Communication (RPC)

System properties

- **Availability = the probability that the system is operating correctly at a given instant in time**
  - A(C, t) = fraction of time that the component C has been functioning correctly in the interval [0, t)
- **Reliability = the probability that the system will be available for the entirety of a given interval of time**
  - R(C, t) = probability that component C, which was functioning correctly at time T = 0, functions correctly for the entirety of the interval [0, t)
- **Safety = when not operating correctly** (ie when not available)**, there is a very low probability of the system causing catastrophic events**
  - *Hopefully your definition of operating correctly is such that, when the system is available there is zero probability of the system causing catastrophic events*

System architecture

- An architecture defines: (Hardware and software) Components with well-defined interfaces; the way that components are connected to each other; the data exchanged between components
  - Components can be swapped out for others that comply with the same interface
- **Layered architecture style**
  - Three-tier web applications: Presentation level, Application level, Data level
    - Client-server between client (presentation) and web server (application) but this itself uses a client-server between web server and database server (data) behind the scenes
- **Service-oriented architecture style** — can be object-based, resource-based, microservice-based
  - **Divided by business functions of components, whereas layered divides the system into components based on the technical architecture of the system**
- **Publish-subscribe architecture style** — can be event-based (events are pushed onto subscribers in realtime), or shared-data-based (subscribers pull events by periodic polling)
  - Whether to use shared-data or event-based depends on the distribution of the number of subscribers that events are relevant to
  - **Provides strong separation between processing by components and communication between components**

Communication

- Different components (which eg may be running in parallel) of the same program share a physical address space and so can communicate using global variables or procedure calls (ie local variables)
- Components that are different programs but are running on the same computer can communicate using shared memory (middleware can provide shared virtual address space) or files or signals as they are running under the same OS
- Components that are different programs running on different computers can communicate using shared memory (using middleware that operates over the network) or message passing (e.g. sockets or RPC)
- In message passing, data is copied into a message as the recipient of the communication does not have direct access to our memory
- **RPC (remote procedure call)** adds middleware to sockets to **allows remote services to be treated like local procedures**
- **Parameters and results must be marshalled (packed into a message) by the RPC system**
    - For simplicity, only pass-by-value is possible, not pass-by-reference *(functional programming vindicated)*
    - It is necessary to have a shared form of data representation for RPC messages, as **client and server may use different CPU architecture, operating system etc (and so be little-endian or big-endian, 32 bit or 64 bit numbers etc)** — RPC middleware will translate between these and the shared representation when marshalling and unmarshalling

Failure

- **Failure = the system is not available**
- **Error = one or more components are in a state that could lead to a failure**
- **Fault = the cause of an error**
- **Maintainability = how easily a failed system can be repaired**
- **Byzantine generals theorem: 3f + 1 is always enough total nodes for** the (2f + 1) non-faulty nodes to come to **consensus despite there being f nodes that may fail** (even in the worst case scenario where the f fallible nodes are actively malicious and of unknown identity). Moreover for every f, there exists situations where any fewer total nodes means that the trustworthy nodes will end up with different views of the value being decided

Types of failure

- **Crash failure = halts** — this a permanent state in the absence of human intervention, ie is not a temporary freeze
- **Omission failure = fails to respond to requests**
- Timing failure = response lies outside a specified time interval — eg due to business constraints, beyond a certain level of lag a system might as well have halted
- Response failure = response is incorrect
  - Value failure = value of the response is wrong
  - State-transition failure = deviates from the correct flow of control
- Arbitrary failure = may produce arbitrary responses at arbitrary times

- **Perfect failure detection = node is labelled as faulty if and only if node has actually crashed — not possible!** (because halting problem is undecidable)
  - The best we can do is on a regular basis send heartbeat messages asking the system if it is up, and if no reply are received within a timeout use exponential backoff with an eventual give up. However, it is possible that the system is just temporarily unable to reply

