LE 1: Introduction

- **The 8 fallacies**
  - The network is reliable.
  - Latency is zero.
  - Bandwidth is infinite.
  - The network is secure.
  - Topology doesn't change.
  - There is one administrator.
  - Transport cost is zero.
  - The network is homogeneous

Availability & high-availability

- Proportion of time a system is in a **functioning state**, i.e., can be used, also as **1 – unavailable**
- **Ratio** of time usable over entire time
  - Informally, uptime / (uptime + downtime)
  - System that **can be used** 100 hrs out of 168 hrs has availability of 100/168
  - System could be up, but not usable (outage!)
- Specified as decimal or percentage
  - Five nines is 0.99999 or 99.999% available

Reliability

- Probability of a system to **perform** its **required functions** under **stated conditions** for a **specified period of time**.
- To run continuously without failure
- Expressed as
  Mean Time Between Failure (MTBF), failure rate

LE 2: Time Sync

Real time clock (RTC, CMOSC, HWC)

- RTC is used even when the PC is hibernated or switched off
  - Based on alternative low power source
  - Cheap quartz crystal (<$1), inaccurate (+/- 1-15 secs/day)

- Referred to as "**wall clock**" time
  - Synchronizes the **system clock** when computer on
  - Should not be confused with **real-time computing**

- IRQ 8

- sysfs interface **/sys/class/rtc/rtc0 … n**
  UNIX: cat /sys/class/rtc/rtc0/since_epoch
  /sys/class/rtc/rtc0/wakealarm

Source: Wikipedia

Measuring time: System clock
(Maintained by OS when turned on)

- Used when computer is online (booted with RTC)
- POSIX
  - #include <time.h>
    time_t time(time_t * t)
    - Returns the time in number of seconds since 1970-01-01 00:00:00 +0000 UTC (the Epoch) in 1s resolution
  - #include <sys/time.h>
    int gettimeofday(struct timeval *tv, struct timezone *tz)
    - Returns time (seconds, microseconds) & timezone
- Java
  - Java.lang.System.currentTimeMillis()
  - 1 ms resolution

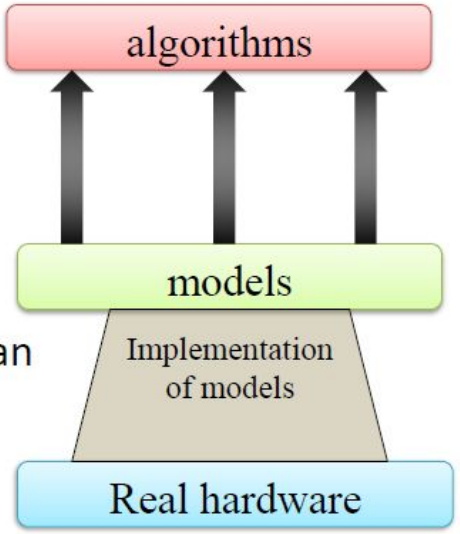
RTC vs. system clock summary

- Real-time clock:
  - Isolated clocks with external battery:
  - Uses a quartz crystal (32.768 kHz)
  - Cheap and imprecise: +/- 15 seconds/day, $1
  - Designed for offline use (eg., wakealarm)
- System clock:
  - Online only
  - System calls affected by pre-emption: time(), gettimeofday()
  - *Can be synchronized through the network*
- Epoch time (POSIX/UNIX time):
  - Seconds since January 1st, 1970
  - Uses a 32-bit signed integer
  - Y2038 problem: January 19th, 2038 -> 13th December, 1901

LE 3: Coordination and Agreement

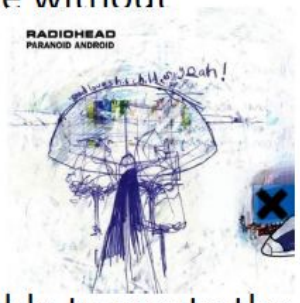
Distributed system model

- Model captures all the **assumptions** made about the system.
- This includes network, clocks, processes, etc.
- Algorithms always assume a certain model.
  - Some algorithms are only possible in stronger models.
- A model is **theoretical**: whether it can be implemented is another issue!
- There is a trade-off between the **difficulty of implementing a model** and **difficulty of implementing an algorithm**



Real systems vs. Models

- Asynchronous model is too weak
  - Real systems have clocks, albeit inaccurate
  - Most communication falls within reasonable bound
- Synchronous model is too strong
  - Precise time synchronization is impossible without specialized hardware
- Clean fail-stop model is too weak
  - Crashes are not usually **clean**
  - Failure detection is not **reliable**
- Byzantine model is too strong
  - Assumes a powerful adversary which is able to create the worst case scenario (paranoid!)
- Therefore, describe your model **in details**!



Mutual exclusion requirements

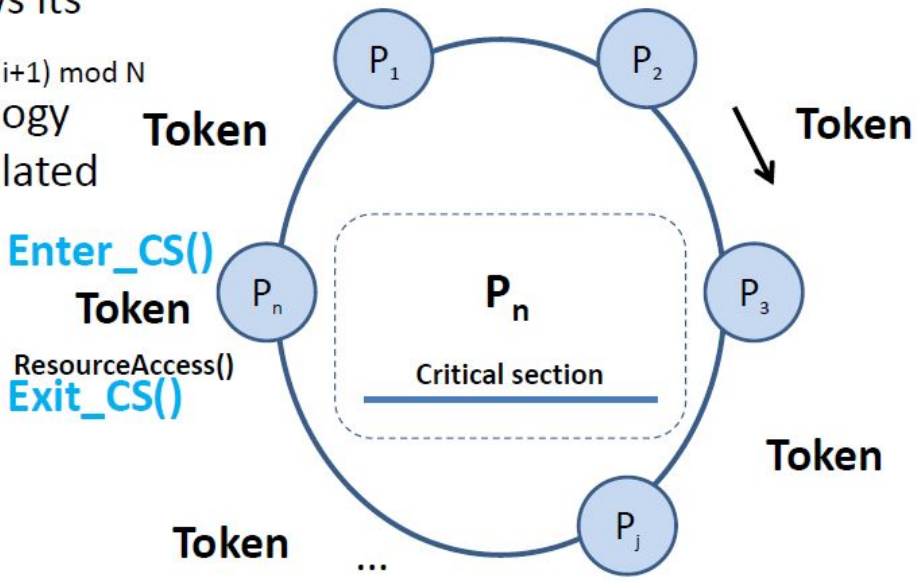
- **Safety**
  - At most one process in the critical section at a time
- **Liveness**
  - Requests to enter & exit CS eventually succeed
  - No deadlock
- **Fairness** (order & starvation)
  - If one request to enter CS **happened-before** another one, then entry to CS is granted in that order
  - Requests are ordered such that no process enters the critical section twice while another waits to enter (i.e., **no starvation**)

Solution strategies

- **Centralized strategy**
  - Divide processes into **master** and **slaves**, master dictates actions of slaves
- **Distributed strategy**: Each process independently decides actions, based on local knowledge of others' state
  - **Token-based**: A node is allowed in the critical section (CS) if it has a token. Tokens are passed from site to site, in some (priority) order.
  - **Non-token-based**: A node enters CS when an **assertion becomes true**. A node communicates with other nodes to obtain their states and decides whether the assertion is true or false.

Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



Ring-based algorithm analysis

- **Safe**: Node enters CS only if it holds the token
- **Live**: Since finite work is done by each node (can't re-enter), the token eventually gets to each node
- **Fair**: Ordering is based on ring topology, no starvation (pass between accesses)
- **Performance**
  - **Constantly consumes network bandwidth**, even when no processes seek entry, except when inside the CS
  - **Synchronization delay**: Between 1 and N messages
  - **Client delay**: Between 0 and N messages; 0 for exit (asynchronous)

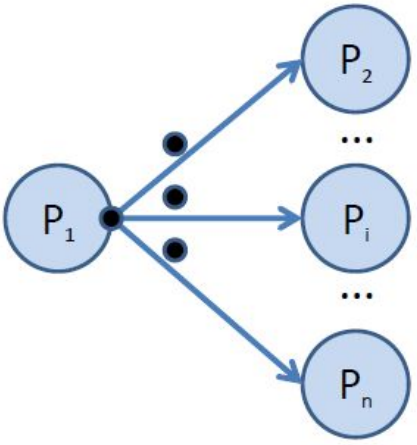
Problems with the ring-based algorithm

- Lost token
  - In a omission failure model
  - How to regenerate token
  - Violates **liveness**
- Duplicate token
  - Token regenerated when it is not lost!
  - How to tell the difference between failed and late token?
  - Violates **safety**
- Timeouts on token passing
  - Very difficult because processing time inside a CS is arbitrary!
  - Can not tell between a slow process and a failed one.

Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among N processes)

- **Basic idea**
  - Processes wanting to enter CS, **broadcast a request to all processes**
  - Enter CS, once **all** have **granted request**
- Use **Lamport timestamps** to order requests: <T, P_i>, T the timestamp, P_i the process identifier



Ricart & Agrawala: Distributed strategy

- Each process is in one of three states
  - **Released** - Outside the CS, Exit_CS()
  - **Wanted** - Wanting to enter CS, Enter_CS()
  - **Held** - Inside CS, RessourceAccess()
- If a process requests to enter CS and **all** other **processes** are **in the *Released state***, **entry** is **granted** by each process
- If a process, P_i, requests to enter CS and another process, P_k, is inside the CS (***Held state***), then P_k will not reply, until it is finished with the CS

Election algorithm requirement

- **Safety**
  - A participating process, $\mathbf{P_i}$, has variable $\mathbf{elected_i}$ = "⊥" or $\mathbf{elected_i = P}$, where P is chosen as the non-crashed process at the end of the election run with the **largest identifier.** (Only one leader at a time!)
- **Liveness**
  - **All processes participate** in the election and **eventually** either set $\mathbf{elected_i \neq}$ "⊥" or **crash**.
- Performance
  - **Total number of messages** sent (bandwidth)
  - …

Chang & Roberts Ring-based algorithm, 1978

- Essentially three phases
  1. Initialization
  2. Election phase (concurrent calls for election)
     - Determine election winner (voting phase)
     - Reach point of message originator
  3. Announce the leader phase (victory announcement phase)

The FLP result

- Fischer, Lynch and Paterson, 1985

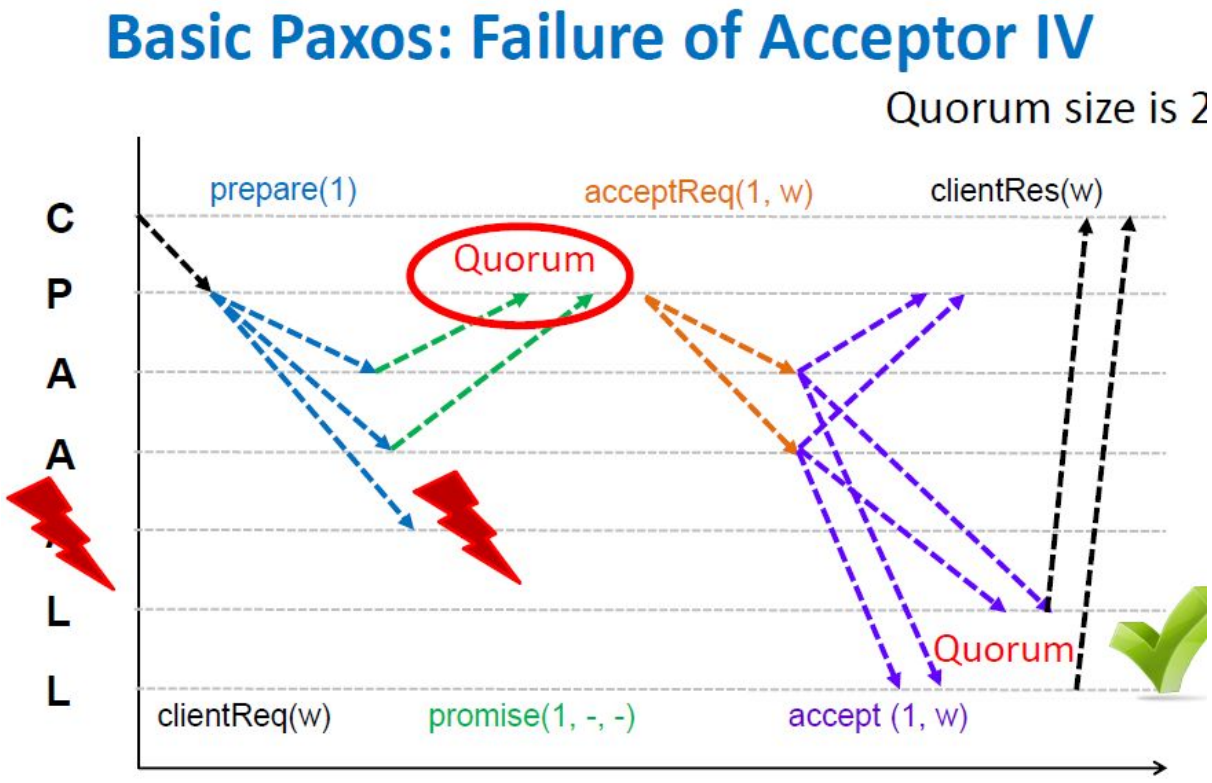
  In asynchronous systems, with only one process crashing, there is no guarantee to reach consensus

- Does not mean that consensus can never be reached
- But under the model's assumptions, no algorithm can **always** reach consensus in bounded time
- Safety is still possible to guarantee
- In practice, we can make the "no guarantee" scenario hard to occur (cf. Paxos algorithm)

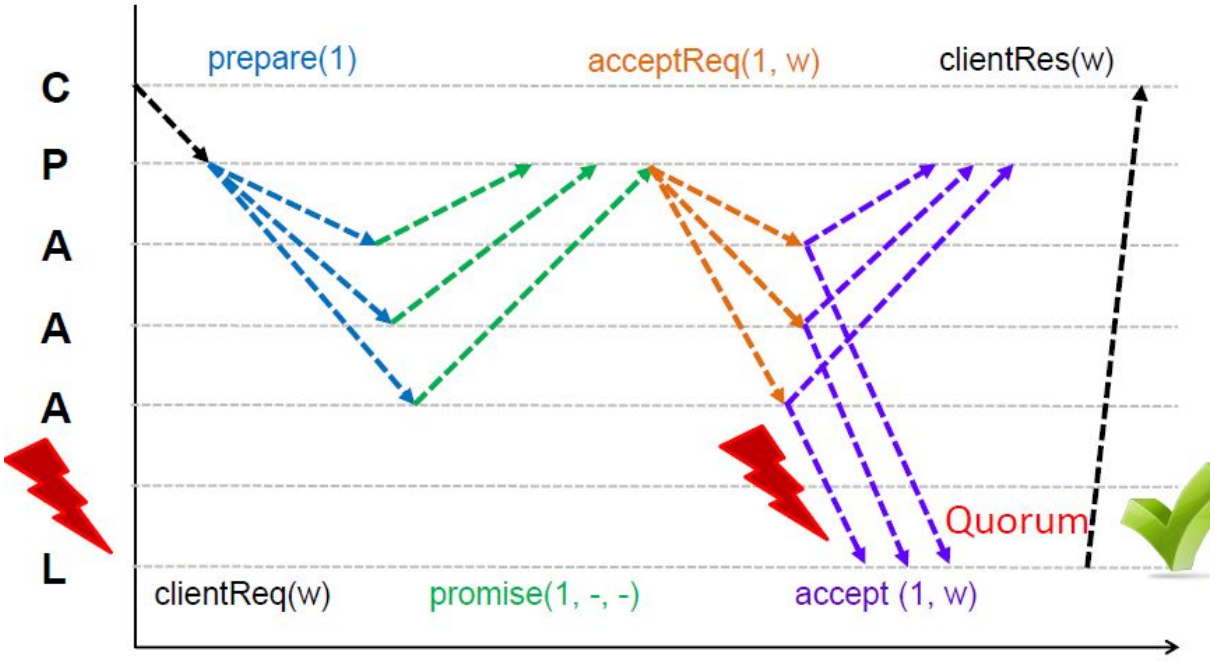
Last week review

Algorithm name	Type	DS and Failure Models
Centralized strategy	Mutual Exclusion	Asynchronous, inactive processes can crash, reliable messages
Token-based ring algorithm	Mutual Exclusion	Asynchronous, no process failures, reliable messages
Ricart & Agrawala	Mutual Exclusion	—
Changs & Roberts (Ring-based)	Leader Election	Asynchronous, no failures during election, reliable messages
Bully algorithm	Leader Election	Synchronous, process failures, reliable messages
Dolev-Strong algorithm	Consensus (aka agreement)	Synchronous, *f* failures with f+1 rounds, reliable messages

