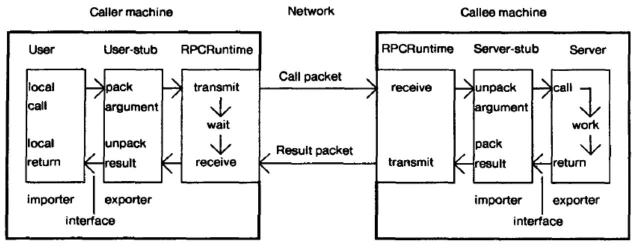
# CS5223 Distributed Systems

## Introduction

- **Distributed system**: interconnected computers that cooperate to provide some service.
  - Benefits:
    1. Higher capacity and performance.
    2. Ability to connect geographically separate systems.
    3. Reliable, always-on systems.
  - Challenges:
    1. **Fault tolerance**: keep doing useful work in the presence of partial failures.
    2. **Distributed state management**:
       * Keep data available despite failures through **replication**.
       * Make popular data fast for everyone by having **copies in different geographical locations**.
       * Store huge amounts of data through **sharding** and **partitioning**.
       * Ensure consistency and correctness of data through **consensus**.
         · Many problems are provable impossible to resolve, and trade-offs are often required.
    3. System design and architecture.
    4. Performance.
    5. Security.
    6. Testing and debugging.
- **Remote procedural calls (RPC)**: communication between nodes in a distributed system.
  - Core idea: use **abstractions** rather than explicit message patterns to achieve communication.
    * Compile protocol into stubs in charge of marshalling/unmarshalling.
    * Hide code for socket setup and message recv/delivery within a RPC library.
  - ⇒ Make a remote call look like a local function call.




  - Semantics
    1. **At-least-once**: procedure is executed on the server $\geq 1$ times, clients retry request until they get a response.
       * Useful for **idempotent** operations (i.e., same effect whether done once or multiple times).
    2. **At-most-once**: procedure is executed once or not executed at all.
       (a) Client includes a unique ID (e.g., seq #/timestamp + identifier) in every request.
       (b) Server keeps a history of request it has already answered, their ID, and the result.
       (c) If duplicate, server re-sends result (without re-execution).
       (d) RPC history can be garbage collected by the server in one of the following manners:
          i. Client includes "seen all replies $\leq x$" with every RPC, server discards all $\leq x$.
          ii. Each client is only allowed one outstanding RPC at a time, thus the arrival of $x + 1$ allows the server to discard all $\leq x$.
    3. **Exactly-once**: at-most-once + client retries until success.
       * This is not possible to achieve in general, because the server would have to store the RPC information in persistent memory in case it crashes.
         · Issues arise regardless of the order of (1) storage to persistent memory and (2) function call execution.

