fault tolerance: MTBF (between failure) = MTTF (failure) + MTTR (recovery), availability = MTTF / MTBF
error correcting codes [forward recovery] [add info (bits) to verify correctness/bring data to correct state]:
(1) detect up to 1 (parity/XOR bit), 2 (Hamming code, $x_1, \dots, x_4, \sum_{i=2,3,4} x_i, \sum_{i=1,3,4} x_i, \dots$) single bit errors,
(2) correct 1, 2 erasure/ 0, 1 bit-flip errors, (3) rate = bits of info / total bits = 7/8, 4/7
redundant array of inexpensive disks (RAID): fail-stop model [either working or failed/erased, state known].
B: # of blocks per disk, ***R***: R/W throughput per disk, ***N***: # of disks, ***D***: time to R/W one block

	capacity	w tpt	w lat	r tpt	r lat	scheme
RAID-0	$N \times B$	$N \times R$	D	$N \times R$	D	stripping
RAID-1	$N/L \times B$	$N/L \times R$	D	$N \times R$	D	mirroring
RAID-4	$(N-1)B$	$R/2$	$2D$	$(N-1)R$	D	parity
RAID-5	$(N-1)B$	$N/4 \times R$	$2D$	$N \times R$	D	rotating parity

0- parallel transfer for large files, load balancing for small files (hot files distributed evenly)
4- all writes go to parity disk (bottleneck)

MTTDL (mean time to first data loss) = MTTF / n if n disks in set, suppose data has capacity of 200 drives
RAID-1, 400 drives → (E[time to 1st disk failure] + E[2nd failure]) / [# disk sets] = (M/2 + M)/200
RAID-4 (*fails if 2 disks in set/one set fail, 50 sets of 4 data+1 parity*), 250 drives → (M/5 + M/4)/[50 sets]

cryptography [symmetric key] [**security**] [**confidentiality**] adversary cannot read msgs, def: IND-CPA (attacker chooses 2 msg (same len), challenger encrypts one, attacker guesses which one) – deterministic schemes not secure, OTP/CTR with unique nonce||counter, [Shannon's thm: key size ≥ msg size for perfect security] [**integrity**] cannot tamper msgs undetected, [**authenticity**] can prove msg came from person who claim to have written it, use MAC (msg auth codes) (keyed checksums), e.g. hash/HMAC for existential unforgeability against CPA, deterministic/efficient/(preimage/collision) resistant, use encrypt-then-MAC [**asymmetric key model**] everyone can encrypt with public key, only recipient can decrypt with private key [RSA] pick P, Q large primes, N = PQ, choose E relatively prime to k=(P-1)(Q-1), private key D = E^{-1} mod k,
digital signatures: correct, efficient, EU-CPA secure (same as MAC, except attacker also receives pk).
transport layer security (TLS): used for protocols like HTTPS, special TLS socket layer between application and TCP, handles CIA, asym crypto to generate per-session sym keys for fast communication. Diffie-Hellman guarantees forward secrecy, not secure against MITM; certificate authority (CA) binds pk to entity → share random P, G, sends (and sign) G^A, G^B mod P, generate premaster-secret G^{AB} mod P

Byzantine Fault Tolerance: [**network setting**] synchronous (shared global clock, bounded msg delays), async (no shared global clock, arbitrarily long but finite msg delays, every msg eventually delivered), partially sync (async phase followed by sync, e.g. attack on network, followed by recovery) [**Byz Broadcast**] network of n nodes (leader S_1), honest nodes follow protocol exactly, Byz nodes (at most f) can collude, every honest node eventually halts (**termination**) with same output (**agreement**), if leader honest, equal to private input v* of leader (**validity**). [**Dolev-Strong protocol**] sps sync network, Public Key Infrastructure