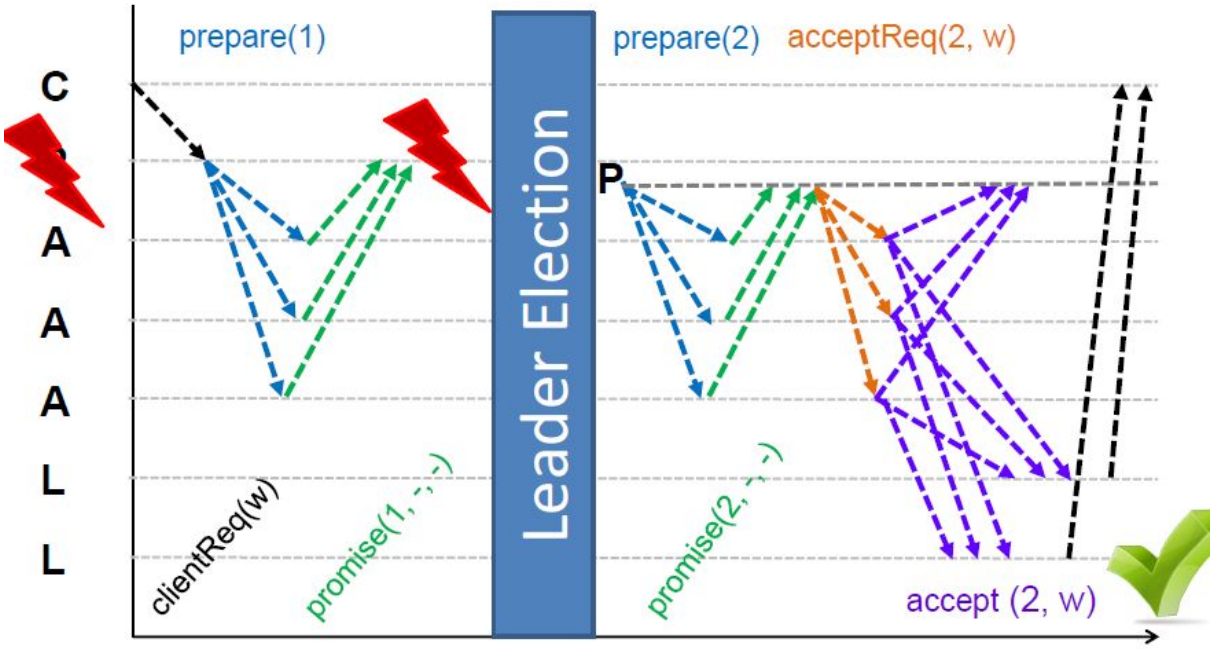
LE 4: Consensus and Replication



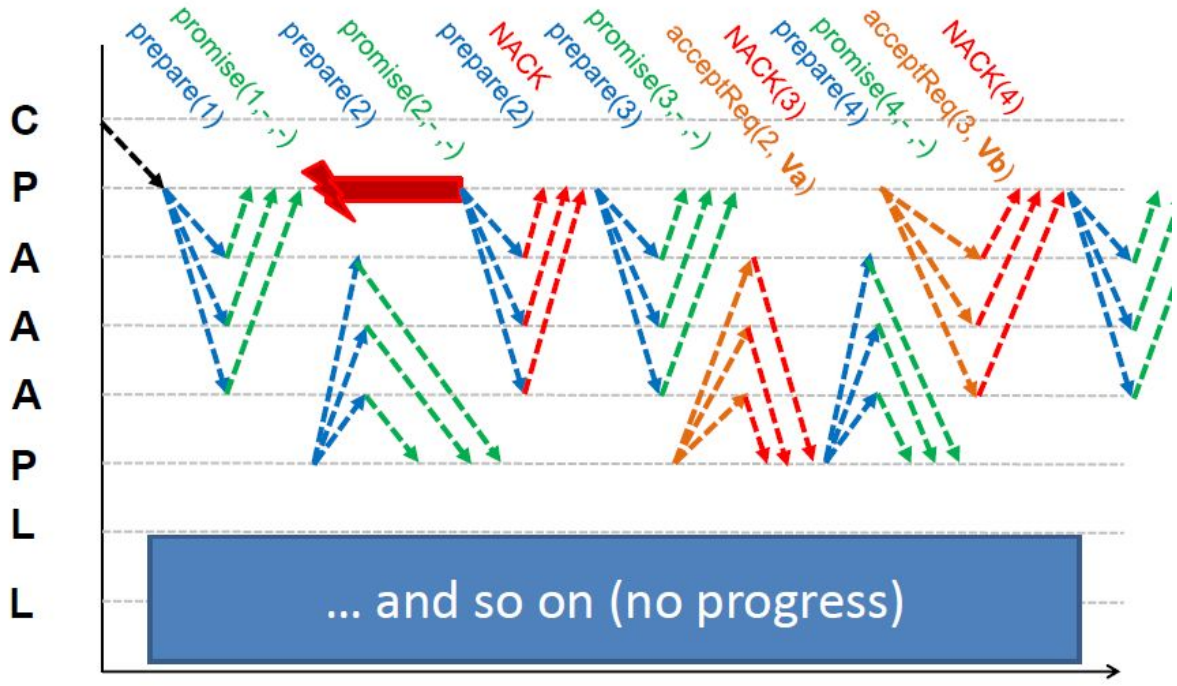
Failure of redundant Learner II



Basic Paxos: Failure of Proposer III



Dueling Proposers



Active Replication

- The client must send their request to every replica
- Requires total order broadcast messaging to guarantee each replica will receive ALL requests (**from ALL clients**) in the same order
- Since this is a RSM, every replica will send the same response back
- Easier to implement since it is like client-server
- Also faster to get a response, since the first result received can be used

Passive Replication: Primary-Backup

- A Replica is chosen to be the Primary (use Leader Election)
- **Primary**
  - Receives invocations from clients
  - Execute requests and sends back answers
  - Replicates the state to other replicas
- **Backup**
  - Interacts with the primary only
  - Is used to replace the primary when it crashes
- Called **Eager** replication if the replication is performed within the transaction boundaries (e.g. before the reply is sent)

Gossiping protocols

- Gossiping protocols disseminate information in an incremental manner
  - Avoids overloading processes with heavy broadcast messages
  - The drawback is that it takes more time to fully propagate some information
- Each peer maintains a **partial view** of other peers
- During each gossip round, each peer chooses a **random** node from its view to exchange information about:
  - Some application data (e.g., current state)
  - Its partial view
- The peers then update their state and partial view based on the information received
- Gossiping happens **periodically** and **non-deterministically**
- Used in Cassandra for propagating the status of each node (alive/dead)

