L1: What is DS?

* DS = collect° computers that com via networks
  - ↳ Inherent distribut°
    - └ necessary
    - └ sufficient
  - ↳ Better reliability
    - └ one node fails ⇒ syst cont work
  - ↳ Better performance (fast)
  - ↳ Solve bigger & cplx problems
    - └ single node cannot handle all data / processing capacity
    - └ efficient solve

Problems → coms Fail
→ nod crash
→ Faults rand/coord
→ security & privacy

Repectives
- Architecture
- Proces
- Coms
- Coord
- Naming
- Consistency & Replicat°
- Fault tolerance
- Security

Design goals

1) Resource sharing
   - achieve effectiveness
   - manages costs
2) Distribut° transparency
   acces / locat° / relocat° / migrat° / replicat° / concurrency / failure
3) Openness
   compon° easily integrated onto other systems
4) Dependability
   - ↳ Availability : readiness usage
   - ↳ Reliability : continuity service
   - ↳ Safety : low catastrophe pros
   - ↳ Maintainability : easy repair
5) Security
   confidentiality / integrity / trust
6) Scalability
   - ↳ Size scalability

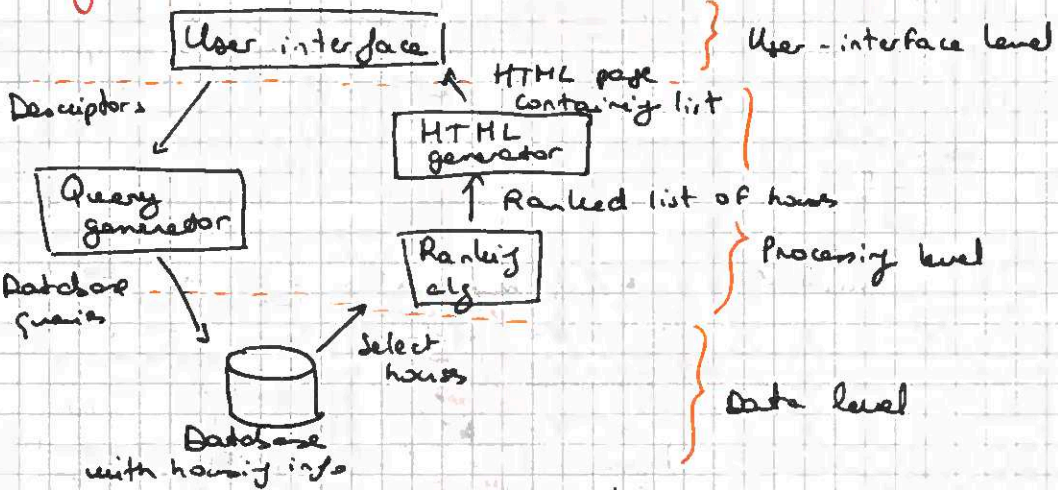
     = n° users / proces / nodes
     - Limited comp / storage / network capacity
     - scale up vs scale out
   - ↳ Geographical scalability → latency : Timeouts (↑)
     - hard LAN to WAN
     - high latency : some appl/protocols may not work
     - unreliable WAN links
   - ↳ Admin ss
     - Conflicting policy concerning usage
       (- Many systems avoid this)

Architecture

Style

- components with well-def interfaces
- how components are connected
- data exchanged btw components
- how components & connectors are jointly configured
  = mechanism mediates coms, coord, coop among components