Causes of (RPC) failures

- Client cannot reach server — network or server may be actually down, or the client may be querying the wrong socket due to misconfiguration of the RPC service
- Request message is lost or reply message is lost
- Server crashes after receiving request but before sending response — could crash before actioning message or after completing the operation but before sending the response
- Client crashes after sending request but before receiving response

Recovery from (RPC) failure

- **The client cannot distinguish between: system working but slow; server crashed after accepting request on socket but before actioning the request; server crashed after actioning the request but before replying; server replied but the network failed**
- **Client should repeat its request message** (after an exponential backoff) **if it reaches a timeout without receiving a reply** (or if it was unable to reach the server to lodge the request in the first place)
- **Server should carry out operations idempotently wherever possible as a client may repeat a request for an action it already carried out**
    - It is possible to carry out operations that are not inherently idempotent in a safe manner by having the client assign each operation a unique identifier, as then duplicates can be detected by the server
        - It is unlikely to be practical to atomically commit arbitrary operations in order to achieve exactly-once semantics, but at-most-once semantics are typically (although this is somewhat dependent on the context) preferable to at-least-once
- Assume that the client will be told if itself has been restarted from a crash
    - **Client should store regular checkpoints so it can resume close to where it left off**

Distributed Filesystems

Filesystems: Inodes

- **Each file has exactly one inode number which uniquely identifies it within the given filesystem (at some instant in time, as those of deleted files are eligible for recycling)**
- **Inode number is an index into the inode file (ie an array)** which is stored at a fixed location **and contains pointers to each block (files are fragmented into blocks of fixed size) of the file** — for large files indirect pointers are used as each slot in the inode file has fixed size so can only fit so many pointers
- **A directory is a file that stores (pointers to) files — a directory is a map from filenames (including immediate subdirectories) to inode numbers**

Filesystems: Abstraction

- **A virtual file system (VFS) is an abstraction layer provided by the operating system. The same system calls (provided by a VFS) can be used to interact with a variety of different local and network file systems** (different implementations of the same interface of the system calls (typically implementations are provided as OS kernel modules))
- **Mounting allows multiple filesystems to share one logical namespace (directory hierarchy with a single root)** — each filesystem is assigned a directory to live inside
- **`open` in C returns a file descriptor which indirectly stores the inode number and the offset (how many bytes into the file the process' read pointer is currently)**
    - **The C file descriptor is an `int` — it is an index into an array in a file descriptor table stored for the process by the operating system**
    - Following the pointers of each directory in a path to find the inode number of the file we actually want (so called path traversal process) is expensive — we do this once when we `open` and then cache the inode number through the file file descriptor table and pass the file descriptor on each call to `read` or `write`

Caching in distributed filesystems

- **A distributed filesystem can cache data in: server main memory, client main memory, client disk**
- To reduce network traffic, clients buffer up writes locally before flushing across the network — taking it to the extreme (AFS will), wait for `close`
- What if the client reads from a stale cache? **Client can ask server for the last modified timestamp when they `open` the file and compare it to what it was when the populated their cache**
    - Even though each a small amount of data, this timestamp traffic can add up to a lot of load on the server if there are a lot of active clients — **NFS takes this timestamp approach but allows bounded staleness**

NFS

- Ordinary RPC of the UNIX system calls (VFS) (`open`, `read`, `write`, `close`) is not sufficient for distributed filesystems as **if the server crashes** (loses the contents of main memory) **then it won't have the entry in its file descriptor table anymore** (as it has lost the state created by the open request) and so won't be able to convert the file descriptor (index into the file descriptor array) to an inode number to be able to respond to retransmitted `reads`/`writes`

- *Network File System (NFS) was developed by Sun Microsystems*
- **NFS is stateless: Clients send the file handle (<volume ID, inode number, generation number>) and current `offset` to the server with the request — file handle directly contains the inode number**
  - **Generation number is incremented by the client** (in their local virtual filesystem implementation) **each time an inode number is reused** — solves the difficulty that inode number of a deleted file is eligible to be reused
- **NFS can lead to overlapping writes** as the flush happens one block at a time but the true atom is the files not the blocks that make up the file
  - Applications using NFS must implement their own serialization (locking)