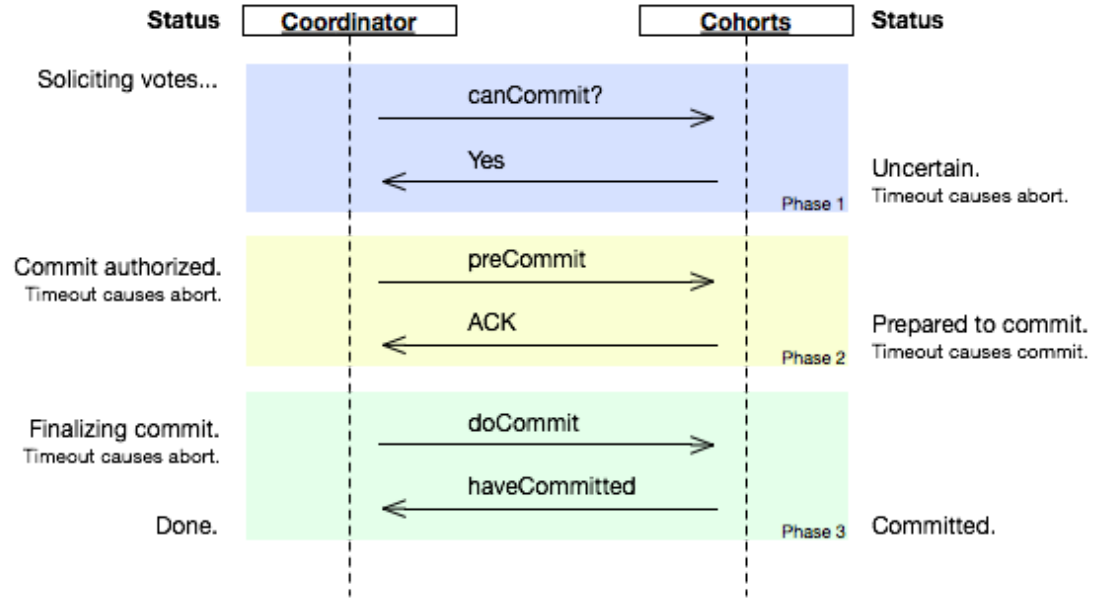
https://en.wikipedia.org/wiki/Consistency_model#Client-centric_Consistency_Models[19]

2PC

Message flow [edit]

```
Coordinator                                        Cohort
                        QUERY TO COMMIT
                    ---------------------------------->
                        VOTE YES/NO                 prepare*/abort*
                    <----------------------------------
commit*/abort*          COMMIT/ROLLBACK
                    ---------------------------------->
                        ACKNOWLEDGMENT              commit*/abort*
                    <----------------------------------
end
```

3PC



LE 6: CAP, Hashing etc.

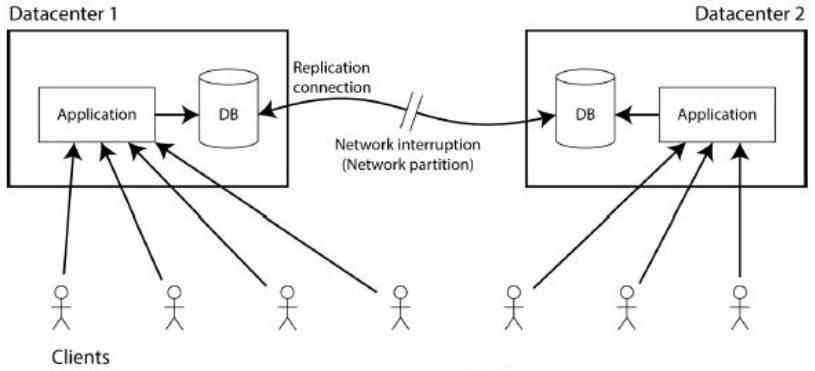
Definition of Consistency

- Refers to **replication consistency**
  - This is NOT related to *ACID properties* for transactions
- Ideally means **strict consistency**
  - But we know this is impossible without hardware clock synchronization
- Assumed to be **linearizability**
- This usually means replication across sites should be done **eagerly**

Definition of Availability

- **Every request** received by a non-failed node must result in a **non-error response**
  - **Non-triviality** requirement: a system which always responds with errors is **not available**
- Assumes a **crash failure** model for processes
  - **Functioning nodes** must continue to operate even if there are **failed machines**
- No requirement on **latency**: response can be very slow
- Both a **weak and strong** definition: no latency guarantee, but 100% response success

Definition of Partition-tolerance



- System works using an unreliable network model
- System model:
  - Asynchronous
- Network failure model:
  - Arbitrary delays
  - Message losses
- Partition: 100% loss

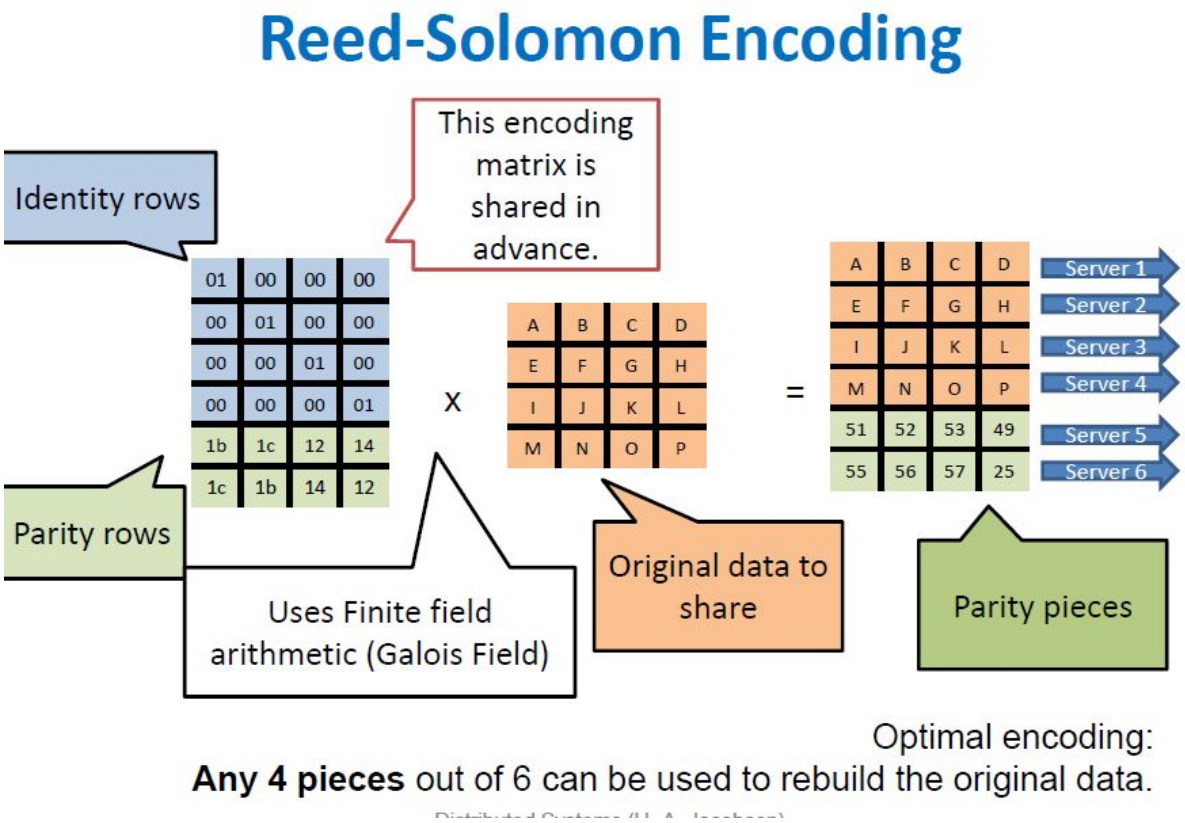
- If the system requires a **stronger system model**, or a **weaker failure model**, then it is **not partition-tolerant**
- No guarantee that **partitions recover**, but it doesn't mean they are **always present** either

