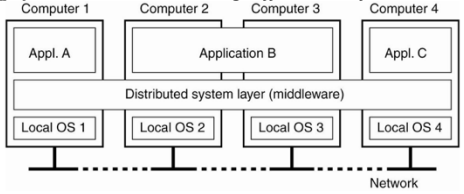
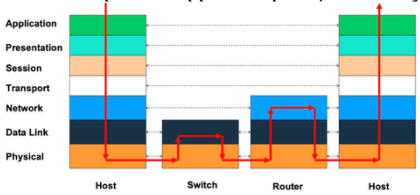
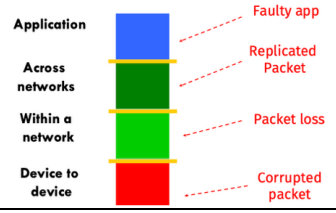
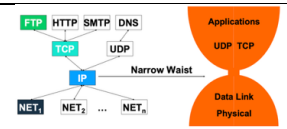
network: nodes and links. sender/receiver use protocol to understand each other (e.g. HTTP for web content). *direct physical connections* do not scale well (either N^2 wires for everyone or all nodes linearly connected to single wire). alternative: *switched network* via forwarding nodes

1. **circuit switching** [fast, stable, predictable perf] [resource ineff]: source establish dedicated conn to dest, forwarding nodes store state about conn, data travels along same circuit path, conn explicitly torn down
2. **packet switching** [general for many apps, efficient/robust resource sharing] [lack isolation (congestion, variable delay)]: source sends info as self-contained packets that have source/dest addr, each packet travels indpt to dest host, switches decide how to forward (store&forward: FIFO queue to buffer packets) [*physical link shared among diff network flows; **bursty** source: more packets appear in queue/delivered*]

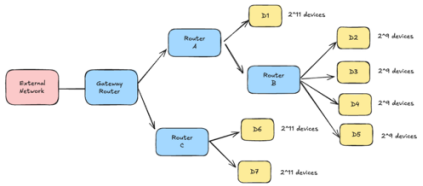




data passed from higher to lower layers via **encapsulation** [*lower layer treats data as raw bytes*]; receiver strips off header by header; **de(multiplexing)**: layer can have multiple protocol impl., how does IP [*network*] know to pass info to TCP and not UDP [*transport*]? Sender adds extra info (demultiplexing field) in IP header [*incl. source/dest addr*], receiver can pick the right protocol in the next layer up

Open Systems Interconnection (OSI) model	**where to handle reliability?**
• Application: the basis of WWW [↓ top-down model] • Presentation: format of data exchanged between peers • Session: provides name space used to tie together potentially different transport streams that are part of a single application • Transport: exchange message from between end hosts • Network: handles routing among nodes [packet-switched] • Data link: collects stream of bits into larger aggregate (frame) • Physical: transmission of raw bits over communication link	
where to place functionality? at the edges	**Internet Protocol Suite**
If you must implement function end-to-end anyway (e.g., because it requires knowledge/help of end-point host or application), don't implement it inside communication system, *unless there's a compelling performance enhancement* [*left*] [***hourglass model***] *Narrow waist facilitates interoperability, but evolution is hard*	

flat addressing: MAC addr (local area networks/LAN), unique 48-bit identifier hardcoded by manufacturer
hierarchical addressing: IP addr (wide area/WAN), dynamically assigned, dotted quad notation (8 bits ea) prefix (network component/CMU) + suffix (host/laptop, unique identifier picked via DHCP)



Gateway Router:
(128.2.0000 0000.0/20, A),
(128.2.0001 0000.0/20, C)
A: (128.2.00000 000.0/21, D1),
(128.2.00001 000.0/21, B)
B: (128.2.0000101 0.0/23, D2),
(128.2.0000110 0.0/23, D3),
(128.2.0000100 0.0/23, D4),
(128.2.0000111 0.0/23, D5)
C: (128.2.00011 000.0/21, D6),
(128.2.00010 000.0/21, D7)

classless interdomain routing: flexible/specified **network** size; forwarding node reduces fwd table size by aggregating fwd table entries with common prefixes in dest (*longest prefix matching algo: router pick entry w longest prefix/"closer to host")*

