning multiple database

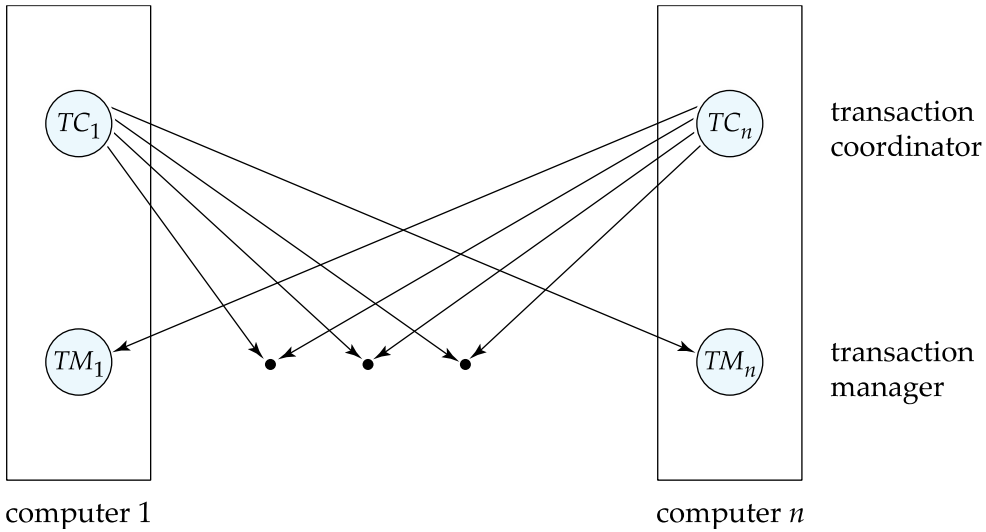
Distributed Transactions

- Transaction may access data at several database servers
  - Each server has a local **transaction manager**
  - Each server has a **transaction coordinator**
    - Global transactions can be submitted to **any** transaction coordinator



Distributed Transactions

- Each transaction coordinator is responsible for:
  - Starting the execution of transactions that originate at the server
  - Distributing sub-transactions at appropriate servers for execution
  - Coordinating the termination of each transaction that originates at the server
    - Key idea: **transaction must be committed at all servers or aborted at all servers**
- Each local transaction manager responsible for:
  - Maintaining a log for recovery purposes for the node it belongs to
  - Coordinating the execution and commit/abort of the transactions executing at that server.



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a server
  - Loss of messages
    - Handled by networking protocols such as TCP/IP
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them

Commit Protocols

- Commit protocols are used to ensure atomicity across servers
  - A transaction which executes at multiple servers must either be committed at all the servers, or aborted at all the servers.
    - Cannot have transaction committed at one server and aborted at another
    - Why?
- **The *two-phase commit* (2PC) protocol is widely used**
- *Consensus protocols* solve a more general problem, but can be used for atomic commit
- We assume **fail-stop** model – failed servers simply stop working, and do not cause any other harm, such as sending incorrect messages to other servers (they do not become "malicious")
  - This is the assumption in many distributed systems, but not all
  - For example, blockchains assume that each node can be malicious (**Byzantine failure model**)
  - Generally a reasonable assumption when all the nodes reside under the same organization, with no conflicting interests

Two Phase Commit Protocol (2PC)

- Execution of the protocol is initiated by any coordinator
- The protocol involves all the servers that store data the transaction requires for its execution
- Protocol has two phases
- Writes records in the WAL at each one of the servers to track the status of the distributed transaction
- Let T be a transaction initiated at server S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision -- Prepare Phase