PACELC

Examples

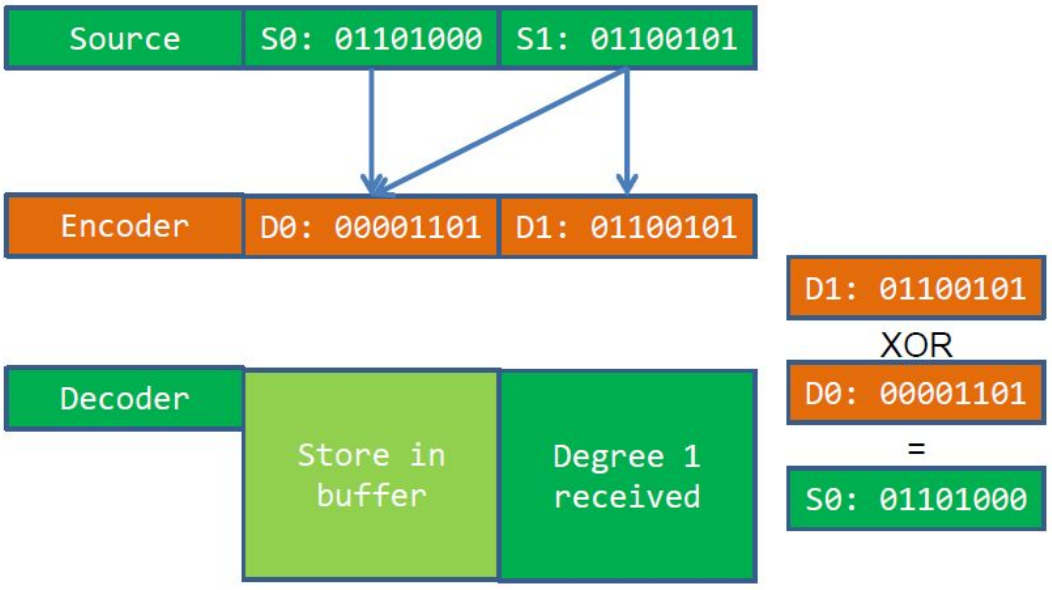
- PA/EL: Give up consistency at all times for availability and lower latency:
  - Dynamo, Cassandra (tuneable), Riak, web cache
- PC/EC: Refuses to give up consistency, pay the cost in availability and latency:
  - BigTable, Hbase, VoltDB/H-Store
- PA/EC: Gives up consistency when a partition happens, keep consistency normally:
  - MongoDB
- PC/EL: Keep consistency when partition occurs, but gives up consistency for latency during normal operations:
  - Yahoo! PNUTS

LE 7: Distributed FS



Micro Example (LT Code)

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



LE 8: MapReduce

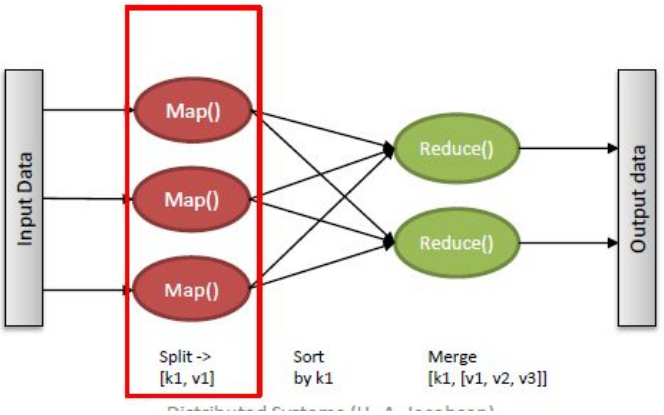
MapReduce

- Programming model & runtime system for processing large data-sets
  - E.g., Google's search algorithms (PageRank: 2005 – 200TB indexed)
  - Goal: make it easy to use 1000s of CPUs and TBs of data with commodity
- Inspiration: Functional programming languages
  - Programmer specifies only "what"
  - System determines "how"
    - Schedule, parallelism, locality, communication..
- Ingredients:
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

Map()

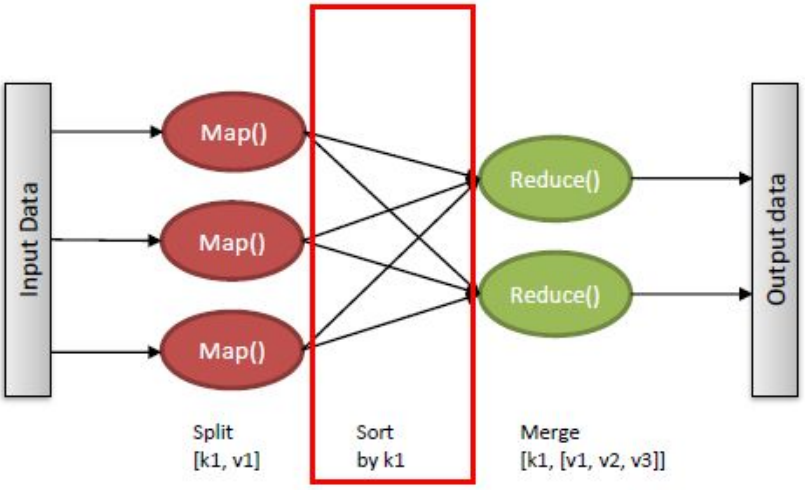
- Records from the data source (lines out of files, rows of a database, etc.) are fed into the map function as key-value pairs: e.g., (filename, line)

- Map() produces one or more intermediate values along with an output key from the input



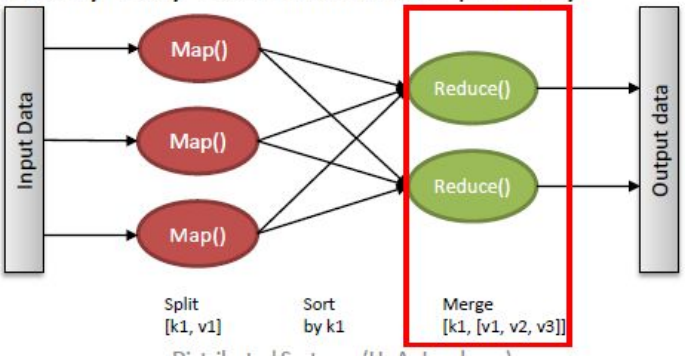
Sort and schuffle

- Map reduce framework
  - Shuffles and sorts intermediate pairs based on key
  - Assigns resulting streams to reducers



Reduce()

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- Reduce() combines those intermediate values into one or more final values for that same output key
- In practice, usually only one final value per key



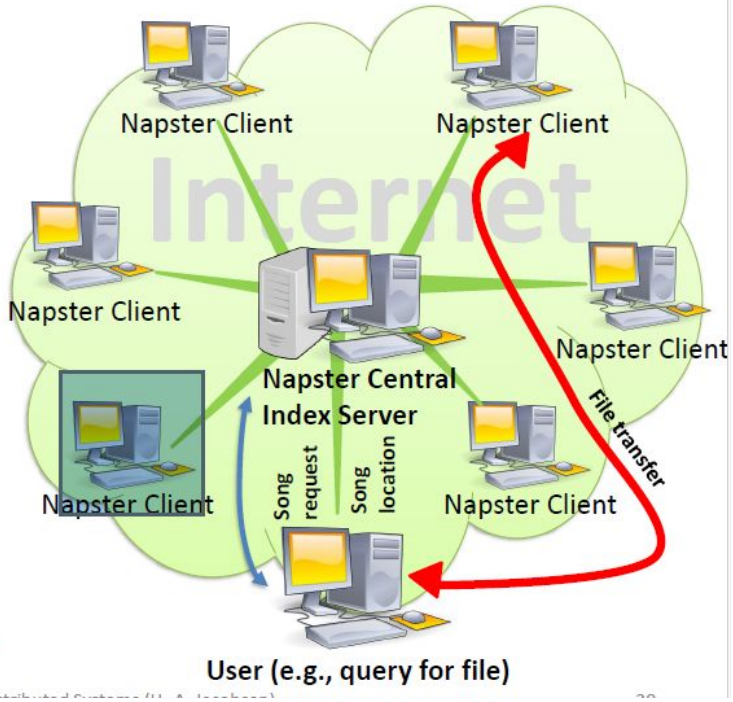
MapReduce execution stages

- Scheduling: assigns workers to map and reduce tasks
- Data distribution: moves processes to data (Map)
- Synchronization: gathers, sorts, and shuffles intermediate data (Reduce)
- Errors and faults: detects worker failures and restarts
  - Also for avoiding the **straggler** problem: slow workers