IP addressed to hosts is best-effort delivery; transport layer process multiplexing on the same host (using ports); **TCP** provides reliable, in-order, two-way byte stream service with **flow control** (avoid sender outruns receivers), **error control** (recover from packet loss, corruption, reordering), **congestion control** (controls the transmit rate of sender) using seqnum + ACK. **stop-and-wait** send next packet after receiving ACK, retransmit if ACK not received before timeout (estimates good timeout based on round trip time RTT), throughput = [1500 bytes packet size] * [8 bits] / [0.1s RTT] * [3 sliding window size] = 360 kbit/s

- [batching to improve throughput] cannot advance sliding window beyond earliest un-ACK'd packet as receiver needs to buffer later packets (to present in-order data delivery to application)

Mutual exclusion (correctness): 1 process in critical section (code accessing shared resource) at 1 time
Progress (efficiency): processes don't wait for available resources/no spin-locks → no wasted resources
Bounded waiting (fairness): No process waits forever for a resource, i.e. a notion of fairness
semaphores: atomic operations P(test), V(increment) (mutex: 0 locked, 1 unlocked/resource available)
conditional vars: NewCond(&ch.mu), Wait() (*called after locking mutex; atomically release mutex & suspend operation; when resume, lock mutex*), Signal() (*no-op if no thread suspended, otherwise wake up (at least) one suspended thread*), Broadcast(); at start of function, ch.mu.Lock() + defer ch.mu.Unlock(); Signal at end
UTC time: comp with receivers sync clocks with timing signals from GPS/broadcast radio stations; **sync. networks**: msgs always arrive with delay at most D; **real networks** async (arb msg delay), unreliable (msg don't alw arrive); **skew**: diff between times on two clocks, **drift rate**: diff per time from some ideal ref clock
Cristian's time sync: time server *S* receives signals from UTC source, process *p* requests time, receive time value *t*, sets its clock to *t* + RTT/2; time by *S*'s clock when m_t arrives is in range [*t+min, t+*RTT-*min*], RTT is round trip time recorded by *p*, *min* is estimated minimum one way delay; accuracy ± (RTT/2 - *min*)
Berkeley [internal sync]: time daemon polls/collect clock values from workers, uses RTTs to estimate workers' values, takes average, send required adjustment; *if master fails, elect new master to take over*
Network Time Protocol [synchronizes clients to UTC, reliability from redundant paths, scalable, authenticates time sources]: hierarchy of time servers (class 1 connected directly to atomic clocks, etc.; class 2 get time from C1/C2 servers; class 3 get time from any server), synchronization via Cristian's + modified to use multiple one-way messages (via UDP) instead of immediate round-trip
Lamport clocks: partial → total order via $L(e) = [\max \#\text{processes}] \times L_i(e) + i$, $A \to B \Rightarrow L(A) < L(B)$
Vector clocks: $V(e) = [c_i]$, c_i = # events in process i that casually precedes e, $A \to B \Leftrightarrow L(A) < L(B)$
Distributed System: no shared memory/msg-based comm; each runs its own local OS; heterogeneity
Distributed mutex requirements [assume total # processes fixed, no proc fails/misbehaves, comm never fails but msg may be reordered] low message overhead, no bottlenecks, tolerate out-of-order messages, allow processes to join protocol or to drop out, tolerate failed processes, tolerate dropped messages