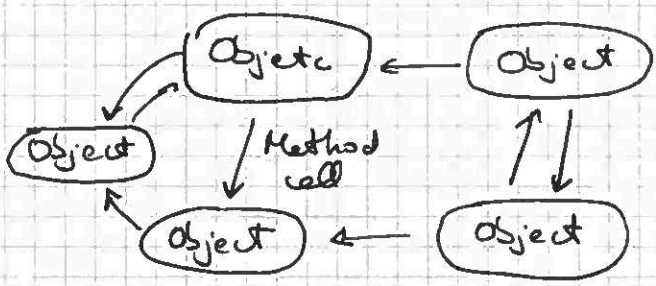
Layered



Service oriented

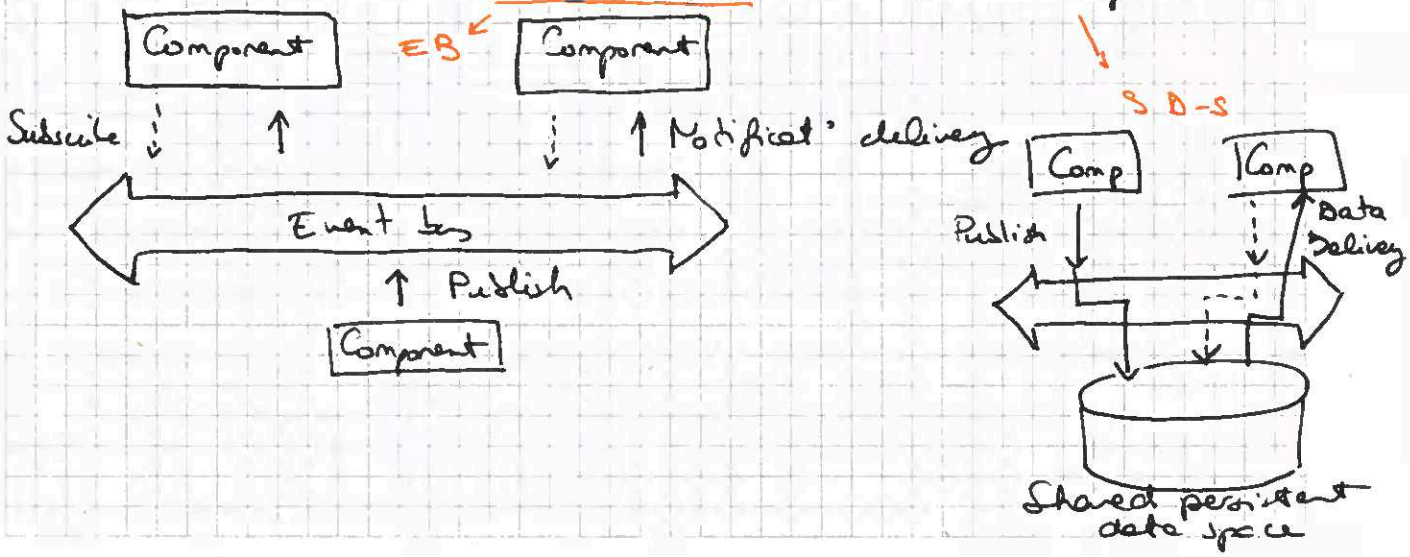
= components, placed dist nodes

object oriented, microservices ÷ larger apps



Publish-Subscribe

syst = collect of autonomous operating proces sharing separat, processing & coord
Event-based / shared data-space

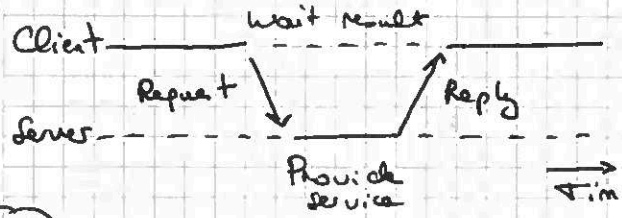


Systems

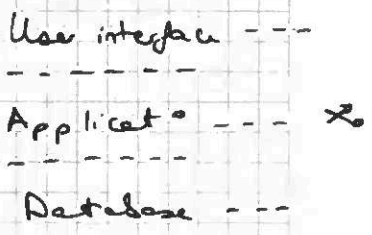
→ Centralized

Client - Server
process using services — process offering services

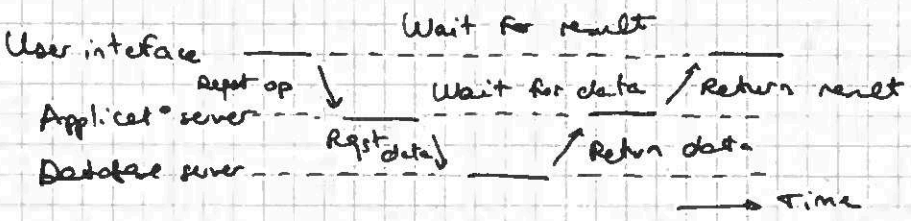
(2 tier)



Multi-tiered



(3 tier)

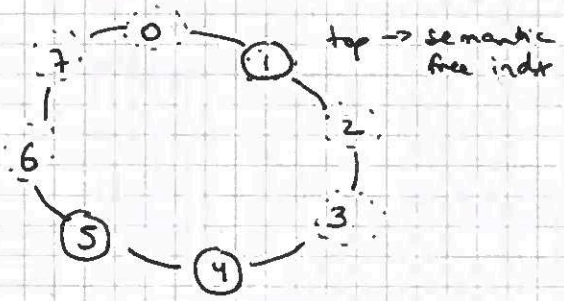


→ Decentralized

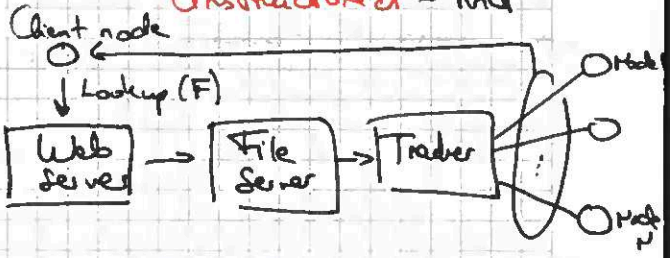
Peer-to-peer

Client server physically split into equiv parts which operate on own share data + load balancing