LE 9: P2P-Systems

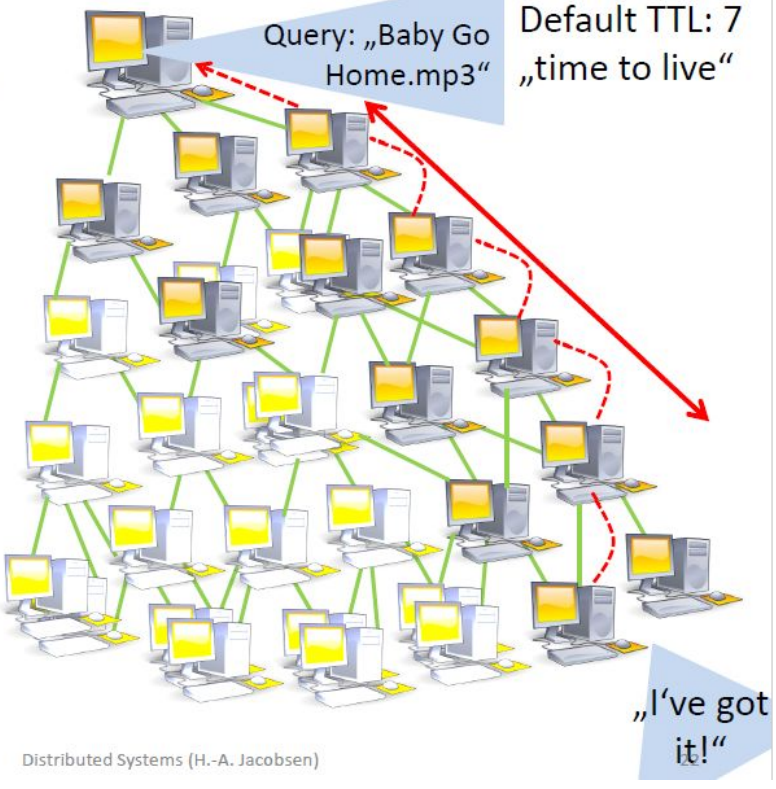
Centralized: Napster "June 1999 – July 2001"

- Centralized search indexes music files
  - Perfect knowledge
  - Bottleneck
- Users query server
  - Keyword search (artist, song, album, bit rate, etc.)
- Napster server replies with IP address of users with matching files
- Querying users connect **directly** to file providing user for download



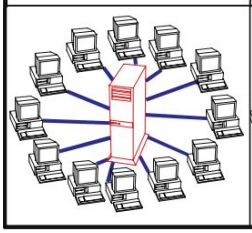
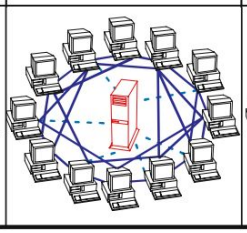
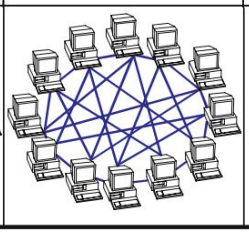
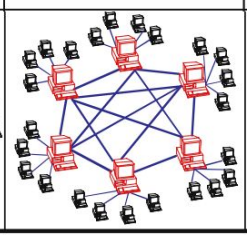
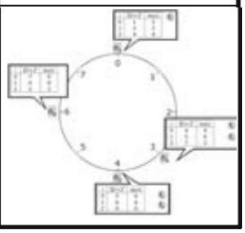
Pure P2P: Gnutella 0.4 (2000 – 2001)

- Share any type of files
- Decentralized search
  - **Imperfect** content availability
- Client connect to (on average) 3 peers
- **Flood a QUERY** to connected peers
- Flooding propagates in network up to TTL
- Users with matching files **reply with QUERYHIT**
- Flooding wastes **bandwidth**: Later versions used more sophisticated search



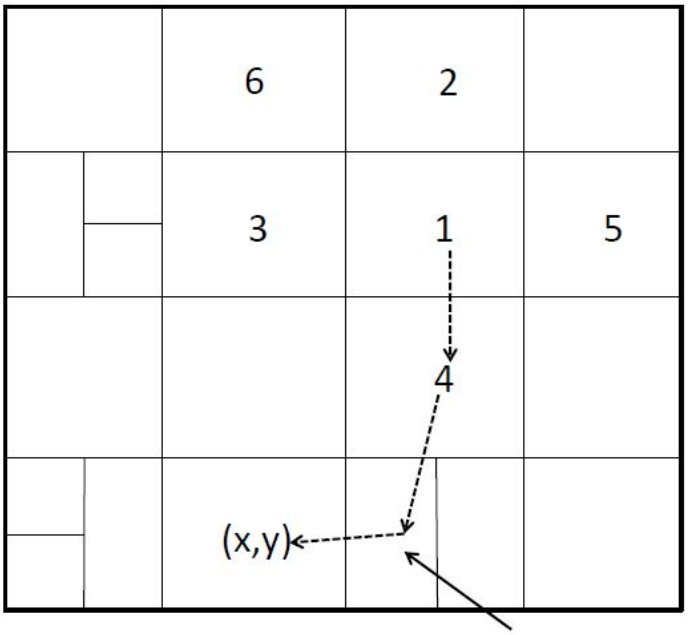
Hybrid P2P: Skype Protocol (2004)

- Both previous approaches have advantages and drawbacks
  - Centralized: single point of failure, easy to censor, but perfect content availability
  - Pure: decentralized (resistant), costly and unreliable search
- Hybrid P2P combines both approaches
  - Hierarchy of peers
  - Superpeers with more capacity, can be chosen dynamically
  - Normal peers (leaf nodes) are the users
- Superpeer responsibilities
  - Participates in the search protocol, indexes some data
  - Improves content availability
  - Reduces message load

Client-Server	Peer-to-Peer			
	1. Resources are shared between the peers 2. Resources can be accessed directly from other peers 3. Peer is provider and requestor (Servent concept)			
	Unstructured P2P			Structured P2P
	1st Generation		2nd Generation	
1. Server is the central entity and only provider of service and content. → Network managed by the Server 2. Server as the higher performance system. 3. Clients as the lower performance system Example: WWW	*Centralized P2P* 1. All features of Peer-to-Peer included 2. Central entity is necessary to provide the service 3. Central entity is some kind of index/group database Example: Napster	*Pure P2P* 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities Examples: Gnutella 0.4, Freenet	*Hybrid P2P* 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → dynamic central entities Example: Gnutella 0.6, JXTA	*DHT-Based* 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities 4. Connections in the overlay are "fixed" Examples: Chord, CAN
				

CAN routing

- Put(key, data), get(key)
- Greedily forward message to neighbor closest to destination in Cartesian coordinate space
- Nodes maintain a routing table that holds IP address and zone of its neighbours



Sample routing path from node 1 to point (x,y)

1`s coordinate neighbor set = {2,3,4,5}

LE 10: Pub / Sub

Why Decoupling?

- Decouple publishing and subscribing clients
  - Removes explicit dependencies
  - Reduces coordination
  - Reduces synchronization
- Increases scalability of distributed systems
- Creates highly dynamic, decentralized systems
- Decoupling in at least three dimensions
  - Space
  - Time
  - Synchronization (flow)

Summary of Pub/Sub

- Pub/sub provides **many-to-many** communication between publishers and subscribers.
- Communication is decoupled with respect to **time, space, and synchronization**.
- Many applications are **event-based** or require **event notifications**, which leverage pub/sub.
- Many flavors of pub/sub, which differ in **matching** and **routing**.