## Time and clocks

- Physical clocks have limitations (e.g., **time drift** → requires routine synchronization, which is itself a problem).
- **Happens-before** relationship: captures logical/causal dependencies between events.
  - Defines a **partial ordering** (→) over event times, specifically:
    1. **Irreflexive**: $a \not\rightarrow a$.
    2. **Antisymmetric**: if $a \rightarrow b$, then $b \not\rightarrow a$.
    3. **Transitive**: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
  - Rules:
    1. If $a$ comes before $b$ **within a process**, then $a \rightarrow b$.
    2. If $a = \text{send}(M)$ and $b = \text{recv}(M)$, then $a \rightarrow b$.
- Ordering events without physical clocks:
  1. **Logical clocks**: preserves happens-before relationships, a.k.a. Lamport clocks.
     - If $a \rightarrow b$ then $C(a) < C(b)$; but $C(a) < C(b)$ does not necessarily mean $a \rightarrow b$.
       * If $C(a) < C(b)$, it could also mean that events $a$, $b$ are **concurrent**.
     - Implementation:
       (a) Keep a local clock $T$.
       (b) Whenever an event happens (incl. `send` events), increment $T$.
       (c) On message receipt, $T = \max(T, T_m) + 1$, where $T_m$ rep. clock value of incoming message.
     - Can be used to define a **total ordering** (⇒) over event times, where:
       * If $C(a) < C(b)$, then $a \Rightarrow b$.
       * If $C(a) == C(b)$, then tie breaking is done by process ID.
  2. **Vector clocks**: preserves happens-before relationships and their converse.
     - $a \rightarrow b \iff C(a) < C(b)$.
       * If $C_x[i] < C_y[i]$ and $C_x[j] > C_y[j]$ for some $i$, $j$, then $C_x$ and $C_y$ are **concurrent**.
       * If $\forall i$, $C_x[i] \leq C_y[i]$ and $\exists j$, $C_x[j] < C_y[j]$, then $C_x$ **happens before** $C_y$.
     - Implementation:
       (a) Keep a local vector $C$ whose length = # nodes.
       (b) On node $i$, on each event, increment $C[i]$.
       (c) On message receipt with clock $C_m$ on node $i$:
          i. Increment $C[i]$.
          ii. For each $j \neq i$, $C[j] = \max(C[j], C_m[j])$.
- **Distributed mutual exclusion**:
  - Goals: similar to standard mutexes.
    1. **Mutual exclusion**: there is $\leq 1$ process in the CS at any time.
    2. **No starvation**: requesting processes eventually acquire the lock.
    3. **Order preservation**: locks are granted in *request order* (i.e., total ordering based on logical clocks).
  - Assumptions:
    * **No failures** (incl. node, link).
      · If node/link failures are present, the implementation below will run into a deadlock.
    * **In-order point-to-point message delivery** (i.e., messages from the same node are received in the order they were sent).
  - Implementation:
    * Each message carries a timestamp $T_m$.
    * Each node stores:
      1. A **queue** of **request messages**, ordered by $T_m$.
      2. The **latest timestamp** it has received from **each node**.
    * 3 message types: **request** (broadcast), **release** (broadcast), and **acknowledge** (on receipt).
      1. On receiving **request**: record message timestamp → add req to queue → send ack.
      2. On receiving **release**: record message timestamp → remove corresponding req from queue.
      3. On receiving **acknowledge**: record message timestamp.
    * To **acquire lock**:
      1. Send **request** to everyone, including self.
      2. Lock is acquired when:
         (a) My request is at the **head of my queue**, and
         (b) I have **received higher-timestamped messages** from everyone (compared to the request timestamp).
    * To **release lock**: send **release** to everyone, including self.

## Distributed State

- **Distributed state**: information retained in one place that *describes something*, or is *determined by something*, somewhere else in the system.
  (+) Performance.
  (+) Reliability.
  (+) Coherence.
- **Caching**: client-side.
  (+) **Reduces load** on a bottleneck service (by exploiting *locality*).
  (+) **Better latency**.
  – Compared to RPC, which **moves computation** to where the data is, caching **moves data** to where we need it.
- **Approaches**:
  - **Network file system (NFS)**: focuses on *simplicity*.
    * Features:
      · "**Stateless**": all essential information is kept on file server's disk, but servers do not cache client information.
      · **Idempotent** operations.
    * **Update protocol**: when a client writes to a file,
      1. Updates local cache.
      2. Sends write request to server.
      3. Server writes data to disks.
      (−) **Performance**: every client write request synchronously writes to server disks.
      (−) **Consistency**: other client caches are not notified of the updates → inconsistent data read.
        · Solution: **periodic polling**, clients eventually receive updates after some time.
  - **Sprite file system**: Unix-like distributed OS from Berkeley.
    * Features:
      1. Server tracks which clients are reading/writing which files → resolves the **consistency** problem.
         · `open()/close()` need to be used to contact the server.
      2. Only one client opens file ⇒ **write-back cache** used, where modified blocks are kept in memory and written back to disk after <u>30s</u>.
      3. Multiple readers and $\geq$1 writers ⇒ all reads and writes will go through the server (i.e., not cacheable).
    (+) **Performance**.
    (+) **Consistency**.
    (−) **Complexity**.
    (−) **Durability**: some files may be lost.
    (−) **Recovery**: server needs to reconstruct the information regarding its opened files upon restart → **recovery storm** (high load when there are many clients).
  - **Invalidation**: writer invalidates all other cached copies.
  - **Write-update**: writer updates all other cached copies.
- **Consistency**: the allowed semantics of a set of operations to a data store or shared object.
  - Consistency is a *safety* property, not a *liveness* property.
    * **Safety property**: specifies the "bad things" that shouldn't happen in any execution.
    * **Liveness property**: specifies the "good things" that should happen in every execution.
    Consistency properties specify the *interface*, not the *implementation*.
  - **Anomaly**: violation of the consistency semantics.
  - Levels of consistency:
    1. **Strong consistency**: the system behaves as if there's just single copy of the data, caching and replication are invisible to clients.
       (a) **Sequential consistency** (a.k.a. **serializability**): requires a history of operations to be *equivalent* to a *legal sequential history*.
          * **Legal sequential history**: a history that respects the *local ordering* at each node.
          * Sequentially consistent systems allow read-only operations to return **stale data**.
       (b) **Linearizability**: sequential consistency + respects **real-time ordering**.
          * If $e_1$ ends before $e_2$ begins, then $e_1$ appears before $e_2$ in the sequential history.
          * If they are **concurrent**, then any order is okay.
          ⇒ one of the strongest guarantees for concurrent objects.
    2. **Weak consistency**: allows behaviours significantly different from the single store model.
       (a) **Causal consistency**: non-concurrent writes (based on *happens-before*) must be seen in that order + concurrent writes can be seen in different orders on different nodes.
          * Linearizability implies causal consistency.
       (b) **FIFO consistency**: writes done by the same process are seen in that order + writes by different processes can be seen in different orders.
    3. **Eventual consistency**: if all writes to an object stop, all processes will eventually read the same value.
    Different levels of consistency are needed to strike a balance between:
    1. **Performance**: consistency requires synchronization/coordination when data is cached/replicated → generally slower.
    2. **Availability**: to enable clients who are offline to be able to use a service, weak/eventual consistency will be needed.
    3. **Programmability**: weaker models are harder to reason against.