Structured - Deterministic



Unstructured - Rnd



→ Hybrid

ex: Cloud, Edge, Blockchain

- Cloud: layered pay per use SLAs
- Edge: internet servers edge network + performance + reliability + security & privacy
- tricky

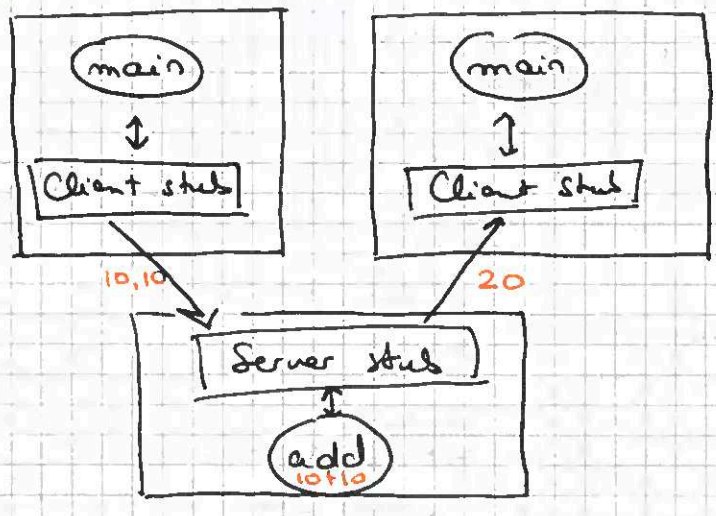
L2 : Remote Procedure Calls

IBM

- RPC => allow remote services to be called as procedures
  - transparent w.r.t
    - location
    - implementation
    - lang

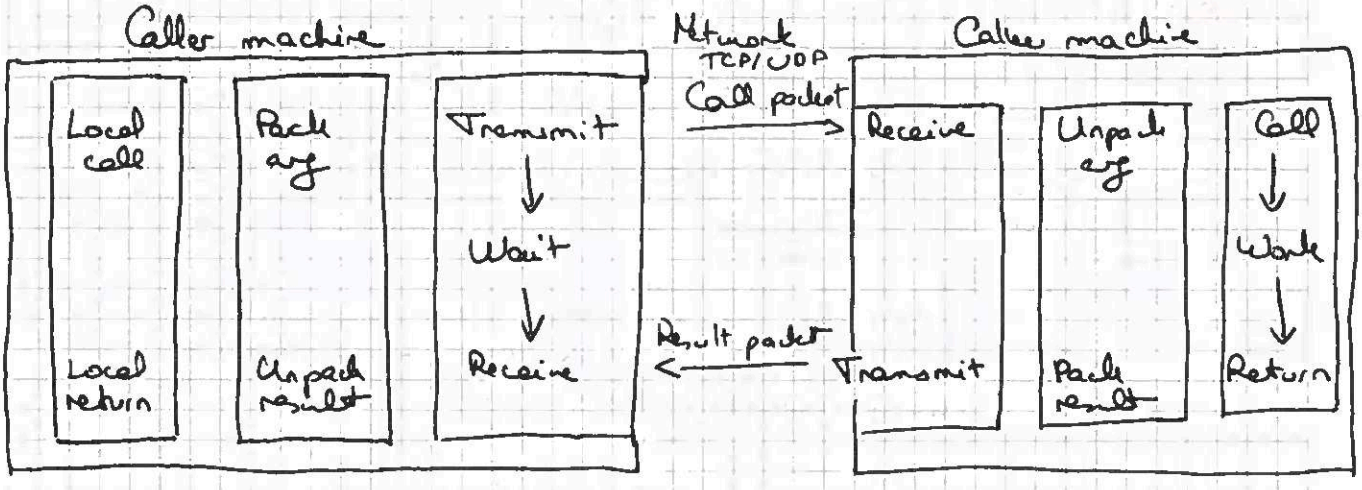
Goal : Make dist comp look like centralized comp

(≈) Programs can call procedures on other machines



Challenges

- separate caller & callee address space
- machines may be different
- thousands of procedures
- client & server may fail (indep)



How do we handle scaling?