Round 1: every node other than the leader initializes set$_i$= {}, leader broadcasts (v∗, [S1]), outputs v∗
Round t = 2, … , f+1, for each S_i, for each (v, [$sigs$]) received in round t−1, if $v \notin$ set$_i$ and [$sigs$] is a (t−1)-signature chain not already signed by S_i, add v to set$_i$, and append S_i's own sig to sig chain, broadcast msg
End of protocol: for msg (v, [$sigs$]) received in round f+1, if [$sigs$] is an (f + 1)-sig chain not already signed by S_i, add v to set$_i$. Decision: if |set$_i$| = 1, then S_i outputs the value, otherwise output some default value ⊥

FLM impossibility result: in a sync model with f ≥ n/3, there is no Byz broadcast protocol that satisfies termination, agreement, and validity, assuming no trusted setup (PKI) (idea: indistinguishability)
FLP: For every n ≥ 2, even with f = 1, no deterministic protocol for the Byz agreement problem satisfies termination, agreement, and validity in the async model (even with crash-fault, PKI)
but! there exists a deterministic protocol for the Byz agreement problem that satisfies agreement, validity, and eventual (post-GST) liveness in the partially sync model if and only if f < n/3 (w(o) PKI) (pre-GST: f nodes not responding might be f honest nodes partitioned away, must have majority among n−f)

P(ractical)BFT: 1 distinct primary p for a view v (view change protocol initiated if p detected as faulty) + multiple backups, all nodes maintain a local log of <index, op, status>, always safe, not necessarily always live, assumes PKI, total n = 3f+1 nodes, uses supermajority quorum 2f+1
request, pre-prepare: c sends request to primary, signs own msg with unique c-specific seqnum, p sends pre-prepare msg to all replicas, {σ_p({p-prep, v, i, d}), m}, records op in log as **p-prep**. replica accepts p-prep msg if r currently in view v, d = H(m), and r has not previously accepted a msg for index i in view v
prepare: if p-prep ok, r broadcasts a prepare message, σ_r({prepare , v, i, d}): if r receives 2f prepare msg identical to the one it sent, then least 2f+1 replicas/f+1 non-faulty nodes agree, set status to **prepared**
commit: if r prepared, broadcasts commit msg σ_r({commit , v, i, d}): if r receives 2f+1 matching commit msg, it is **committed** (f+1 honest nodes prepared), once value committed (at index), it stays committed

blockchains solves State Machine Replication problem under Byz failures using proof-of-work [state = acc associated with pk, log = transactions]. permissionless (unknown and ever-changing set of nodes running consensus → similar to no PKI, Sybil nodes-resistance: chance proportional to compute power)
(classic BFT protocols (d-s, PBFT) are permissioned: nodes running protocol known in advance)

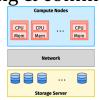
longest-chain consensus: nodes initialized with hard-coded genesis block, in each round, randomly choose one node L as leader (puzzle winner, verifiable), L proposes set of blocks, each specifying a single predecessor block (hash chains: append-only, immutable data structure), blocks gossiped to all nodes.
• blocks are "proposed values" with multiple forks/chains (each represents possible version of log), chosen log = longest chain, L vote on which fork to pick by appending blocks, *honest nodes append to longest chain*
• satisfies consistency (safety) and liveness in synchronous model if ≤49% of nodes are Byz
• k consecutive Byz leaders can roll back k − 1 honest blocks (p-o-w restricts to one block/round)
• if partially sync: each partition of nodes will solve crypto puzzle independently & grow different chains, once network partition ends, only one chain can win ⟹ potentially many finalized blocks rolled back
• alternative: proof-of-stake (assume adversary controls < 1/3 of stake), 1 vote/$$ (avoid energy waste)

domain name system [use human-readable strings, map to IP]: (goals) scalable, highly available, correct (no conflicts/unique, trades off consistency), fast lookups. (idea) hierarchical namespace (vs flat namesp.) (Top Level Domains (.edu, .sg) globally recognized, domain names (.cmu), subdomains (.cs)), hierarchically administered (vs centralized administrator), hierarchy of servers (vs centralized storage) (zone (contiguous section of namespace) has associated set of name servers to keep name → IP mappings for that zone) servers high in hierarchy use **iterative queries**, lower levels → **recursive** (faster response due to caching, cached DNS entries expire based on Time To Live, top-level NS records have very high TTL to alleviate load on root, low level records have small TTL for consistency concerns, negative queries/mistakes also cached).