### Primary/backup replication

- Goal: to provide **consistency** + **availability**.
  - **Consistency** can be achieved by *at-most-once* semantics.
- Components: **primary**, **backup**, and **view server**.
  - At any given time:
    * Clients talk to the primary.
    * Data is replicated on the primary and backup.
    * If the primary fails, the backup becomes the primary.
  - Basic operation:
    1. Clients send operations to primary.
    2. Primary decides on order of operations.
    3. Primary forwards sequence of operations to backup.
    4. Backup does either of the following:
       (a) Performs operations in the same order (i.e., **hot standby**).
       (b) Simply saves the log of operations (i.e., **cold standby**).
    5. After backup has executed/saved operations, primary replies to client.
  - Challenges:
    * **Non-deterministic operations**: the above approach no longer works, since executing the same operations does not necessarily lead to the same outcomes on the primary and backup.
      · Solution: transfer the *state delta* (i.e., how state has changed) to backup, instead of individual operations.
    * **Messages** between the primary and backup **may be dropped**.
    * Clients, primary, and backup **need to agree on the same primary** (there can be only one at a time) → requires the view server.
    * **Lack of progress**, due to:
      · View server failure.
      · Total network failure.
      · Client cannot reach primary but can ping the view server.
      · No backup, and primary fails.
      · Primary or backup fails before completing state transfer.
    * **Duplicate writes**: at-most-once semantics need to be upheld to prevent this.
- **View service**: decides who is primary and backup.
  - **View**: a statement about the current roles in the system (i.e., who is the primary/backup).
  - Failure detection:
    1. Each server periodically pings the view server (i.e., **heartbeat** message).
    2. To the view server, a node is:
       (a) "dead" if it missed $n$ pings.
       (b) "alive" after a single ping.
  - Server management:
    * View server maintains a set of "alive" servers (primary, backup, and idle).
    * A new view is declared during:
      1. **Primary failure**:
         (a) View server declares a new view, and moves backup to primary.
         (b) View server promotes an idle server (if any) as the new backup.
         (c) Primary initializes new backup's state, before processing any further requests.
         (d) When state transfer has completed, primary will ACK on the new view.
         (e) View server sets the new view as its current view.
      2. **Backup failure**.
      3. View has no backup, and there is an idle server.
- **Rules**:
  1. Primary in view $i + 1$ must have been the backup or primary in view $i$.
     – Prevents the **split brain** problem: new primary (elected from idle servers) has no information regarding the previous state.
  2. Primary must wait for backup to accept/execute each operation (both writes and reads) before replying to the client.
     – Prevents the issue with **missing writes**: primary crashes before writing to backup.
     – Propagating reads prevents **stale reads**: required for linearizability, but can be tolerated under sequential consistency.
  3. Backup must accept forwarded requests only if the view is correct.
     – Prevents the **partially split brain** problem: if backup responds to the old primary, the old primary will reply to the client sending the request, whereas the new primary will not have any info regarding the forwarded request.
  4. Non-primary must reject client requests.
     – Prevents **inconsistencies**.
     – However, if the new view contains the same primary as the old view, it is perfectly fine for the primary to respond to a request tagged with the old view.
  5. Every operation must be before or after the state transfer (i.e., **atomic state transfer**).
     – Prevents any lost in state (due to overwriting on the backup).