- Hide coms latency < asynchronous coms (spot - reply behavior) / delegate work
- Partitioning & Distribut° < split data & spread / decentralized ~ cache data
- Replicat° : drawback consistency

L3: System Models

IBM

Dependability requirement = Availability / Reliability / Safety / Maintainability : continuity service

- MTTF = Mean time to failure ↑
- MTTR = ... repair ↓
- MTBF = Mean time btw failure ↑

Availability = $\frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$

- Failure = syst where component(s) not working
- Error = part component that can lead to failure
- Fault = cause of error

Fault tolerance = syst continues to work under faults — transient / intermittent / permanent

Two general's problem NO SOLUT°

only able to com with each other by sending messages through enemy territory

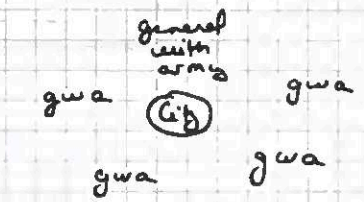
: no guarantee delivery
: ∞ acknowledgement chain

→ probabilistic agreement
→ retry mechanisms
→ multiple channels

Bizantine general's problem

3F + 1 nodes tolerate F malicious nodes

$\frac{3F+1}{\text{total}} - \frac{F}{\text{bad}} = \frac{2F+1}{\text{majority good}}$



friends must agree plan
don't know bad

bad may collude

Network Behaviour

Arbitrary : malicious adversary may eavesdrop, modify, ...
↓
Fair loss : message may be lost, duplicated, re-ordered, ...
↓
Reliable : message received if sent, m can be re-ordered, ...

Halting Fail

- stop : crash failure, reliably detects failure
- noisy : eventually reliably detect failure, noisy behavior] multiple fail resp
- silent : omiss° or crash failure (client won't know what's wrong)
- safe : benign failures (no harm)
- arbitrary (byzantine) : with malicious failures

↓ other° failu

Timing assumpt°s

2 delivery times

→ Synchronous : proces execut° speeds ✓ bounded
can detect omiss° & timing failures

→ Asynchronous : no time assumpt°s
cannot reliably detect failures

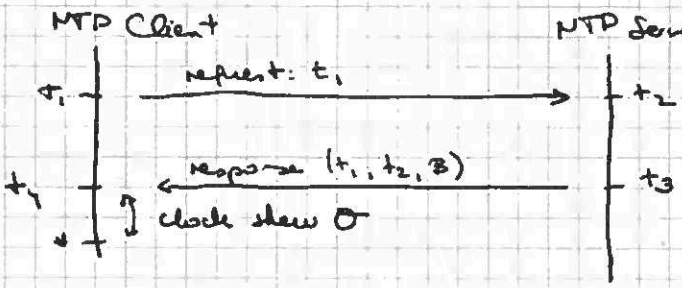
→ Partially synchronous :

L4: Ordering events

Clocks (computers) = quartz oscillators → drift

* Clock skew: difference btw 2 comp clocks (separate)

⇒ Synchronisat° – External: external server bounded by D : $|C(i) - S| < D$
– Internal: $|C(i) - C(j)| < D$ = bound error



Round trip time $RTT = (t_4 - t_1) + (t_3 - t_2)$

$\theta = t_3 + \frac{RTT}{2} - t_4$

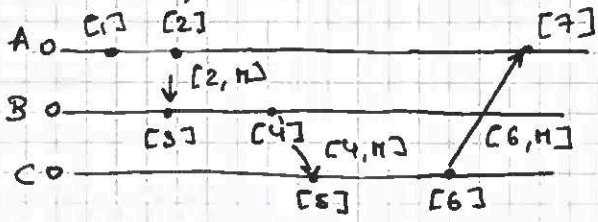
Synchro issues
Time relativity
Variable com latency ⇒ Absolute time NOT A GOOD IDEA
Causality vs time
Faults

Logical Clocks order events (causality)

Rules → each process maintains counter val
→ each process increments counter when send / execute
→ send message event counted (timestamp)
→ ~~recv~~ receive message event +1 (local counter

max (local counter, message counter (timestamp)) + 1

• $a \rightarrow b \Rightarrow$ val counter (a) < val (b)] Lamport* clocks do not guarantee ordered or unrel concurrent events



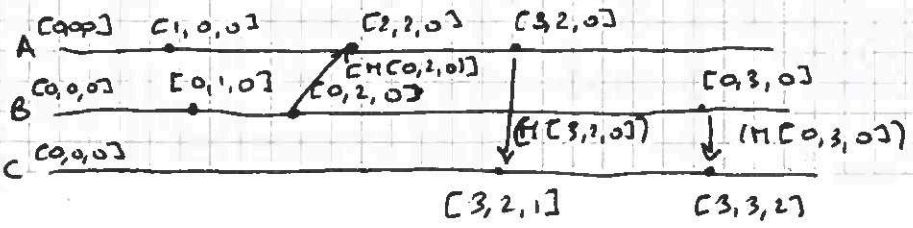
* L clock = single scalar val to track causal ordering