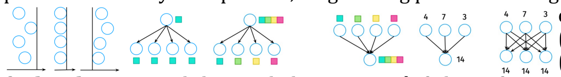
DNS resource records: (class, name, value, type, ttl),	iterative query (example):
• class = Internet (IN), type = A | NS | CNAME	<client, Local DNS, c.m.cmu.com, cm_ip>
• A: name=hostname, value=IP address	<Local DNS, Root NS, c.m.cmu.com, com_ip>
• NS: name=domain, value=name of authoritative	<Local DNS, .com NS, c.m.cmu.com, cmu_ip>
name server for this domain	<Local DNS, cmu.com NS, c.m.cmu.com, m_ip>
• CNAME: name=alias name, value=canonical name	<Local DNS, m.cmu.com NS, c.m.cmu.com, cm_ip>

content delivery network (DNS-based client routing): origin servers (in Internet core) store definitive version of content, users located at the edge, CDN work with Internet Service Providers to install Points of Presence at edge, user retrieves data by first contacting closest PoP, cache miss leads to origin server.
DNS's indirection: names → IP, CNAME (canonical name) as alias, point domain name to another more permanent DN (instead of IP): (1) load balancing: map multiple IP to same name, (2) performance optimization: configure DNS server to serve different IP based on client's region, (3) multiplexing: same IP can be used with different DNS names. origin server distributes content to CDN servers → client does name lookup for service → authoritative name server resolves to CDN name (using CNAME) → CDN high-level name server chooses appropriate CDN instance → CDN low-level name server choose specific cache server.
(1) load balancing via **round robin**, requests for same URL forwarded to different servers, high cache miss rate, low cache storage efficiency. (2) **static partition**: if by name, what if a.com had >> pages than b.com. (3) **hash-based**: "even" distribution across servers but add/removing servers hard (4) **consistent hashing**: map both nodes (severs) and keys (queries) to same identifier space (organized as a ring), keys mapped to next successor node, on average only 1/nth of entries moved when add/removing a node (smooth), keys evenly distributed across nodes (load), a given object is directed only to a subset of nodes (spread).

HPC machine: compute nodes (high end processor(s) + RAM), network (specialized, very HP), storage server (RAID-based disk array). **message passing model**: processes communicate and synchronize via exchange of msgs, programs described at very low level (specify detailed control of processing & comms), rely on small number of software packages (limits classes of problems & solution methods).



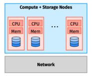
typical HPC operation: long-lived processes, partitioning (exploit spatial locality), hold all program data in memory (no disk access), high bandwidth comm, [strengths] high util of resources, [weaknesses] req careful tuning of app to resources, intolerant of any variability.
messaging passing interface: highly optimized for low latency, high scalability over HPC grids/LANs, standardized comm. protocol specifies range of functionality: virtual topology (finding # of processes, processor identity for a process, neighboring processes in logical topology), synchronization (barrier).



communication methods:
(sending) barrier, broadcast, scatter;
(receiving) gather, reduce, all-to-all;

fault-tolerance: tightly-coupled processes → failure of one process prevents all others from progressing; checkpoint (periodically store state of all processes, significant I/O traffic), restore when failure occurs (reset state to that of last checkpoint, all intervening computation wasted), poor performance scaling (very sensitive to # of failing components) → **MPI wrong level of abstraction for application writers**.
actor model [abstraction/facilitate reasoning]: each actor maintains map of other "live" actors in system, process msgs serially as received in (FIFO) mailbox, can send to any other actor (point-to-point msging).
cluster: collocate compute and storage, medium-performance processors, modest memory, a few disks, ethernet switches: partition compute tasks, run where data is stored (disks/processors in close proximity)

mapReduce: coarse-grained parallelism, maximum abstraction, zero control. **MPI**: maximum control over hardware, zero abstraction or portability between architectures. **Actors**: zero knowledge of hardware, maximum portability everywhere, middle abstraction/control.