AFS

- *Andrew File System (AFS) (named after Andrew Carnegie and Andrew Mellon of CMU) is popular in academia (eg DICE), whereas the legacy NFS is popular in industry*
- **AFS is similar to NFS but works at file-level instead of block-level** — NFS caches blocks rather than whole files, however most clients end up using most of each file they open
    - **AFS** never has the client communicate with the server for read/write, only to **pull the entire file into local cache at open** (unless the client has it cached locally already) **and flush the entire file at close** (if any writes were made)
    - AFS has mandatory (whole-file) caching, whereas **in NFS it was down to the client to decide which blocks to cache and when to flush written blocks**
- **As files are now the atoms, concurrent writes result in only one write taking effect (the final `close` to occur)** which is slightly preferable to the interleaving of blocks in NFS
- **AFS is stateful as (unlike NFS' timestamp comparison approach) cache invalidation is handled by clients registering a callback with the server to be notified if someone writes to a file they have open — the client can then `close` and reopen the file if they are a reader and want to see the latest updates**, and decides for themself what to do with their write if they are a writer
    - As NFS is stateless it didn't matter if the server restarted from a crash, whereas for AFS if the client gets a timeout they should (in case the server crashed) re-register the callback when they re-transmit

GFS: Coordinator

- *Google File System (GFS) is implemented as a library instead of as an OS module*
- **GFS splits files into chunks** (of size much larger than filesystem blocks, eg 64MB)

- **There is a single coordinator** (can and should have replicas to failover to, but only one coordinator should be active at any point in time)
- **Holds all the metadata, and does so in main memory**
- **Maintains a log which it can replay to recover metadata after failure** — only acknowledges a client request once the corresponding log update has been committed to local disk and has been propagated to replicas
- **Manages the chunk-servers — chunks are distributed across chunk-servers so that each chunk is replicated** across (at least) 3 chunk-servers
  - Chunk version numbers allow the coordinator to detect replicas with stale chunks
  - Chunk checksums allow the coordinator to detect replicas with corrupted chunks
- **Converts pairs of filename and chunk-index from client requests to a pair of chunk-handle and chunk locations (which chunk-servers are storing that chunk)**

GFS: Clients and chunk-servers

- C `write` is idempotent as the offset is specified, so repetition overwrites with the same data, whereas `append` is non-idempotent for obvious reasons
- **GFS supports parallel appends** (in fact indeed these are the type of write it was designed around) **— client sends data with no guarantee in advance of exactly what offset it will end up at, only that it will end up somewhere in the file (and all together)** — efforts are made to make whatever offset it ends up at it be the same in all replicas, but this aspect is not infallible
    - **Writes that are larger than the chunk size must be broken up, and so could become separated from each other in the file!** (unless clients implement their own locking)
- **Clients only cache metadata (eg chunk-locations) not data**
    - There is no caching of data by anyone, as Google's files are too large
- **Client can read from and write to whichever chunk-server they want from the list of chunk-locations (ie primary and replicas which have that chunk) —** if a replica receives a write, it will forward the request to the primary for that chunk which will in turn send requests to the replicas — the primary writes first then tells the replicas which offset it wrote it to

Coordination

Physical clocks

- **Clock drift = rate of deviation of a clock from true time** *(does true time actually exist if no clock can measure it?..)*
- **Clock skew = difference between two clocks at a point in time**
- Drift is analogous to relative velocity between vehicles and skew is analogous to distance between vehicles
- NTP: Clocks in stratum i+1 synchronise with stratum i and with each other
  - Stratum 0 = atomic clocks
  - Stratum 1 = primary time servers
- Clock synchronisation (eg NTP) allows us to limit the skew of a clock
  - Error cannot be completely removed, as network communication during the synchronization introduces uncertainty
  - Drift will continue, causing skew to accumulate again