Vector clocks maintain n vals for n nodes in syst. Vector stamp of an event

$v(a) = \langle t_1, t_2, \dots t_n \rangle$
t_i n° events on node N_i
each node has current timestamp T

Rules → initialisat° $T = \langle 0, 0, \dots \rangle$
→ event occurs $T[i] = T[i] + 1$ ~~send (T, M)~~
→ send $T[i] = T[i] + 1$ & send (T, M)

→ receive $T[j] = T[j] + 1$
$T[k] = \max(T[k], T'[k])$
$\forall k \neq j,\ k \in [1, \dots, n]$

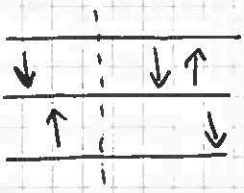


* Global snapshot = Global state, comprise → indiv state of each process
→ indiv state each comm channel

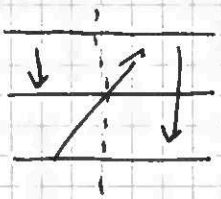
Require
↳ each process must record self
↳ record content comm channels
↳ no pause system or interference with normal flow
↳ collect in distributed manner

- support checkpointing
- garbage collect
- deadlock detect
- terminat computat

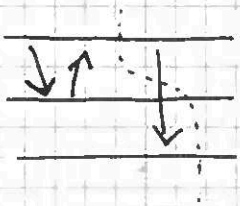
Consistent cut — receiver include ∴ so should sender



~~events not rec~~ ✓



✓



X : events not causally related
(consistent events should be in same cut)

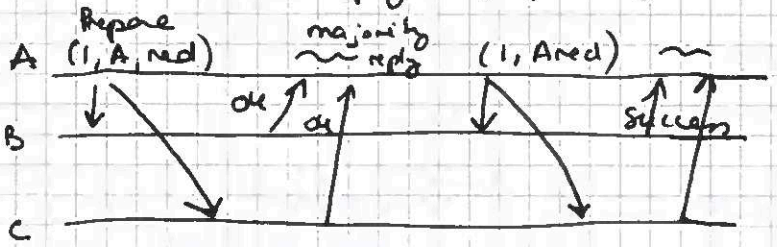
Paxos — agree each client op

Prepare
1. Propose val
2. Acceptor: ok as long as $n > n_h$

Accept
3. Proposer got majority → here is the val
4. Reply acceptors to proposer ✓, otherwise try again (terminate)

Proposer: proposal n

Acceptor: if $n > n_h$
- $n_h = n$
- if no prior proposed accepted
  - reply <promise, n, NULL>
- else
  - reply <promise, n, (n_a, v_a)>
- else
  - reply <prepare failed>

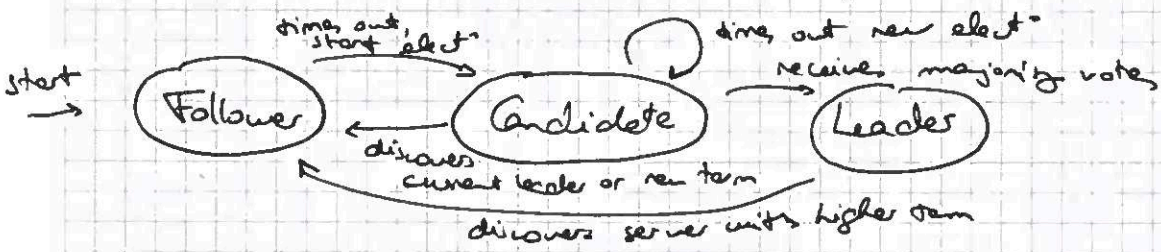


Priority: safety > liveness: not guaranteed, e.g.: competing proposers
- safety: only one val chosen, val remains consistent
- liveness: system will eventually make progress

Raft — alt to Paxos, design simpler: optimized for log appends; agree on new leader

Roles
- Follower - passive: respond reqs
- Candidate: used to elect new leader
- Leader - handle all client reqs

- Leader log: truth
  - keep, track all entries
  - update, F
    - clean up
      - missing entries
      - extra uncommitted
      - repeated
      - ...