[**Hadoop**] **mapReduce** [cluster model]: **mapping**: dynamically map input file blocks onto mappers, each generates key/value pairs from its blocks, writes R files on local file system; each reducer – **shuffling**: handles 1/R of possible key values, fetches its file from each of M mappers, sorts all of its entries to group values by keys, **reducing**: executes reducer function for each key, writes output values to cluster filesystem. [*computation broken into many, short-lived tasks (independent processes), file-based communication, failure (detect via heartbeat) → reschedule; stragglers (tasks that take v long to execute): when done with most tasks, reschedule remaining tasks, keep track of redundant executions (load balancing; reduce overall run time)*]

```
mapped = impactRDD.map(lambda x: (x[0], (x[1], 1))) # (player, gis) -> (p, (gis, 1))
totals = mapped.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1])) # (p, (total_gis, t_games))
averages = totals.map(lambda x: (x[0], x[1][0] / x[1][1])) # (p, avg_gis)
joined = averages.join(totals.map(lambda x: (x[0], x[1][1]))) # (p, (avg_gis, total_games))
eligible = joined.filter(lambda kv: kv[1][1] >= min_games)
top3 = eligible.map(lambda kv: (kv[0], kv[1][0])).top(3, key=lambda x: x[1]) # (p, avg_gis)

# cache totals: performs wide transformation and shuffle (most expensive part of computation)
```

MapReduce: force every iteration to write/read intermediate state to/from disk and across network → real-world iterative jobs (ML) spend vast majority of time on I/O (~90%) rather than compute.

[**Apache Spark**] keep reusable datasets in memory via **Resilient Distributed Dataset** cache, execute DAGs of transformations lazily (execution triggered by actions), recover via **lineage**: if partition lost, recomputes it by replaying transformation recipe (since RDDs are immutable and transformations are deterministic) [**components**] driver program (user application), context (SparkContext/sqlContext, user hands to cluster), worker program (what runs on cluster nodes), executor (what actually does the work on each cluster node_

• RDDs immutable (object's state cannot be modified after creation): enables lineage (recreate any RDD any time; RDDs must be deterministic functions of input), simplifies consistency (caching/sharing RDDs across Spark nodes), compatibility with storage interface (Hadoop DFS: chunks are append-only)

• immutability and functional transformations (can be applied in parallel) enable automatic rebuild on fail

• *Transformations* create new RDD (map, filter, sample, groupByKey, sortByKey, union, join, cross), lazy eval

• *Actions* return value to caller (count, sum, reduce, save, collect) (eager, immediately triggers evaluation)

• *Persist* RDD to memory (.persist()) (lazy/eagerness limit network comm via programming model)

drawbacks: needs lots of memory + high overhead (copying data; no mutate-in-place); not good for non-batch workloads, apps with fine-grained updates to shared state, datasets that don't fit into memory.

bulk synchronous parallel model (iterative mapReduce/approximately implemented by Spark): no computation during barrier, no comm during computation, limitation: constantly waiting for stragglers. Leslie Valiant, 1990: "Any distributed system can be emulated as local work + message passing" (=BSP).

HDFS is the open-source version of **Google File System**: treat failures as the norm → maintain high data/system availability, handle failures transparently (automatically); for large files with large, streaming reads, large, sequential writes (mostly append); high throughput over low latency: low synchronization overhead between entities of GFS (one master server + multiple chunk servers), exploit parallelism of numerous disks/servers, atomicity for concurrent appends by multiple clients (e.g. in producer-consumer queues)

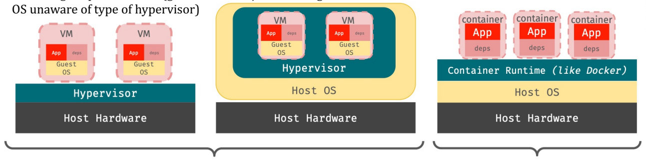
NFS: small, interactive workloads with frequent in-place updates and modest parallelism; provides close-to-open consistency and efficient attribute caching [client polls server] for fast, user-facing file operations.