LE 11: Blockchain

Byzantine generals, commander variant

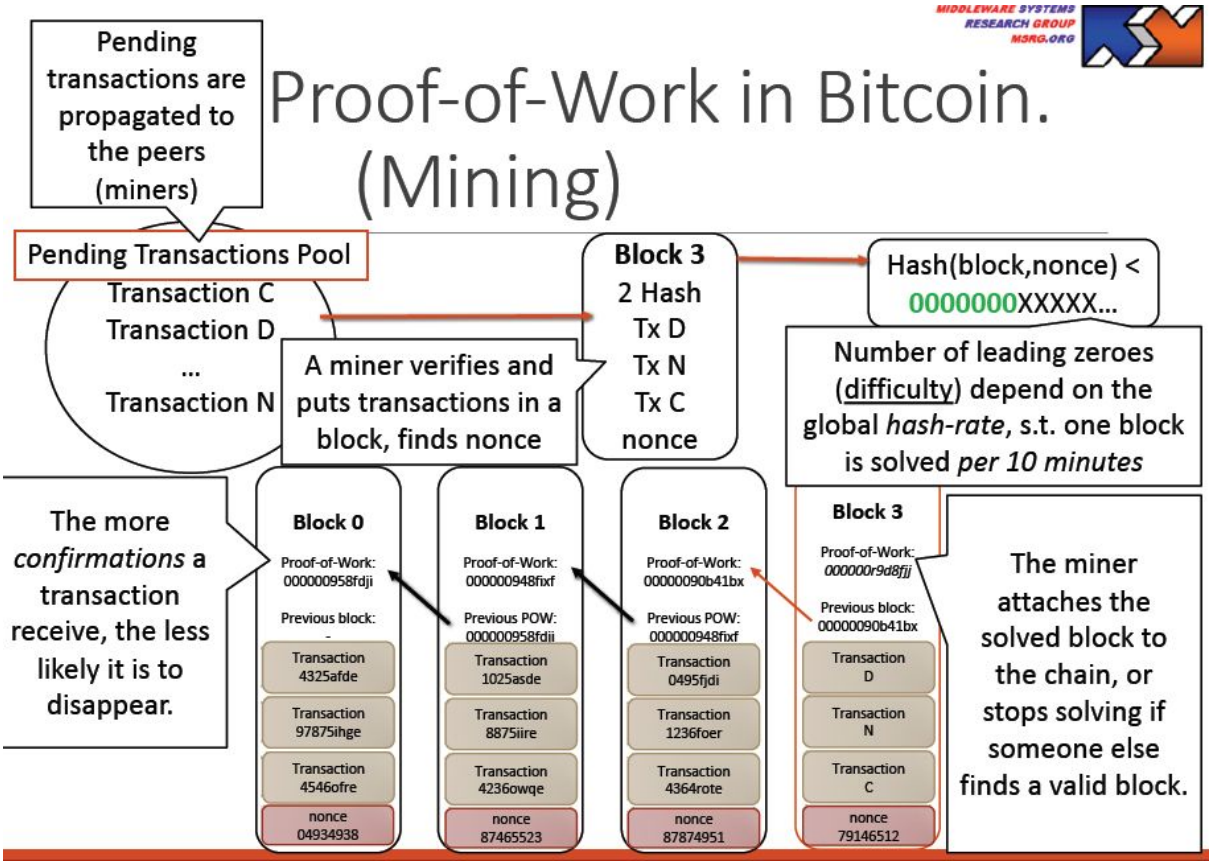
- A **distinguished process** (the *commander*) proposes initial value (eg. "attack", "retreat")
- Other processes, the *lieutenants*, **communicate the commander's value**
- **Malicious processes** can lie about the value (i.e., **are faulty**)
- **Correct processes** report truth (i.e., **are correct**)
- Commander or lieutenants may be faulty
- Consensus means
  - If the commander is correct, then correct processes should agree on his proposed value
  - If the commander is faulty, then all correct processes agree on a value *(any value, could be the faulty commander's value!)*

Byzantine generals: 3f + 1

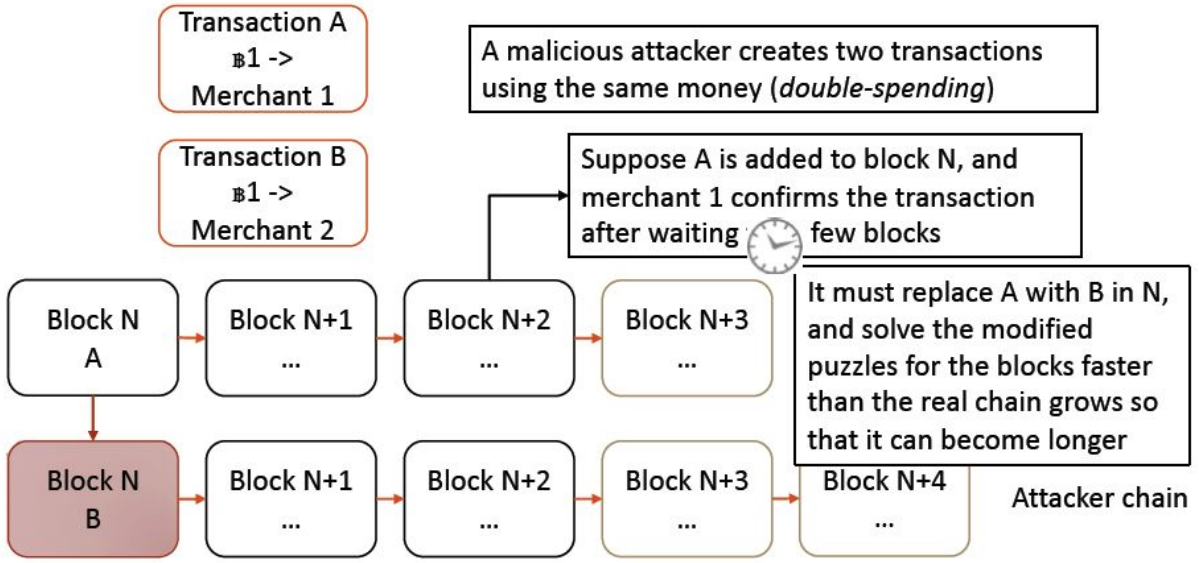
- Lieutenants **are not voting for their own preference**. They are reporting what they heard from the commander.
- Lieutenants decide based on the majority of reports received.
- **For *f* faulty processes, $N \geq 3f + 1$ required to reach consensus**

Blockchains Overview

- A distributed **ledger** of data which is replicated across many miners consisting of blocks chained together.
- To **add a block** to the chain, a miner must correctly **solve a puzzle** involving the content of the new block and the previous block.
- Miners should **continuously add blocks** to the **longest chain** they have seen so far.
- **Old blocks** gain **trust** the longer the chain grows.
- To **modify an old block**, an attacker has to produce a chain **longer** than the current one, by solving its puzzles **faster** than the rest of the miners working on the real blockchain.
- This requires an **enormous of resources** and/or luck.