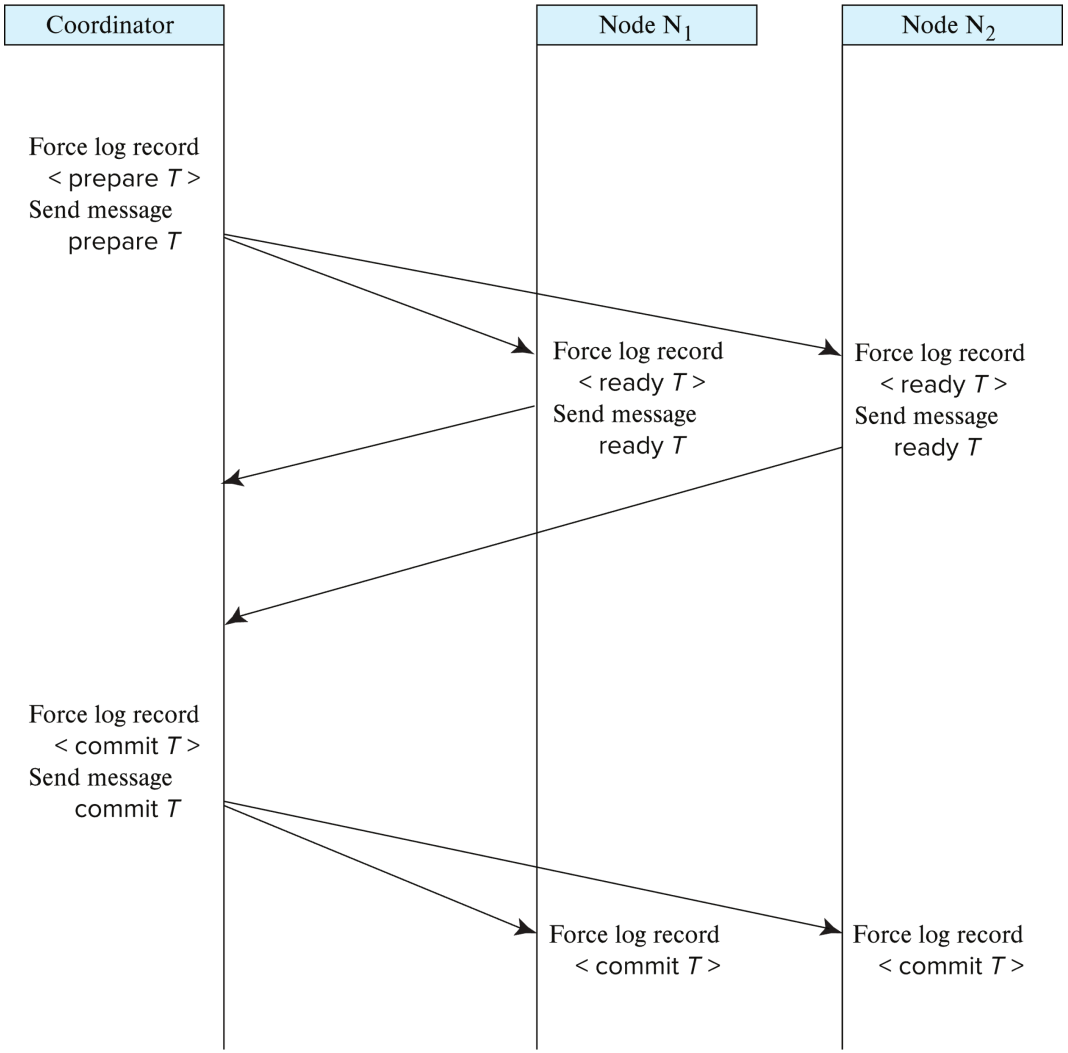
- Coordinator asks all participants to *prepare* to commit transaction T_i
  - C_i adds the records <**prepare** *T*> to the log and flushes log to persistent storage
  - sends **prepare** *T* messages to all servers where *T* needs to be executed
- Upon receiving message, transaction manager at server determines if it can commit the transaction
  - E.g., it tries to acquire locks using conservative 2PL
  - If not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - If the transaction can be committed, then:
    - add the record <**ready** *T*> to the log
    - flush *all records* for *T* to stable storage (i.e., execute the transaction locally), without committing
    - send **ready** *T* message to C_i

    Transaction is now in ready state at the server

Phase 2: Recording the Decision

- *T* can be committed if C_i received a **ready** *T* message from all the participating servers: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <**abort** *T*>, to the log and flushes record onto stable storage
  - Once the record is in stable it is either committed or aborted (similar to the WAL protocol)
  - If it is committed, it can return a commit message to the client
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally
  - If the transaction was committed, they commit locally (flush a commit record into their WAL)
  - If it was aborted, they need to undo the transaction based on their local logs