## Distributed Consensus

### Paxos

- **Goals** of consensus:
  1. **Safety**:
     (a) Only a value that has been proposed can be chosen.
     (b) Only a single value is chosen.
     (c) A process never learns that a value has been chosen, unless it has been.
  2. **Liveness**:
     (a) Some proposed value is eventually chosen.
     (b) If a value is chosen, a process eventually learns it.
- **FLP impossibility theorem**: it is impossible for a deterministic protocol to guarantee consensus in bounded time, in an asynchronous distributed system (even with only 1 faulty process).
- **Paxos**: achieves non-blocking consensus as long as
  1. a majority of participants are alive, and
  2. provided there is a sufficiently long period without further failures.
  - Assumptions:
    * Processes can crash and recover.
    * Asynchronous communication, via messages.
    * Messages can be lost and duplicated, but not corrupted.
  - Terminology:
    1. **Value**: a possible value/operation to reach consensus on.
    2. **Proposal**: to select a value; uniquely numbered.
    3. **Accept**: action on a specific proposal/value.
    4. **Chosen**: proposal/value is accepted by a majority of acceptors.
    5. **Learned**: proposal that is chosen is known by all participants.

- Roles:
  1. **Proposers**: propose values (proposals) to be chosen.
  2. **Acceptors**: accept/reject proposals, collectively decide when a value is chosen.
  3. **Learners**: once a value is chosen, learn the decided value.

  Multiple roles can co-locate on the same physical node.
- Protocol:
  1. Phase 1:
     (a) A proposer chooses a new $n$, and sends $<$ prepare, $n$ $>$ to a majority of acceptors.
     (b) If an acceptor $a$ receives $<$ prepare, $n'$ $>$ where $n' > n$ of any $<$ prepare, $n$ $>$ to which it has responded, it then responds to $<$ prepare, $n'$ $>$ with:
        i. A promise not to accept any more proposals numbered less than $n'$.
        ii. The highest numbered proposal (if any) that it has accepted.
  2. Phase 2:
     (a) If the proposer receives a response to $<$ prepare, $n$ $>$ from a majority of acceptors, then it sends to each acceptor $<$ accept, $n$, $v$ $>$, where $v$ is either:
        * The value of the highest numbered proposal among the responses, or
        * Any value, if the responses reported no proposals.
     (b) If an acceptor receives $<$ accept, $n$, $v$ $>$, it accepts the proposal unless it has in the meantime responded to $<$ prepare, $n'$ $>$, where $n' > n$.
  3. Once a value is chosen, learners will find out about it using either of the following strategies:
     (a) Each acceptor informs each learner whenever it accepts a proposal.
        * Once a learner hears that a majority of acceptors has accepted a single proposal, it knows for sure that this value has been chosen.
     (b) Acceptors inform a *distinguished learner*, who informs the other learners.

     In the event that a learner does not learn that a value has been chosen (due to crash failures/message loss), they can:
     (a) Check for the highest numbered proposal and its corresponding value $v'$.
     (b) Verify the value $v'$ with a proposer (i.e., by getting it to re-run the Paxos protocol, and checking if it is able to send $<$ accept, $n$, $v'$ $>$).
- Limitations:
  1. **Liveness/Progress** is not guaranteed.
     * Solution: elect a single distinguished proposer (generally works in practice, but not theoretically sound).
  2. Every operation requires 2 rounds of communication.
  3. Gaps in the log/numbering.
  4. No clear way of selecting a value to propose for an operation.
  5. No clear way of electing a distinguished proposer.