- **Centralized mutual exclusion**: safe, fairness depends on queuing policy, 3 msg/cycle [complete round of protocol: enter/exit of critical section] (request, grant, release); problem if coordinator crash/reboot
- **Bully leader election**: *P* notices leader failed, sends an ELECTION message to all processes with higher numbers (*H*), if no one responds, *P* wins the election and becomes coordinator, otherwise *H* takes over
- **Decentralized**: get majority vote from $m > n/2$ coordinators, reply immediately with GRANT or DENY
  - if < *m* votes, backoff/try later, large # nodes requesting access can affect availability → starvation
  - majority ensures safety, fairness depends on random chance, $2mk + m$ msgs per attempt, *k* retries
- **Totally ordered multicast** [all msg delivered in same order to each *R*]: [TO Lamport clocks- assume all msg from one *S* received in same order + no msg lost] msg to be sent timestamped with *S*'s logical time, msg multicast [incl. to *S*], when msg received, put into local queue, ordered according to timestamp, *R* multicasts Ack, msg delivered to appl only when it is at head of queue + Ack-ed by all involved processes
- **Lamport**: each proc maintain queue of pending req to enter CS; when node wants to enter CS, multicast time-stamped req (*R* adds to own queue), if own req is at head of queue and all replies (timestamp) received, enter CS, upon existing, remove its req from queue and send release msg to every process (*R* remove req from own queue); (n-1) req/reply/release msgs; crash of any process/inefficient
- **Ricart & Agrawala**: node sends timestamped req to other nodes, *R* queues/doesn't reply if it's currently accessing/own msg has lower timestamp, otherwise sends OK; enter CS after receiving *n-1* replies
  - ↑ 2(n-1) msg/cycle + delay, no deadlock/starvation, crash of any process, ↓ lost token/process crash
- **Token Ring**: single token passed around logical ring; [1, ∞) msg/cycle, [0, *n*-1] delay before entry

Low transparency: more complexity, looks like DS (unlike single centralized server), less abstractions
[**PRC**] client procedure calls client stub the normal way → client stub build msg [marshal arg into machine-indpt format], calls local OS → client OS sends msg to remote OS → remote OS gives msg to server stub → server stub unpacks parameters [unmarshals arg, builds stack frame], calls server → server does work, returns result to stub → server stub packs/serializes result in a message, calls local OS → server's OS sends msg to client's OS → client's OS gives msg to client stub → client stub unpacks result and returns it to client
PRCs want to look like LPCs, but achieving transparency is difficult: **memory access** (e.g., address spaces); **failures**: impossible to tell communication vs machine failures; need to make partial failures transparent: [strawman sol] make remote behavior identical to local behavior, every partial failure result in complete failure, abort + reboot entire system → clients block for long periods, system might not recover);
[real sol] break transparency; possible semantics: **exactly-once**, **at least once** (only for idempotent ops), **at most once** (re-send prev reply/not process twice → cache of handled req); **latency** (e.g. perf/overhead)

ACID properties of transactions: **atomicity** (each transaction completes in its entirely, or is aborted (then should have no effect on shared global state), ex: update account balance on multiple servers), **consistency** (each transaction preserves a set of invariants about global state; nature of invariants is system dependent; ex: in a bank system, law of conservation of $$), **isolation** (serializability → each transaction executes as if it were the only one with the ability to read/write shared global state), **durability** (once a transaction has been completed/committed, its effects will persist, even in the presence of failures)

[single server] **wait-for graph**: edge (i, j) if T_i waits for lock held by T_j, cycle ⇔ deadlock; no cycle if locks acquired in same global order → **2-phase locking** (concurrency control via locking to provide strong isolation/serializability): [phase 1] acquire locks as transaction progresses, [phase 2] release locks, [commit]: update changes, release locks, [abort]: throw away changes, release locks; *deadlocks can occur*: txn may not know all locks it needs ahead of time → (1) lock manager builds wait-for graph, on finding a cycle, choose and abort offending transaction, (2) if timeout, find and abort transaction waiting for a lock

[distributed transactions] [partitioned DB offer better scalability; ACID require all servers to agree]

2PC [P1] Coordinator sends Prepare to participants; P write to disk, respond to C [VoteCommit/VoteAbort]; [P2] C checks votes, if all VoteCommit, then write to disk, send DoCommit, else write to disk, send DoAbort; P receive final decision, write Commit/Abort to disk, send Ack back to C, C logs END and ends transaction

Correctness: atomicity/consistent across nodes + commit pt after C writes to stable storage, but **blocking**!