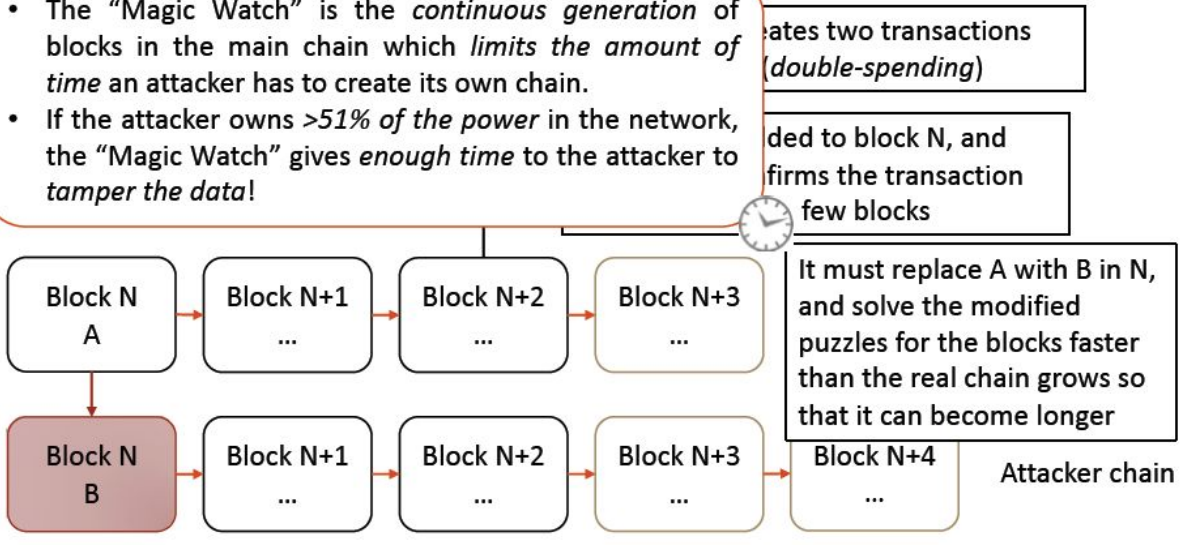


Preventing Double Spending: 51% Attack



Preventing Double Spending: 51% Attack

- The "Magic Watch" is the *continuous generation* of blocks in the main chain which *limits the amount of time* an attacker has to create its own chain.
- If the attacker owns *>51% of the power* in the network, the "Magic Watch" gives *enough time* to the attacker to *tamper the data*!



Proof-of-Stake

- Instead of mining with computing power, mine with virtual currency
- Essentially, each coin a miner has is a lottery ticket.
- A random generator draws the winning coin; the probability of a winner is proportional to the number of coins it has.
- 51% attacks are only possible if a miner owns 51% of the total currency in the system
- Can be used to reduce the difficulty of the work in PoW
- First system: PeerCoin (2012)
- Ethereum to adopt later

Proof-of-Stake Details

verify() function in PoS:

- *sha256(PREVHASH + ADDRESS + TS) <= 2^256 * BALANCE / DIFFICULTY*
- ADDRESS of wallet of the miner, BALANCE is the recorded stake for the wallet
- TS is the timestamp in UNIX time (seconds)
- Thus, only one hash needed per second (per wallet)

Branches can still exist in PoS:

- Due to propagation delays, multiple timestamps are valid for a block
- The puzzle function does not return an unique winner

Nothing-at-Stake problem:

- PoW: cannot mine parallel branches since splitting resources is not effective
- PoS: mining parallel branches is easy since it only requires 1 hash/s
- Slasher algorithm: detection of parallel mining confiscates the stake

LE 12: Cloud Computing

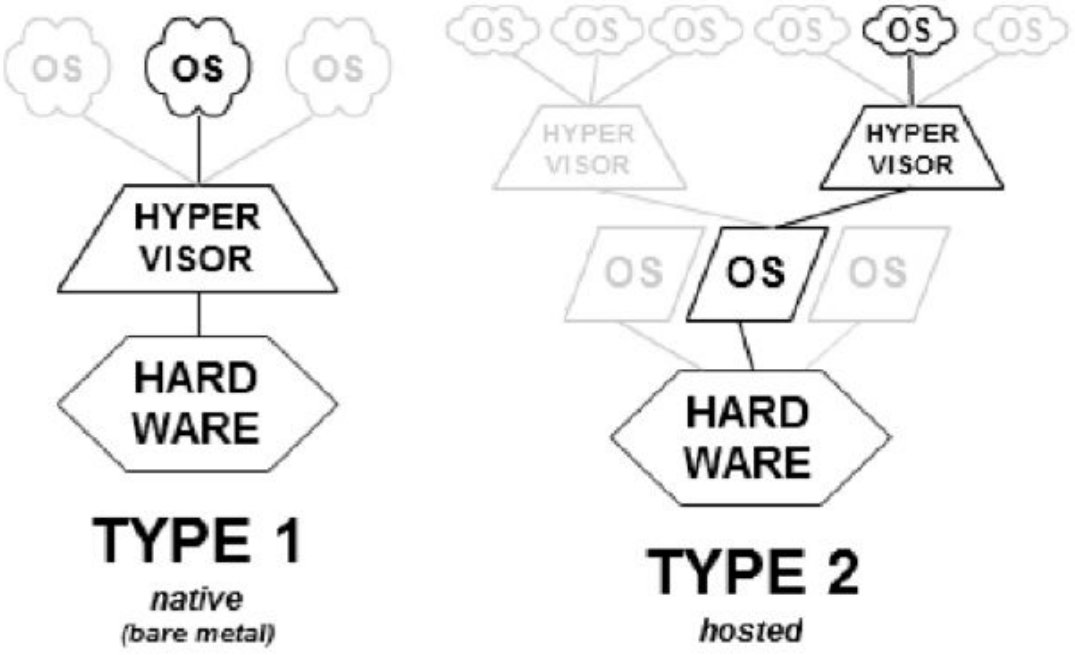
Hypervisor

- Hypervisor
  - Creates the virtualization layer that makes server virtualization possible
  - Contains the Virtual Machine Monitor (VMM)
- Example of Hypervisors
  - KVM
  - Oracle VirtualBox

Type 1 vs. Type 2 Hypervisor

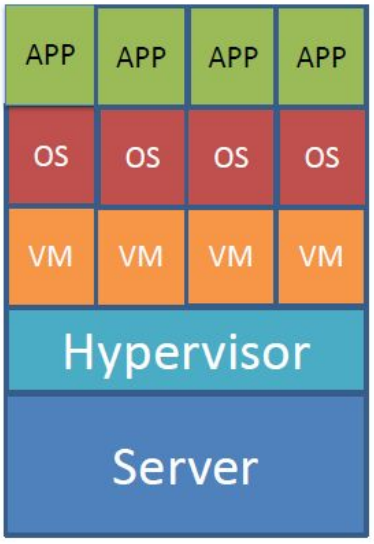
- Type 1 Hypervisor is loaded directly on the hardware (a.k.a native or bare-metal)
  - Microsoft Hyper-V
  - VMware ESX/ESXi
  - KVM
- Type 2 is loaded on an OS running on the hardware (a.k.a hosted hypervisor)
  - Oracle VirtualBox
  - VMware Workstation

Type 1 vs. Type 2 Hypervisor



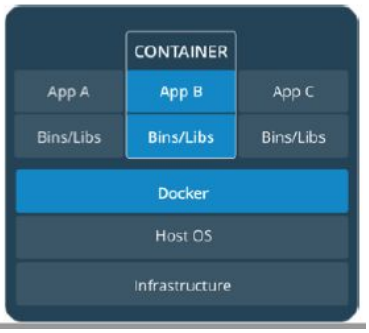
Disadvantages of Hypervisors

- The Server's resources are shared among VMs
- Execution of OS consumes resources (CPU/RAM/Disk) which is independent of the app
- OS can be expensive and requires administration



Containers

- Containers help with execution of apps
- Containers provide isolated area (processes) on top of OS areas where apps can run
  - Docker, most well-known Container Management
- OS does not provide much isolation for apps by default, which is not secure enough and leads conflicts
  - Virtual Memory Isolation
  - Privileges of the process owner
- Less overhead than VMs



Source: Docker

What is Serverless?

- A cloud-native platform for short-running, stateless computation, and event-driven applications which scales up and down instantly and automatically and charges for actual usage at millisecond granularity
  - Auto-scalability and maintenance
  - Pay for what you use
- Serverless does not mean no servers, means worry-less servers
- Also known as Function-as-a-Service (FaaS)

What is Serverless Good For?

- Serverless is a good solution for short-running stateless event-driven operations
  - Microservice
  - Mobile Backends
  - Bots, ML Inferencing
  - IoT
  - Modest Stream Processing
  - Service Integration

What is Serverless NOT Good For?

- Serverless is not good for long-running stateless computationally heavy operations
  - Databases
  - Deep Learning Training
  - Heavy-Duty Stream Analytics
  - Spark/Hadoop Analytics
  - Video Streaming
  - Numerical Simulations