Logical clocks

- Very often we only care about maintaining the ordering of operations rather than the literal accuracy of their timestamps — we can use logical clocks instead of physical clocks, and thus not have to worry about drift
- *Einstein's theory of relativity tells us that no global physical clock can exist, however it also tells us that causality is consistent for all observers*
- **The happens before relation (which we denote →) is an irreflexive partial order over events — when we say happens before we mean that causality requires "happens before"**, some events which "happens before" in physical time will not be comparable
- **We write a || b iff a and b occur concurrently iff neither a → b nor b → a**. Concurrent events are not necessarily concurrent in physical time, however **for the purposes of causality their ordering is irrelevant**
- **We say that a strict total order ≺ is a causal order iff (a → b ⇒ a ≺ b)**

Logical clocks: Lamport clock

- **Lamport clock: Each process maintains a local `counter`**
- **Initialize:**
  `counter = 0`
- **On carrying out a local operation:**
  `counter++`
- **On sending a message:**
  `counter++`
- **On receiving a message:**
  `counter = max(counter, message.counter)`
  `counter++`

- counter can be seen as a global state, which each process has a local view of which is kept sufficiently in sync for how it will be used

Logical clocks: Lamport clock

Proposition: **counter(a) < counter(b) is necessary but not sufficient for a → b.**
That is that:

i) If a → b, then counter(a) < counter(b) — this is what we wanted
ii) If counter(a) < counter(b), then it is not necessarily the case that a → b — this is not ideal

Proof:

i) awlog (use induction) that there are no intermediate causes between a and b. Then, counter(b) = max(local_counter, counter(a)) + 1 > counter(a)
ii) If a || b, then no synchronization of counters occurs in the time between a and b, and so the relationship between the counters can be arbitrary

Corollary: **All a → b can be correctly classified, but some a || b could be misclassified as a → b**

Logical clocks: Vector clock

- Vector clock: Lamport clock but counter is a vector of counters (an entry for each process, but still every process has a local view of the entire global counter)
- **Each process initializing:**
  ```
  counter = new Array(num_processes)
  ```
- **Process `i` on carrying out a local operation:**
  ```
  counter[i]++
  ```
- **Process `i` on sending a message:**
  ```
  counter[i]++
  ```
- **Process `i` on receiving a message:**
  ```
  for j in 0 until num_processes
      counter[j] = max(counter[j], message.counter[j])
  counter[i]++
  ```

Logical clocks: Vector clock

- Proposition:
  i) **a → b iff ∀ i `counter`(a)[i] ≤ `counter`(b)[i] and**
  ∃ j: `counter`(a)[j] < `counter`(b)[j] — note unstrict then strict inequality
  ii) **a || b iff ∃ i: `counter`(a)[i] > `counter`(b)[i] and**
  ∃ j: `counter`(a)[j] < `counter`(b)[j] — note inequalities are strict
  Proof sketch:
  i) Trivially modify the proof for Lamport clocks
  ii) Follows from i) and a || b iff ¬(a < b) ∧ ¬(b < a)

Global snapshots

- **A global snapshot = a record of the state of every process and the state of every communication channel**
- **A cut C is a set of cut events (each of which is a snapshot of the state of a node) where each node has exactly one cut event** (and thus |C| = n)
- **A cut C is a consistent cut iff for every node i (with corresponding cut event f); for every event e such that e → f; e was captured by C (let e' be the cut event in C of the node on which e occured, e → e') also**
- **We call a global snapshot consistent iff it corresponds to a consistent cut**

Global snapshots: Chandy-Lamport algorithm

Let P_i be the node that initiated the snapshot:

1. **P_i records it own state**
2. **P_i sends "marker" message on outgoing channels to all other nodes and starts recording the incoming messages on the incoming channel of each of these nodes**

Let P_j be a node that has received a marker message and has not received a previous marker message for this snapshot:

1. **P_j records its own state**
2. **P_j records the state of the channel it received the marker message on as empty**
3. **P_j sends "marker" message on outgoing channels to all nodes other than itself (note includes sending a marker to the node that sent a marker to it) and starts recording the incoming messages on the incoming channel of each of these nodes**

Let P_j be a node that has received a marker message and has already sent marker messages itself:

1. **Stop recording the channel the marker message was received on, and record the state of this channel as the contents of the recording**

- **Eventually, every process will have recorded its own state and every channel will have had its state recorded by exactly one process (that which it was incoming to)**. A central authority (eg the node that initiated the snapshot) can ask each process for these and together they form a consistent global snapshot

Leadership election

- **Bully algorithm: Nodes are numbered sequentially. If a node k notices that the leader is unresponsive, send a message to all processes with higher IDs. If none of these respond, k takes over the leadership. If a node responds, it will go on to send its own messages and so on**

- **Ring algorithm: Nodes are numbered sequentially. If a node k notices that the leader is unresponsive, send a message [k] to node (k + 1) mod n. If they do not respond, send a message to node (k + 2) mod n and so on. Once a node l that does respond is found, have it try l+1 mod n etc with message [k, l].**
  The message will eventually make its way back to k by one of its predecessors and will now contain all the IDs of nodes which are responsive. Finally, k sends to each element of the message, the maximum element of the message, and that node takes up the leadership

Consensus

Foundations

- **Safety property = a given bad event will never happen — correctness**
  - As this is a ∀, it's violation can be witnessed with a counterexample
- **Liveness property = a given desirable event will happen eventually — forward progress**
  - As this is a ∃, demonstrating it's violation requires a general proof
- Termination = every non-faulty node eventually decides — this is a liveness property
- Agreement = all non-faulty nodes decide on the same value — this is a safety property
- Validity = the decided value is the choice of at least one node — does not need to be "fair", only "correct" in this very loose sense as our focus is on fault tolerance!

- **Synchronous system = execution speed and message delivery times are bounded** — can detect omission and timing failures — bound converts crash failures into timing failures
- **Asynchronous system = not synchronous system** — cannot reliably detect crash failures
- **It has been proven that valid consensus with agreement cannot be guaranteed to terminate for an asynchronous system even if only a single process has crash faults**
  - Whereas, for the synchronous case, we have already seen such a protocol: the solution to Byzantine generals

Paxos

- **Roles for nodes (can be multiple of simultaneously):**
  - **Proposers: Propose values**
  - **Acceptors: Accept proposed values — ideally all nodes are acceptors**
  - **Learners: Adopt the decision after consensus has been reached** e.g. crashed during consensus but subsequently restarted
- **We assume that messages can be delayed, duplicated, lost, but not corrupted** (or equivalently corrupted messages can always be detected) — being able to detect corrupted (or malicious) messages bypasses the impossibility result for a fully asynchronous fully Byzantine system
- **Paxos requires 2m + 1 total nodes to be able to handle m nodes failing**
- **Paxos consists of a prepare phase followed by an accept phase** — *designed to reflect a parliament negotiating to construct bills that ought to get enough votes, then voting to try to actually pass them*

Paxos: Prepare phase

- Proposer (of value v):
  Generate a proposal number n
  send <prepare, n> to acceptors

- Acceptor upon receiving a prepare message <prepare, n>:
  if it already promised to back a proposer with a larger proposal number then
  reply <prepare-failed> (or just do nothing)
  else if it had already accepted a value v′ in this round (with a proposal number n′) then
  reply <promise, n, (n′, v′)>
  else then
  reply <promise, n, null>

Paxos: Accept phase

- Proposer of value v having received promises from majority of acceptors in prepare phase:

```
if it received any replies of the form <promise, n, (n', v')> then
    Update v to the v' with the largest n'
In any case send <accept, n, v> to acceptors // note it keeps its own n
```

- Acceptors upon receiving a message <accept, n, v>:

```
// nh = largest proposal number it has seen over both the prepare phase and the accept
phase, prior to this point
If n >= nh
    nh = n
    Adopt value v
reply <accepted, nh>
```