Backward recovery: return incorrect state to some previous correct state (via checkpoint), then continue executing; easier to implement, but expensive. **Forward recovery**: bring system into a correct new state, from which it can then continue to execute; harder to implement, recovery could be faster.

Checkpointing: periodically save system state in reliable storage that can withstand targeted failure, roll back to error-free state in case of failure and re-execute; **consistent snapshot**: for any recorded event *e* somewhere in the snapshot, any events that happened before *e* should also be recorded in the snapshot.

Independent checkpointing: process independently records local state, easy to implement but difficult to recover consistently, prone to cascaded rollbacks. **Chandy-Lamport consistent snapshot algo** [assumes global state changes when event happens, FIFO channels] [event *e* in process *p* is atomic action that may change state of *p* itself and at most one channel *c* (send or receive msg)]. *Snapshot initiation by process P: P records its own state, sends markers to all other processes on outgoing channels, starts recording on its incoming channels. Propagation: when process P_i receives marker message, if receiving marker for the first time, {record its own state, mark that incoming channel as "empty", propagate marker to all other processes on outgoing channels, start recording incoming messages on other channels}, else, stop recording on that incoming channel. Termination: all processes have received a marker on all incoming channels.*

Logging [fault tolerant technique to complement checkpointing] [**write-ahead logging**]: log op to durable storage before system performs op; [recovery] go through logged ops, replay them; fast thanks to sequential I/O; in practice, frequency chkpt too slow → fine-grained logging of operations, log grows unbounded → periodic chkpt to truncate logs; ARIES: WAL + fuzzy/fast chkpt, redo + undo (for uncommitted transactions)

Redundancy [via replication; provide identical service to non-replicated version]: workloads with writes require consistency (ordering of op across mult proc); **strict consistency** require writes instantaneously visible to everyone, **sequential const** require all nodes see all op in some single valid global sequential order, **casual const** require all nodes see potentially causally related (excl. concurrent) writes in same order

Fischer-Lynch-Paterson: no deterministic 1-crash-robust consensus algo exists with async comm

State Machine Replication: replicas should be in sync with each other; **setup**: async comm, no shared clock, no bound on max message delay, some "fairness" assumption: every message arrives eventually.

Single decree Paxos [consensus for a single value] one or more servers propose values, only a single value is chosen; once a value is chosen, it will stay chosen. **Correctness** (safety): only a single value may be chosen (agreement), the chosen value must have been proposed by some node (validity). **Liveness** (termination): some proposed value is eventually chosen, if a value is chosen, servers eventually learn about it (*liveness is not guaranteed, sacrificed in favor of correctness; e.g. dueling proposers lock each other out*).

Fault-tolerance: if less than *f* out of *n* nodes fail, the rest should reach consensus eventually W.H.P.

(1) **Proposers** choose new proposal number *n*, (2) Broadcast Prepare(*n*) to all servers, (4) When responses received from majority: if any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal, (5) Broadcast Accept(*n*, value) to all servers, (7) When responses received from majority: if majority Accept-OK(), value is definitely chosen; could retry if no majority-accept	(3) **Acceptors** respond to Prepare(*n*): if *n* > minProposal then minProposal = *n*, Prepare-OK(acceptedProposal, acceptedValue), else, Prepare-REJECT(), (6) Respond to Accept(*n*, value): if *n* ≥ minProposal then acceptedProposal = minProposal = *n*, acceptedValue = value, Accept-OK(), else Accept-REJECT()