- terms act as logical clock
- each node stores current n° increases monotonically
- current terms exchanged while normal ops
- reqs with stale n° rejected

L6 : Mutual excl° & Concurrency

↓
non det behaviour

* Critical sect° = uninterruptable group of code (ensures atomicity)
* Race conditions = error dep on timing

Mutual excl° —ensure→ Critical sect° executes uninterrupted

Deadlock condit°s — eliminate 1 ⇒ fix

ME HW NP CW

- Mutual excl°: at least one resource in "non-shareable" mode
- Hold & wait: process holding resources at least one resource waiting to acquire additional
- No preempt°: resource cannot be forcibly removed from a process holding them
- circular wait: circular chain of procs, where each proc holds a resource that the next proc in chain is waiting for

Consistency ⇒ strong operat°s on share data are sync without sync ops

Causal concurrent write may be seen diff order on diff machines

FIFO writes performed by a single proc are seen by all other procs in order they were issued

P1 W(x)a

P2 R(x)a W(x)b

P3 R(x)b R(x)a] ! violates

P4 R(x)a R(x)b

P1 W(x)a

P2 R(x)a w(x)b w(x)c

P3 R(x)b R(x)a R(x)c

P4 R(x)a R(x)b R(x)c ✓

wouldn't work if R(x)c before R(x)b

P1 W(x)a

P2 W(x)b ✓

P3 R(x)b R(x)a

P4 R(x)a R(x)b

- STRICT (strong) absolute time ordering of all shared accesses matters
- SEQUENTIAL (strong) all procs see shared accesses in same order
- LINEARIZABILITY (strong) sequential consistency + ops ordered according to global time
- CAUSAL (strong) all procs see causally related shared accesses in same order
- FIFO (strong) all procs see writes from each other in order they were issued

⇒ Weak = shared data can be consistent only after synchro done (release/acq)
Weaker model = more scalable

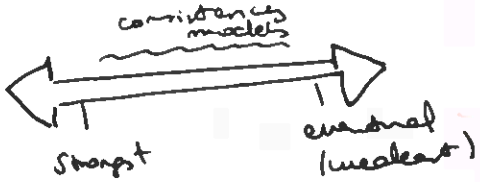
L7: Replicat°

transact° properties
- Atomicity - all or nothing (commit / abort)
- Consistency - follow rules
- Isolat° - concurrent transact°s don't interfere with each other
- D Persistence - remember everything (commit → permanent change)

Replicat° ← availability
- ↑ scaling
- ↑ performance
- ↑ fault tolerance

Consistency Models
- Data Centric: defined consistency experienced by all clients system wide
- Client Centric: defined consistency of data stored only from one client's perspective (diff clients may see diff data)

→ L8: Client centric consistency models



6 Guarantees consistency models

1) Read my writes: client will always see effect of its own prev write
2) Monotonic reads (indiv cons): no going back in time
3) Bounded staleness (incr cons): read guaranteed to return data no older than predefined window (temporal consistency)
4) Consistent Prefix (temporal cons): client will see updates in correct order
5) Eventual consistency (causal cons): no updates → all replicas will eventually converge
6) Strong consistency (system wide cons / global cons): all clients see same data simultaneously after write complete

Strong - see all prev writes
2-5

Eventual - see subset prev writes
0-0, 0-1, 0-2, 0-3, 0-4, 0-5,
1-0, 1-1, 1-2, 1-3, 1-4, 1-5,
2-0, 2-1, 2-2, 2-3, 2-4, 2-5

Consistent prefix - see initial seq of writes
0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5

Bounded staleness - at most one incr out of date
- ← how out of sync U can tolerate

2-3, 2-4, 2-5

Monotonic reads - once client reads val all subsequent reads will reflect same or more recent vals, no "time travel" ← if we first read something older we jump back

1-3, 1-3, 1-4, 1-5, 2-3, 2-4, 2-5
(= → jump back)

Read my writes - client will always see effects of its own writes & reads
writes: 2-5
reads: 0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5 - every opt

Trade-offs

	Consistency	Performance	Availability
S	Excellent	Poor	Poor
E	Poor	Excellent	Excellent
CP	okay	Good	Excellent
BS	Good	okay	Poor
MR	okay	Good	Good
RMW	okay	okay	okay

L9: Network FS

* File = array persistent bytes that can be read/written
* File System = • collect° files
  • part OS manages files

Name → access
↳ inode n° (unix id)
↳ path
↳ file descriptor

* Inode = file that stores data & metadata of a file (in memory rep of file)

Multilevel index

* Directory = specific file that contains other files
  → stores entries in data blocks
  → distinguished from other files via inode

* Superblock = starting point that stores important info about file syst (usually 1st block)
  ↳ n° blocks
  ↳ n° inodes
  ↳ block size