  - Learners: By having the proposer periodically retransmit the <accept, n, v> to the acceptors that did not reply, if they eventually come back up they will adopt v and acknowledge that they have with <accepted, n>

- Proposer of <n, v (possibly now v=v')> once it has received acceptances from a majority:

```
if in every <accepted, nh> nh == n then
    Paxos has completed with v as the consensus choice
else then
    do nothing (if there is no other proposer which has actually succeeded, then the
system will eventually timeout and start a new prepare phase)
```

Paxos: Fault tolerance

- **If an acceptor fails: As long as a majority are still up, the algorithm works fine**
- **If a proposer fails: If it did not have the majority backing it, does not matter. If it did, system will timeout and restart, thus finding a new proposer**
- **Paxos is <u>always</u> safe:**
  - If an acceptor adopts a value, it is a proposed value
  - If an acceptor adopts a value, no acceptor adopts a different value
  - If a learner adopts a value, no acceptor adopts a different value
- **Paxos is <u>often</u> live:**
  - <u>If fewer than half of the acceptors fail,</u> a value will eventually be adopted by all the available acceptors
  - If consensus is reached on a value, learners will eventually adopt it

Raft

- Raft is designed to be easier to understand than Paxos while having the same fault tolerance properties
- **Nodes in raft can be (at any instant in time, only one of):**
  - **Follower — replica**
  - **Candidate — up for election to be a leader, once the election concludes will either be a follower or the leader**
  - **Leader — primary**
- *By randomizing the times at which different nodes stand again as candidates, we can eventually break past any no overall majority situations in elections*

Raft: Logical clock

- **If a candidate wins an election, then they become a leader for an arbitrarily length term — terms are identified by consecutive incrementing integers**
- **Term ID acts as a logical clock as it is monotonic — nodes include the current term number that they are aware of in their messages**
- If a node receives a request with a stale term number (smaller than that it is aware of), reject the request (and in doing so tell the node the (larger) term number we are aware of)
- When any node is told its request is stale, it will then update the term number it is aware of (and likely discover a new leader). If a leader or candidate is told its request is stale, it will additionally change role to follower

Raft: Log synchronization

- Raft is designed to create consensus for append-only logs specifically
- **Raft has the safety property that** (assuming not too many nodes die within a single term) **committed log entries are in the same position on every node which they are on**
- **Clients send writes to the leader who makes them as uncommitted writes, then sends its entire log to followers. Followers then overwrite their entire log with that of the leader**
    - *In practice the log synchronization can be carefully optimized. However, conceptually it is safest to see it as a full overwrite, as there are hidden complexities (eg committed entries that are missing from the follower's log must be filled in as committed entries before we append our uncommitted entries)*
- **Leader only tells followers to commit a write once a majority of followers have acknowledged that they updated their log with it as an uncommitted write. It is also only at this point that the leader acknowledges the client's write**
- **Nodes (both leader and followers) discard uncommitted entries at a change over of term**
    - As they will have not received their acknowledgement, clients with uncommitted entries will eventually timeout and retry their request, so their data is not lost
- **In an election, nodes will vote for a node that (to their knowledge) has the most committed entries** — if the leader has not failed, they ought to be reelected, but we could also just not call an election in the first place in this circumstance

Consistency

OS recap: Critical sections

- **Critical section = an atomic group of instructions. That is to say, a section of code which for correctness must be executed without preemption**
- Mutual exclusion = at any given moment the number of threads that are inside a given critical section is at most one
- *Progress = no deadlock. If at least one process wants to enter a critical section and no process is currently in that critical section, then one of the waiting processes will be allowed to enter the critical section*
- *Bounded waiting = no starvation. Every process that ever waits to enter a critical section will eventually be able to do so*

Distributed locks: Coordinator approach

- Assume messages are eventually delivered, and arrive in order. Then, we can elect a leader to manage a queue to manage access to the critical section
- However, this type of solution is never ideal as the leader is a single point of failure — attempting to detect failure of the leader using timeouts risks violating mutual exclusion as it may not have actually failed