Design principles: (1) multiple proposers and acceptors for fault tolerance, (2) majority quorum for every action (any group decision must have at least 1 overlapping acceptor to arbitrate), (3) uniquely numbered proposal that are globally ordered (once value chosen, stays chosen despite allowing multiple proposals to be chosen (single proposal → unable to learn chosen value under fault)), (4) proposers must discover previously chosen proposals (invariant: once proposal with value *v* is chosen, every higher-numbered proposal proposed by any proposer has the same value *v* → always use highest-numbered accepted proposal out of all discovered proposals from majority quorum), (5) proposers must block unknown future proposals through a "lock" (acceptor no longer accept any proposals with a lower proposal number → prevents conflicting proposals from being accepted in the future)

Distributed file systems: transparent access to remote files; (1) **performance** (overcome network latency 1000x slower than local access, optimizations → consistency problems), (2) **consistency** (maintain coherent views when multiple clients access shared data), (3) **naming** (implement global schemes to locate files across administrative domains, naming schemes affect security boundaries), **security** (protect data integrity and privacy across untrusted networks, mechanisms → impact performance)

	NFS: 4kB block-level caching, efficient for partial access, risk mixed acc within single file	AFS: entire files, wasteful for large files
Design goal	Simple, robust, general-purpose file sharing	Scale to support ~10k users
Key strategy	Stateless server, minimal client-side cache	Aggressive caching, whole file operations
Consistency approach	Client polls server to validate cache	Server callbacks notify clients of changes
Primary use case	General file sharing across heterogeneous systems	University environment with personal workstations

cache coherence: update visibility (when changes become visible), cache validation (discover it's stale)

(1) **broadcast validations**: every time file is modified, S broadcasts invalidation msg to all clients, C must invalidate their cached copy (if any) upon receiving msg → network traffic explosion, wasted msg;

(2) **check-on-use**: every read trigger network round-trip (even if cache valid), eliminates benefit of caching

(2B) **NFS v2** [attribute caching w timeouts]: C caches file + attributes, use cached data w/o checking for 60s, afterwards revalidate with S; eventual consist (after 60s); dirty data buffered locally for up to 30s, written through on file close; (3) **callbacks** (AFS): when C caches file, S records in callback list, promises to callback if file changes/C trusts cache until callback received; complexity costs: S crash loses callback state (must rebuild from Cs), C crash req revalidation of all cached files, network partitions → consist challenges);

[v2] callback invalidates new opens only → **Last Writer Wins**, complete consist/entire file atomic;

(4) **leases**: grants C right to cache data for specified time period, S promises not to allow modifications w/o notif during lease period; bounded consistency (max staleness = lease duration), automatic cleanup (expired leases req no explicit revocation), predictable server state (naturally bounded by active leases), failure resilient (system recovers auto after lease timeout), impl requires loose clock sync (not abs time)

location **transparency** (users don't see physical location, e.g. /home/alice/file.txt) + **independence** (files can move (to another server) without changing names, requires indirection layer/mapping)

Integration problem (if lack loc indp)	
/home/alice	/users/alice ← Inconsistent!
/mnt/server1	[not mounted] ← Accessibility!
/project	/project ← Coordination?

[NFS] **local integration**: each C construct own view, mount remote FS at arb pts, diff C see diff namespaces; **automounter** (instead of static mounting (boot dependency on network/servers)) mount on first access, unmount after timeout, triggered by pathname lookup miss (virtual FS intercepts, triggers mount);

[AFS] **global namespace**: /afs visible identically everywhere, /afs/cell/volume/path, cell = admin domain (institute.edu), can ref each other; volume = management unit (user.alice), can migrate between servers transparently, Volume Location DB tracks locations; natural admin boundaries, req global coord + infra [*path → physical loc*] directory-entry caching for name → inode mappings, [NFS] require getattr() (RPC) per path component (attrib cache helps); [AFS] lookup vol loc cache (updated rarely/vol seldom move, enables transparent migration) → single whole file fetch → callback prevent re-lookups