AFS: read-heavy workload with low parallelism [whole-file caching with callbacks keeps server load low]

Type-1 hypervisors: native/bare metal: higher performance (guest OS unaware of type of hypervisor)

Type-2: hosted ("client hypervisor"), easier to install/use, leverages host's device drivers

Containers

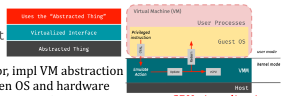


A need to improve:
1. application accessibility,
2. cross-platform compatibility,
3. system stability,
4. and resource utilization efficiency.

[**VM**] [strengths] strong isolation guarantees, can run different OSs, VM migration practical, [weakness]: OS startup, disk, memory, hypv overhead [**containers**] [strengths] fast startup time, negligible I/O overheads, very high density, [weakness] weak security isol.

• Virtualize the **hardware**
• **Heavyweight** → requires **guest OS** per VM

• Virtualize the **OS**
• **Lightweight** → Just contains **app+deps**



CPU virtualization

interposition (transformations on instructions, memory, I/O, e.g. encryption/compression)

Fault and perf **isolation**, **encapsulation** of env/libraries/etc

Fast instantiation, **low overhead** (per-operation) (e.g. I/O)

Reduced portability vs VM (still portable within OS family)

Interposition (no hypv, containers directly comm with kernel → reduced security

encapsulation + portability

isolation

hypervisor: VM monitor, impl VM abstraction
1. software layer between OS and hardware
2. functionally invisible to OS and apps
3. able to multiplex hardware among multiple VMs (hardware virtual.)

guests = VMs running on hypv, hosts = physical machine running hypv

	Compilation to bytecode.	Compilation to native code.
Portability	high	none
Security	Can be high, enforced by VM	tricky
Distribution	"compile once, run anywhere"	compile N times for N targets
Performance	high (with JIT)	higher
Footprint, code size	low	medium
Footprint, runtime	depends, but can be high	none
Footprint, memory	depends, but can be high	generally lower than with bytecode

Containers	Wasm (polyglot: many languages compile to it)
MB's to GB's	KB's to MB's (ideal for high-density workloads (higher resource util))
CPU architecture bound	Portable across heterogeneous envs
Some isolation and security	Sandboxed and secure by default (perfect to run untrusted code; secure the software supply chain!)
Starts in seconds	Starts <1ms (optimized for short-lived, event-driven workloads like Functions as a Service (FaaS); also enables rapid scalability for microservices! no idle service.)

	Java Applets	Wasm
Design Ideology	Ported to the web	Web-first, web-native, zero trust
Security	Sandboxed, but wide attack surface due to privilege handling	Sandboxed, no direct access to system resources, restricted privileges (no dynamic requests)
User Experience	Browser integration not well-designed, constant headache.	No need to install Wasm, runs natively in the browser. Just works!
Performance	Performance hog, launched slowly, refreshed badly.	Designed to be fast, can be "real time," how Photoshop runs in the browser!
Community	Built and owned by a company, maintains tight control.	Open from the start, future driven by a community with diverse interests.

containers: (**1- how to isolate resource access**) containers should only see their resource/are the only users of their resource (e.g. PIDs, hostnames, UIDs) → each process assigned a "namespace", syscalls only show resources within own namespace, subprocesses inherit namespace; (**2- isolate resource usage**) meter resource usage and enforce hard limits per container → usage counters for groups of processes (cgroups) (rate limiting for compressible resources (CPU, I/O bandwidth), terminate containers for non-compressible resources (memory/disk space)); (**3- provide efficient, per-container filesystems**) per-container FSs without overhead of a "virtual disk" for each container → layering of FSs (copy on write), read-write ("upper") layer that keeps per-container file changes, read-only ("lower") layer for original files.
[*OS: provides isolated environments for each service, fast startup, and easy scaling across distributed nodes*]
[*hardware: single physical server host multiple fully isolated OS, each running its own applications (point of VM consolidation), get better CPU/memory util, while maintaining strong isolation between workloads*]

CPU virtualization: even without OSV, instructions handled differently: privileged instructions (from user mode) emulated by VMM (trap to hypv, full control), non-privileged ran directly on native CPU.