Distributed locks: Token ring

- **Token ring algorithm: Nodes are sequentially numbered and pass around a single token** (for each critical section)**. A node can only enter the critical section if it is holding the** (corresponding) **token. If node i has the token, it is allowed to enter the critical section at most once and upon leaving the critical section passes the token to node i+1 mod n**
- **Nodes which are given the token but are not waiting to enter the critical section, just pass it on to node i+1 mod n straight away** — if nobody wants to enter the critical section the token will loop round and round until someone does
- *Token ring intuition: Young children are sat on a carpet in a circle. To prevent them talking over each other, they are only allowed to speak if they are holding a special stuffed toy which they pass around the circle*
- **Issue: What happens if a node crashes while holding the token? As then no other node can ever get the token** *(Suppose a child throws a tantrum while holding the toy, and the school's policies prevent teachers from making physical contact with students. The child might not calm down until hometime, but the teacher doesn't want to get a new toy out in case that child does later calm down and starts talking over a child who has the new one)*
- **What happens if a node crashes but is not holding the token? This can easily be made not a problem by asking for acknowledgement of receipt when passing the token, and passing it to i+2 mod n if i+1 does not acknowledge and so on** *(Suppose a child's neighbour(s) are taking their nap time early. Then they are permitted to throw the toy over the heads of the nappers to reach the nearest awake child)*

Distributed locks: Ricart–Agrawala

- To request a lock:

```
Broadcast <LockName, NodeName, LogicalTimestamp>
Wait for OKs from all nodes
// Execute critical section
Send OK to all queue entries (and thus empty the queue)
```

- On receiving a request for a lock:

```
if in a critical section holding the lock, then
    Add the request to a local queue
else if waiting for the lock in order to enter a critical section then
    if then request has lower timestamp then
        send OK
    else then
        Add the request to a local queue
else then // We are neither in nor waiting for the critical section
    send OK
```

- Issue: What happens if a node crashes? As then other nodes will be waiting forever for its OK

OS recap: Deadlock

- A set of processes is in deadlock iff each process in the set is waiting for an event (usually the release of a resource) that can only be caused by another process in the set and so none of the events can ever occur
    - **Deadlock is a violation of safety** properties
- **The following conditions are each necessary** but (even all together) not sufficient *(unless there is only one instance of every resource involved in the wait)* **for deadlock:**

1. **Mutual exclusion** — The resource instances in question are subject to mutual exclusion
2. **Hold and wait** — There is a process that holds a resource and is waiting to acquire additional resources
3. **No preemption** — The resources being waited on can only be released by the processes holding them and they will only do so once they have finished using them
4. **Circular wait** — There exists a subset $\{P_1', \ldots, P_m'\}$ of processes such that P_1' is waiting for P_2', P_2' is waiting for P_3', …, P_{m-1}' is waiting for P_m', P_m' is waiting for P_0

Preventing deadlock

- **Prevent hold-and-wait:** If we know which collection of locks we will need we can have a **meta-lock that must be acquired before** acquiring any of these locks
  - **Threads** that need disjoint locks that are sometimes needed together **may be unnecessarily serialized**
- **Allow preemption: Can lead to livelock — unlike deadlock which is a safety violation, livelock is a liveness violation**
  - *By introducing randomized delays, we can always eventually break past livelock*
- **We can detect deadlock by: taking a global snapshot, constructing a graph of waits, and detecting cycles** — having detected it we can "do something about it"
  - *Has some false positives if involved resources have multiple instances, but never causes false negatives*

Databases recap: ACID transactions

- **Atomicity: Either all the parts of a transaction or none of the parts of a transaction occur**
- **Consistency: Operations that violate referential integrity or any other constraints are rejected by the DBMS**
  - Because of atomicity, what the transaction has done so far will be rolled back
- **Isolation: The outcome of a transaction is not affected by the fact that there are other transactions being processed concurrently**
- **Durability: Once a transaction has been marked as successful, the changes will persist regardless of what failures occur subsequently**
  - D is closely linked to A but ADCI doesn't spell anything