File System Types

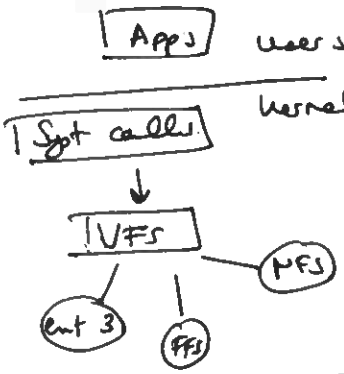
NFS pull

Local ↙ ↘ Network

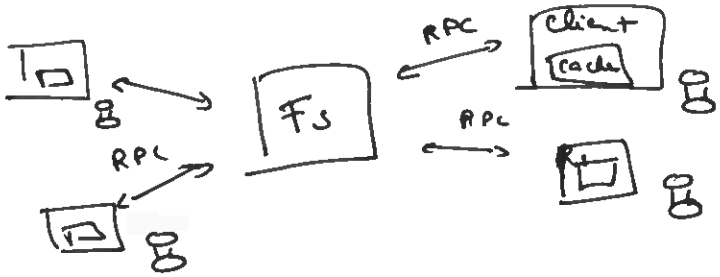
process on same machine as access shared files

process on different machine from access stored files
• fast & simple crash recovery
• transparent access = can't tell remote
• reasonable performance (scaling)

Virtual FS
virtual abstract° — superblocks
— inodes
— file

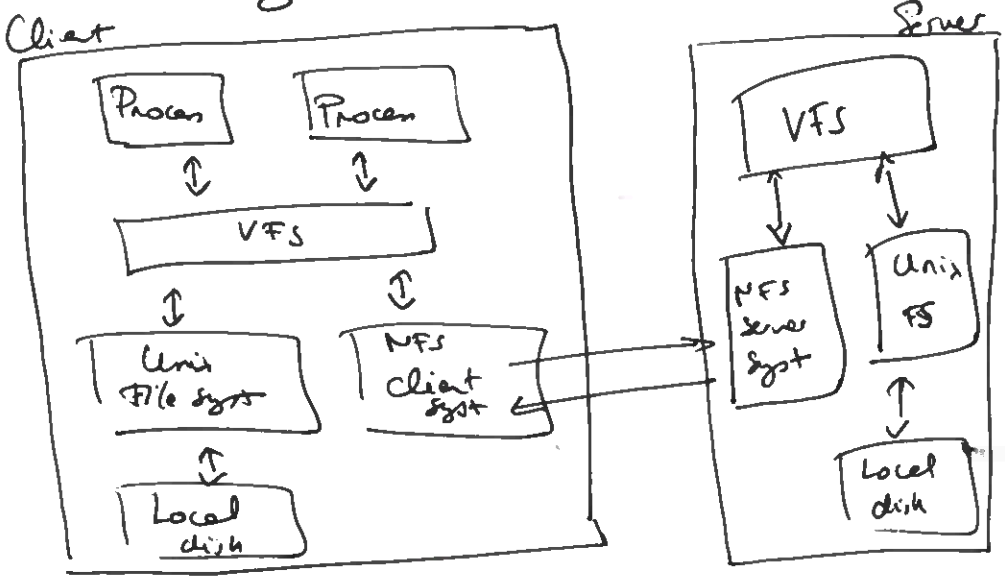


Network FS ~ protocol : stateless
(NFSv2) every req from client completely describes required op
• no client state maintained
• servers can keep state for performance but not correctness
• crash/reboot ⇒ no correctness problem, just slower performance



Stateless Protocol
+
File Handle
+
Client logic

Client ← file handle (returned by server)
↖ current offset



* Idempotent op = same result regardless n° time called

(ex) read, pwrite
↑ write to specific offset of file (pwrite°)

Non-Idemp (x) append, mkdir, rmdir, write

p-print
mk-make
rm-remove
buffer - temp mem storage
caching - comp in main mem

* Caching = temp storage for freq accessed data in fast access storage

NFS problems

- Server crashes
  - ↳ buffer write to improve performance
  - ↳ write might be aknowledged before push

→ solut° → don't use server write buffer: persist data to disk before aknowledging write ⇒ slow
→ use persistent memory ⇒ more expensive

- Caching → update visibility
  - ∴ server doesn't have latest vers°
  - ∴ clients may see old vers°

solut° NFS ⇒ . flush on close (or other time ⇒ problem flush is not atomic (one block at a time) two clients flush at once ⇒ mixed data

- Stale Caching: clients still have stale data they shouldn't read

solut° ⇒ clients recheck in cached copy up to date = STAT reqst

(STAT) gets last modif time stamp t_t for file

↳ problem
overloaded server reqst: 90 %
↳ Rx set to every 3s

AFS (push)

Goals = more reasonable semantics for concurrent file access
improved scalability
willing to sacrifice staleness

- whole file cached (instead of 8kB)
- last writer wins

State cache → Full state protocol
server tell client when data overwritten
server remembers which clients have file open right now
client verifies callbacks when open file (not every read)

Client crash => evict everything cache + recheck entries
avoid re-reading old data