Two-Phase Commit Protocol



Handling of Failures - Server Failure

When server S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: server executes **redo** (*T*)
  - Uses the WAL
  - **Redo:** make sure the committed transaction is also reflected in the database's memory
- Log contains <**abort** *T*> record: server executes **undo** (*T*)
  - Uses the WAL
  - **Undo:** make sure the database's memory goes back to the state before the transaction
- Log contains <**ready** *T*> record: server must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
  - Basic idea: the coordinator knows whether it committed
- The log contains no control records concerning *T* implies that S_k failed before responding to the **prepare** *T* message from C_i
  - Since the failure of S_k precludes the sending of such a response C_i must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating servers must decide on *T*'s fate:
  1. If an active server contains a <**commit** *T*> record in its log, then *T* must be committed
  2. If an active server contains an <**abort** *T*> record in its log, then *T* must be aborted
  3. If some active participating server does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*
  4. If none of the above cases holds, then all active servers must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> of <**commit** *T*>). In this case active servers must wait for C_i to recover, to find decision
     - Why?
- **Blocking problem**: active servers may have to wait for failed coordinator to recover!
  - They don't know if the transaction was aborted or committed

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol
- If the coordinator and its participants belong to several partitions:
  - Servers that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm done, but servers may still have to wait for decision from coordinator
- The coordinator and the servers that are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol
    - Again, no harm done

Recovery and Concurrency Control

- **In-doubt transactions** have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.
- The recovering site must determine the commit-or-abort status of such transactions by contacting other servers; this can be slow and potentially block recovery
- Recovery algorithms can note lock information in the log
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*>
    - *L* = list of locks held by *T* when the log is written (shared locks can be omitted)
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired
  - After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions
  - But any transaction that is "waiting for" these locks has to keep waiting until the stuck transaction is committed/aborted

end of material for mid term

Avoiding Blocking During Consensus

- Blocking problem of 2PC is a serious concern
  - **Any** participant that fails or is delayed will block all other nodes!
- Idea: involve multiple nodes in decision process, so failure of a few nodes does not cause blocking as long as majority don't fail
- More general form: **distributed consensus problem**
  - A set of *n* nodes need to agree on a decision
  - Inputs to make the decision are provided to all the nodes, and then each node votes on the decision
  - The decision should be made in such a way that all nodes will "learn" the same value even if some nodes fail during the execution of the protocol, or there are network partitions.
  - Further, the distributed consensus protocol should not block, as long as a **majority** of the nodes participating remain alive and can communicate with each other
- Several consensus protocols, Paxos and Raft are popular
- Consensus is also used to ensure consistency of replicas of a data item
  - For example, can be used to protect against failure of master nodes (e.g., the NameNode in HDFS)

Using Consensus to Avoid Blocking

- After getting response from 2PC participants, coordinator can initiate distributed consensus protocol by sending its decision to a set of participants who then use consensus protocol to commit the decision
  - If coordinator fails before informing all consensus participants
    - Choose a new coordinator, which follows 2PC protocol for failed coordinator
    - If a commit/abort decision was made as long as a majority of consensus participants are accessible, decision can be found without blocking
  - If consensus process fails (e.g., split vote), restart the consensus
    - Split vote can happen if a coordinator send decision to some participants and then fails, and new coordinator send a different decision
- The **three phase commit** protocol is an extension of 2PC which avoids blocking under certain assumptions
  - Ideas are similar to distributed consensus

Summary of 2PC

- Each server can be both a coordinator and a transaction manager
- Coordinators first prepare transaction, if all transaction managers respond they are ready, then coordinator can commit
- If coordinator fails immediately after all managers respond they are ready but before coordinator committed, we have to wait for it to recover to decide what to do
- This blocking problem can be solved using a consensus protocol, which replicates the state of each coordinator across multiple nodes, so if one fails we can still make progress without waiting for it to recover