Concurrent transaction processing

- If we have concurrent transactions (distributed clients) but only one database server, then life is not too tricky
- Isolation can be trivially provided by serializing overlapping transactions, but some transactions can be safely parallelised
- Look before you leap approach: **Use reader-writer lock (can have any number of readers (as long as no writers), but any writer must wait for exclusive access (including no readers)). Wait for necessary locks at start of transaction (and release at end) — risk of deadlock!**
- Forgiveness is easier than permission approach: Always allow transactions to begin. **Check for conflicts before commiting, if there are conflicts abort the transaction — aborts are wasteful**
    - *How to detect conflicts is for a databases course, which this is not*

Distributed transactions

- When data is replicated, we need to commit or abort based on whether the other involved servers are able to commit or need to abort — **atomicity requires that if any one server cannot commit the data, then all servers must abort the transaction**
    - This is like a consensus problem, but we now need all chose commit iff all vote for commit, otherwise (ie iff at least one votes for abort) all chose abort. Whereas in a typical consensus problem, we simply need a majority to adopt a choice made
- One-phase commit: If primary passes consistency, it commits (as per usual) and tells replicas to commit. If any replicas turn out to not be able to commit (eg run out of hardware resources), then it is too late for primary (or other replicas) to abort instead…
- **Two-phase commit: Primary has replicas vote whether to commit or abort, then multicasts what to do accordingly**

Replication

- Replication:
  - Provides fault tolerance
  - Places data geographically closer to more users
  - Spreads load across multiple servers
- Replication requires effort to maintain consistency
- Conflicting operations may be read+write or write+write
- Carrying out conflicting operations in completely the same order across replicas is expensive, so we want to have the minimal ordering requirements necessary

- **Data-centric consistency models = there is a system-wide consistent view on the data**
- **Client-centric consistency models = data is consistent from the perspective of each individual client, but different clients may observe different data**

Replication: Data-centric consistency models

- **Strict consistency = all writes are instantaneously visible to everyone (and thus each read gets the most recent write)**
  - **Requires physical timestamps**, so is physically impossible
- **Sequential consistency = writes appear to every process in the same order <u>(which may not be the order of the writes)</u>**
  - We now **only need logical clocks** instead of physical clocks
- **Linearizability = sequentially consistent and the ordering of the logical clock timestamps of the operations is the same as the order in which the operations appear**
- **Strict consistency ⇒ linearizability ⇒ sequential consistency**

- **Causal consistency = Events are observed in the order of the happens before relation. Clients may observe concurrent writes in different orders to each other (unlike sequential consistency)**
- **FIFO consistency = for each process, all its writes occur in the same order relative to each other on every process. Clients may observe writes from distinct processes in different orders to each other**

Replication: Client-centric consistency models

- **Strong consistency = see all previous writes = all processes see the same data always**
  - Essentially messages are instantaneously transmitted
- **Eventual consistency = Each process** sees a subset (not necessarily shared with other processes) of previous writes at each point but **eventually sees all writes. <u>No guarantee on ordering of writes</u>**

- **Consistent-prefix consistency =** Eventual consistency with the additional constraints that the versions of the data seen are in the correct order and that each version of the data that the process sees did actually exist at some point
  - Essentially **messages can be delayed but neither reordered nor lost**
- **Bounded-staleness consistency = Eventual consistency with the additional constraint that each read sees data that was not written more than T minutes ago (or K updates ago) for some fixed T (or K). <u>No guarantee on ordering of writes</u>**
- **Monotonic-reads consistency = Eventual consistency with the additional constraint that each read sees data that is not from before the previous version seen by that same process (yes, we did not have this previously!)**
- **Read-my-writes consistency = Eventual consistency with the additional constraint that each process will not read a version of data from before the most recent write they personally made to that data.**