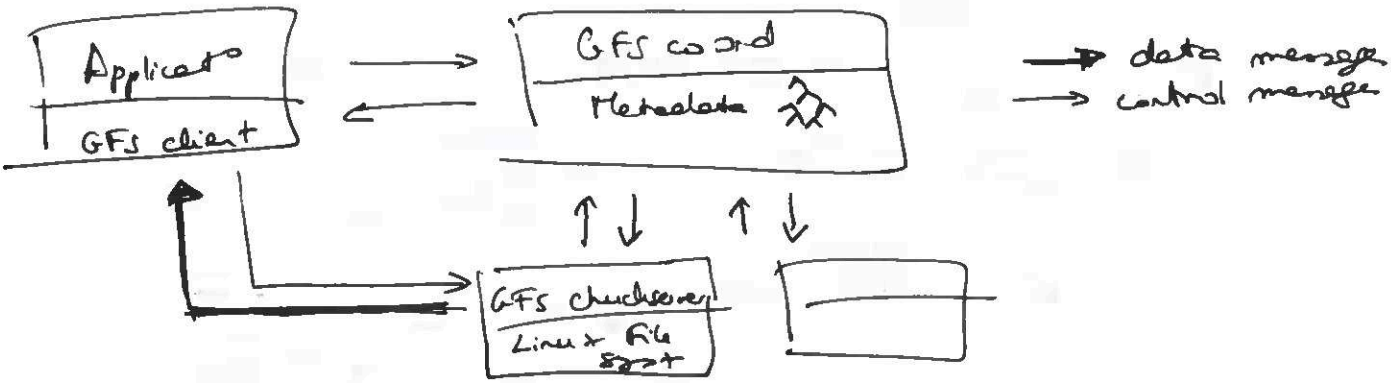
Problems
server => tells client recheck data (it loses track who open file)

L10 : GFs

GFS scalable distributed data intensive applications

Goals
- large storage
- scalable access
- fault tolerance
- transparency
  - location
  - fault

Assumptions
- failure common
- file large
- file access
  - most files appended, not overwritten
  - read writes never done (almost)
  - files mostly read (sequentially)
    - concurrent appends possible
    - reads
      - large streaming
      - small read

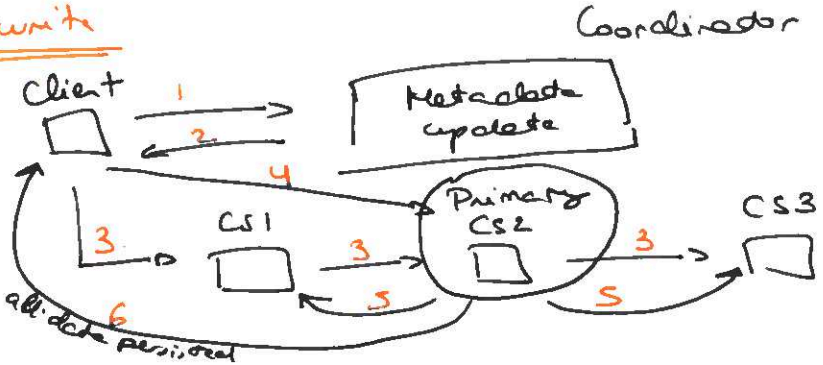


Single coordinator : state replicated on backups
- ↳ Holds all metadata
  - namespace
  - access-control
  - file/chunk locations
- ↳ Manages
  - chunk leases
  - garbage collect° of orphaned chunks
  - chunk migration
- ↳ Fault Tolerance : periodic comm with chunkservers via ♡
  - or log replicated multiple machines

Chunkserver
- stores chunks on local disks
- chunk size = 64 MB
- 32 bit checksum - detect corrupt°
- each chunk 3 replicas
- chunk handle
  - ↳ global unique 64 bit #
  - ↳ assigned coord
  - ↳ read/write req° specify handle

Apps must cope with data inconsistency (failure)

Client write



Metadata consistency

- namespace changes are atomic (all or nothing)
- coordinator uses op log to store critical metadata changes
- log defines timeline → def order concurrent ops
- log stored on coordinator local disk & replicated
- coord recovers file system by replaying op log
- coord only replies to client after log entries safe on local disk & replicas

Data consistency

	Write	Record Append
Serial success	defined	defined
Concurrent success	consistent, undef	int
Failure	inconsistent	inconsistent

def: primary tell client write succeed & no other client write same part

consist: all concurrent writes succeed, readers see same content, mix

inconsistent: primary don't tell client op failed → diff readers see diff content or more

(CA) secondary does not update

Apps

- must deal with inconsistencies
- cannot tolerate duplicates → uniq identif used
- rely on checksum ↑ coordinator detects & removes

Handling Faults

Secondary

- primary retry
- client retry
- coord may remove from list (checksum) CC

=

Primary

- coord remove cc from list
- coord grant lease to any secondary

Coordinator

- replay op log → rebuild state → resume ops
- ask cc what they store
- wait for one lease before granting to any secondary