## PMMC

- Idea:
  1. Leader elects itself by running Paxos phase 1 for all instances.
  2. Once elected, leader only runs Paxos phase 2 for each proposal.
  3. If leader dies, other nodes repeat the process (i.e., elect themselves, etc.).
- Roles:
  1. **Replicas**: keep log of operations, state machine, and configurations.
  2. **Leaders**: drive the consensus protocol (similar to proposers).
     - Current active leader broadcasts heartbeat messages to all machines.
       * No heartbeat message received $\rightarrow$ assumes current active leader has failed $\rightarrow$ tries to elect self as next active leader.
     - Only propose one value per ballot and slot.
       * If a value $v$ is chosen by a majority on ballot $b$, then any value proposed by any leader in the same slot on ballot $b' > b$ has the same value.
  3. **Acceptors**: vote of leaders and accept **ballots** (proposals, numbered as '{seq#}.{leader#}').
     - Only accept values from current ballot.
     - If a value $v$ is chosen by a majority on ballot $b$, then any value accepted by any acceptor in the same ballot $b' > b$ has the same value.

## Raft

- Idea: two main protocols.
  1. Leader election.
  2. Log replication.
- Each **replica/server** plays the same role, with 3 states:
  1. **Leader**.
  2. **Follower**.
  3. **Candidate**.

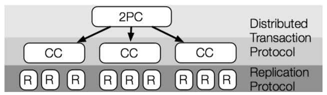
  Only the leader will handle client requests, and replicates them to followers.
- **Leader election**:
  - Properties:
    * Time is divided into **terms**.
    * Each term beings with an **election**.
      · At most one replica will win the election in a term.
      · Alternatively, no leader may be elected (in which case, the election for the next term begins immediately).
    * Each replica stores the current term number observed.
      · Messages with a lower term number will be ignored.
      · Upon observing messages with a higher term number, this value will be updated.
    * If a follower does not receive any heartbeat messages from the leader over a timeout period, it will start the next election.
  - Protocol:
    1. Follower increments current term, transitions to the *candidate* state.
    2. Candidate votes for itself, and issues `RequestVote` RPC to all other replicas.
       * Each replica can only vote for **at most one** candidate in a term.
       * Replicas will reject `RequestVote` if their local log is more **up-to-date** than the candidate's log.
         · **Up-to-date**: the log contains a later term, or ends with the same term but is longer.
    3. If a candidate receives votes from the majority, it transitions to the *leader* state.
       * If no leader is chosen, a new election is started with randomized timeouts on each machine.
- **Log replication**:
  - Properties:
    * If there is only one term, then all logs will be consistent.
      · Some follower(s) may be lagging behind, but their logs can never diverge.
    * Changes in the leader may result in divergence.
  - Protocol:
    1. Leader serves client requests.
    2. Leader appends request to its log, and issues `AppendEntries` RPC to all followers.
       * The RPC is retried indefinitely until all followers respond.
       * `AppendEntries` includes the index (and term #) of the log entry immediately preceding the new entry.
    3. (a) If the follower does not have an old entry, it rejects the request.
       (b) If the follower has a conflicting entry, it deletes the conflicting entry and all entries following it, and rejects the request.
          * If the request is rejected, the leader retires `AppendEntries` with a lower index, until the follower no longer sends a rejection.
       (c) Otherwise, it appends new entries to the log, and responds to the leader.
    4. **Log commit**: after the leader receives an entry from a majority of replicas, the log entry is deemed to be **committed**.
       * In other words, the entry is **durable** and no other command can be chosen at the same index.
       * This also implies that all prior entries in the log are also committed, and it is safe to execute the current command.
    5. Leader applies operation to state machine.
    6. Leader replies client with result.