memory virtualization: OS assumes full control over memory; VMM partitions memory among VMs! must assign hardware pages to VMs, control mapping for isolation (can't allow OS to map any virtual→hardware)

application virtual.: app.java [compiles to] Java bytecode [runs in] Java VM [portable across] hardware [*encapsulates app and dependencies (isolates from underlying OS) so it runs consistently across platforms*]

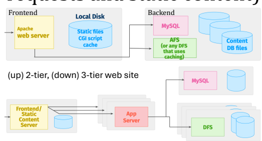
kubernetes (k8s): not virtual., makes container orchestration tractable (load-bal, horizontal scaling, etc)

WebAssembly [ran on Wasm engine] lightweight, fast-starting, and cross-platform execution environment

1-tier: all computation/user requests/data storage compete for same machine resources (CPU, memory, disk); all work centralized → latency rises nonlinearly once system nears saturation even if avg util/load = [avg arrival rate] / [avg service rate] < 1; use queue length as heuristic to scale out/shrink scale, hysteresis to avoid wasteful oscillations; no failure isolation: one overloaded component affects entire service.

2-tier: [frontend] more cores, RAM for running CGI, [backend] more disk for storing content; bottleneck is (1) webserver (dynamic content will consume CPU) cycles or (2) DB (all requests go through it, handle both reads/writes, cannot easily scale horizontally; vertical scaling (add more CPU/RAM) hits physical limits and does not solve the underlying concurrency contention); single DB node serializes critical transactions.

3-tier: adds an application server (runs business logic, dynamic page generation) between frontend (HTTP requests and static content) and DB (stores persistent data) → allow indpt scaling, load-distrib, isolation.



(up) 2-tier, (down) 3-tier web site

DNS redirection: configure same domain to resolve to diff IP addresses based on client's geo-region (reduces latency: send users to nearest site), **geo-replication**: deploy complete service replicas at multiple data centers worldwide, each site serve local traffic, improve scalability/responsiveness. data consistency challenges: updates made at one site take time to propagate to others, so global state may temporarily diverge.

monolithic architecture: multiple app servers (incl. all components), faster start, but cascading failure (no isolation), no async release (all coupled, slow release) → **microservice**: each feature is isolated into its own service, RPC API between components → develop and deploy independently, scale components based on demand; failures harder to reason about because dependencies span multiple services (hard to test inter-service behavior), a single downstream service crash or network delay can cascade and affect many others

time uncoupling [deal with volatile env, network partitions]: sender can send msg even if receiver not yet exists (msg stored and used later), **space uncoupling** [deal with changes in nodes: failure/addition] sender can send msg but does not know to whom it is sending nor if more than one, if anyone, will receive the msg.

Kafka: [**producer**] write records (e.g. orders) to brokers, [**consumer**] read record from broker, [**broker**] server run in Kafka cluster (of many brokers on many servers); **records** have key (optional), value, time.

	t-cp	t-un
s-cp	msg passing, remote invoc	email
s-un	IP multicast	publish/subscribe

data categorized into **topics** (stream of records) by feed name, each topic's data split into **partitions** (append-only logs: ordered + immutable) → replicated to **replicas**.

producer chooses which partition to send a record to (typically based on key), write at their own cadence so order of records cannot be guaranteed across partitions; multiple producers can write to the same topic.

• fire-and-forget (not used in production), sync send block on producer.send(), async send: callback function called when there is response from brokers, higher throughput

consumers read data from partition by pulling updates (no push), control their own pace of consumption (allows to design downstream apps for average load, not peak load), track read offsets for all partitions.

consumer group jointly consume same topic (partition = smallest unit of parallelism, 1 partition consumed by at most 1 consumer each time, but same topic can be consumed by different consumer groups): message only ever read by single consumer in group, order of messages guaranteed within partition, **at-least-once**.

failure of consumer (detected by Group Coordinator via heartbeats) trigger rebalance of paritions.

replicas only used to prevent data loss, never read by consumers, never written to by producers; for each partition, Kafka elects one broker as leader (client producer write to leader → replicate record in followers)