# Transactions

- Goal: group a set of individual operations into an atomic unit.
  - Requirements: **ACID** properties.
    1. **Atomicity**: either the entire statement is executed, or none of it is executed.
    2. **Consistency**: transactions only make changes to tables in predefined, predictable ways.
    3. **Isolation**: concurrent transactions do not interfere with or affect one another.
    4. **Durability**: changes to your data made by successfully executed transactions will be saved, even in the event of system failure.
    * **Write-ahead logging** guarantees atomicity and durability.
      1. Write each operation (and its effects) into a log on disk.
      2. Write a commit record that makes all operations commit.
      3. Update client with the result of the operation only after the commit record has been written.
      · In the event of a crash, scan log and redo txs with a commit record; undo txs without.
    * **Two-phase locking** (an atomic commit protocol) guarantees isolation.
- **Atomic commit protocol (ACP)**:
  - Every node arrives at the same decision.
  - Once a node decides, it never changes.
  - Tx committed only if all nodes vote **Y**.
  - Under normal circumstances, if all nodes vote **Y**, the tx is committed.
  - If all failures are eventually repaired, the tx is eventually either committed or aborted.
- **Two phase commit (2PC)**:
  - Roles:
    * **Participants**: nodes that must update data relevant to the tx.
    * **Coordinator**: node responsible for executing the protocol.
  - RPCs:
    * **Prepare**: can you commit this tx?
    * **Commit**.
    * **Abort**.
  - Protocol:
    1. Coordinator sends *prepare* messages to all participants.
    2. (a) If all participants reply **Y**, send *commit* to all participants.
       (b) If any participant replies **N**, send *abort* to all participants.
    * Note: at each event (i.e., coordinator send, participant reply), write-ahead logging is used.

## Distributed transactions

- Isolation levels:
  - **Serializability**: each tx's reads and writes are consistent with running them in a serial order, one tx at a time.
  - **Strict serializability**: serializability + real time constraints.
  - Weaker isolation levels: **snapshot isolation**, **repeatable read**, **read committed**, etc.
    * Reduce transactional overheads $\rightarrow$ improve performance.
    * Permit various anomalies.
- **Two phase locking (2PL)**: ensures strict serializability.
  - Lock types:
    * **Read/shared lock**.
    * **Write/exclusive lock**.
  - Protocol: acquire and release locks in 2 phases.
    1. **Expanding phase**: locks are acquired, no locks are released.
    2. **Shrinking phase**: locks are released, no locks are acquired.
    * Locks are released when a tx commits/aborts.
    * Locks are acquired in some fixed order, and a tx is aborted if it is unable to acquire all locks.
- Google's Spanner:




  - Features:
    * Each shard is stored in a Paxos group, replicated across data centers.
    * Leader of each group keeps track of read/write blocks.
    * One group leader becomes the 2PC coordinator, the others become participants.
  - Protocol:
    1. Client reads objects from shards.
       * It contacts the appropriate Paxos group leader, acquires read locks, and buffers writes locally.
    2. Client decides to commit.
       (a) It sends *prepare* to all participant leaders.
       (b) Participant leader acquires write locks, Paxos-replicates a *prepare* log entry, votes either **Y/N**, and replies to the coordinator.
    3. If the coordinator receives **Y** from all participants:
       * Coordinator Paxos-replicates a *commit* log entry, replies the client, and sends *commit* messages to all participant leaders.
       * Participant leaders Paxos-replicate a *commit* log entry, applies all writes, and releases all locks.

# Miscellaneous

## Chubby

- **Distributed coordination service**, Google's equivalent of Apache Zookeeper.
  - Goal: allow client applications to synchronize and manage dynamic configuration state.
- Interface: similar to a file system, implemented as a lock service (instead of a library).
- Performance optimizations:
  1. **Batching**: master accumulates requests from many clients, and performs a single round of Paxos to commit all requests to the log.
     - Higher throughput, poorer latency.
  2. **Partitioning**: run multiple Paxos groups, each responsible for different keys.
     - Throughput scales with the number of groups, assuming (1) uniform request distribution and (2) no cross-group operations.
  3. **Leases**: master allowed to process reads alone while holding lease (around 10s).
     - Clients are able to receive responses immediately while master has not changed.
  4. **Caching**:
     - **Write-invalidate cache**: master tracks which clients might have their file cached, and sends invalidations on update.
     - **Heartbeat messages** between clients and server: when clients stop sending heartbeat messages (i.e., a lease), their locks are released and they will empty and disable their cache.
  5. **Proxies**: used to track state for groups of clients, reducing the amount of communication at the server.

---

CS5223 Distributed Systems (incomplete) author: arsatis