Abstractions: • hides implementation details
↳ Libraries: reusable

- Abstraction: A concept in computer science where unnecessary details are hidden to focus on the essential features.
- Purpose: Simplifies complex systems by providing a general view, allowing us to concentrate on broader goals without distraction.
- Illustration Example:
- If we represent “birds” abstractly, we refer to general characteristics without specifying individual features.
- This helps us understand the concept of birds in general, rather than learning about one specific type.
- Everyday Computing:
- We use computers without understanding or building the hardware and operating systems from scratch.
- Developers rely on libraries instead of writing complex functionalities themselves, which speeds up development and makes work manageable.
- Benefits in Programming:
- Libraries: Offer ready-to-use functions while hiding implementation details, making development more efficient.
- Reusability: Well-designed abstractions can be used across multiple projects, saving time and effort.

- **Database Abstraction**: Simplifies handling complex database operations involving multiple users.

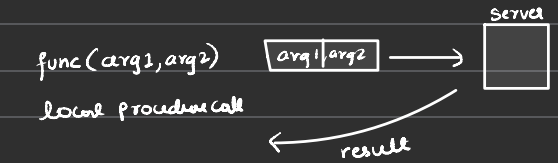
- **Transactions**:
  - Provide a simple interface: commit for success or abort for failure.
  - Ensure data moves from one consistent state to another, even with multiple concurrent operations.
- **Purpose**:
  - Users don’t have to worry about the complex issues of data conflicts or inconsistencies.
  - Allows focusing on business logic rather than the technical details of concurrent data changes.

Distributed systems

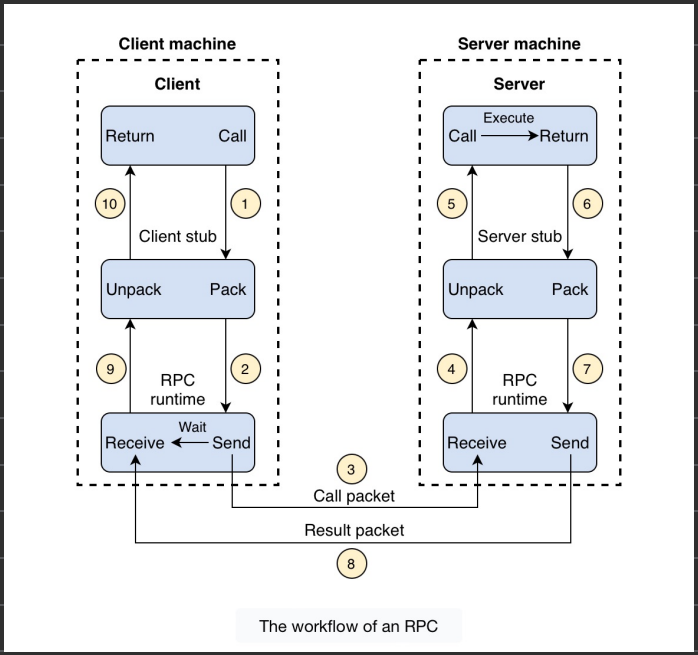
- Abstractions in Distributed SystemYs: Simplify the complexities of managing distributed environments.
- Purpose:
- Allow engineers to focus on application development instead of dealing with complex distributed system mechanics.
- Use in Industry:
- Popular in services like Amazon AWS, Google Cloud, and Microsoft Azure.
- These platforms provide different abstraction levels, hiding implementation details from users.
- Scalability:
- Essential for today’s applications, which need to handle large numbers of users.
- Abstractions enable quick and efficient scaling by transitioning to distributed systems.

② RPC:

Remote procedure calls (RPCs) provide an abstraction of a local procedure call to the developers by hiding the complexities of packing and sending function arguments to the remote server, receiving the return values, and managing any network retries.



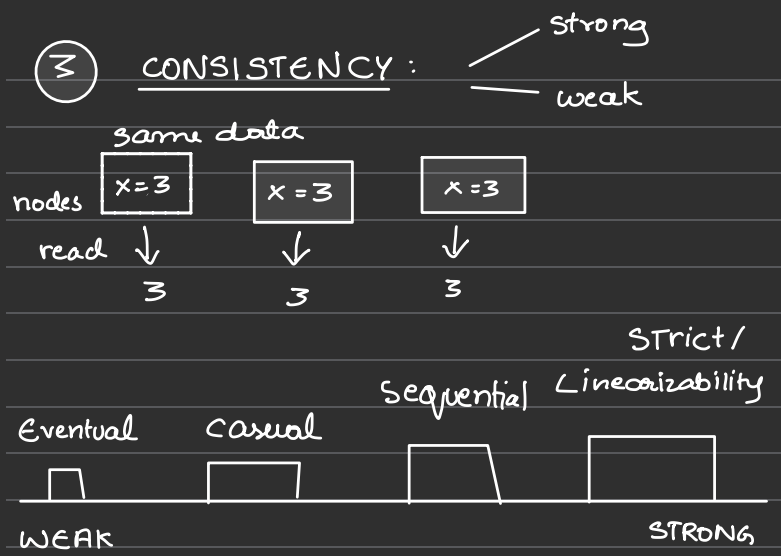
- RPC is called on a different machine



The workflow of an RPC

5 main parts

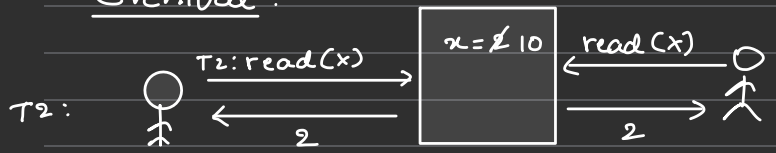
- client
- client stub
- RPC runtime
- server
- server stub



- **ACID Consistency**:
  - Ensures that database rules are followed to maintain data integrity.
  - Examples:
    - **Uniqueness Rule**: If a schema requires a unique value, the system ensures it remains unique across all actions.
    - **Foreign Key Rule**: If deleting a base row requires deleting associated rows, the system enforces this consistently.

- CAP Consistency:
  - Ensures all replicas in a distributed system have the same logical value.
  - Logical Guarantee: Values appear consistent to clients, even if physical replication takes time (e.g., due to network delays).
  - Client Access Control: Prevents clients from seeing different values across nodes, maintaining a coherent view of the data.

Eventual :



at T3 : x = 10 is updated

- no strict ordering
- Write heavy: never converges
- all nodes have diff replicas

he domain name system is a highly available system that enables name lookups to a hundred million devices across the Internet. It uses an eventual consistency model and doesn't necessarily reflect the latest values

causally

Causal — Dependent : related op
\ Independent

Process :

x = a — P1 write (x) a

b = x + 5

y = b — P2

since b = x + 5,

read (x) and write (y) are causally related

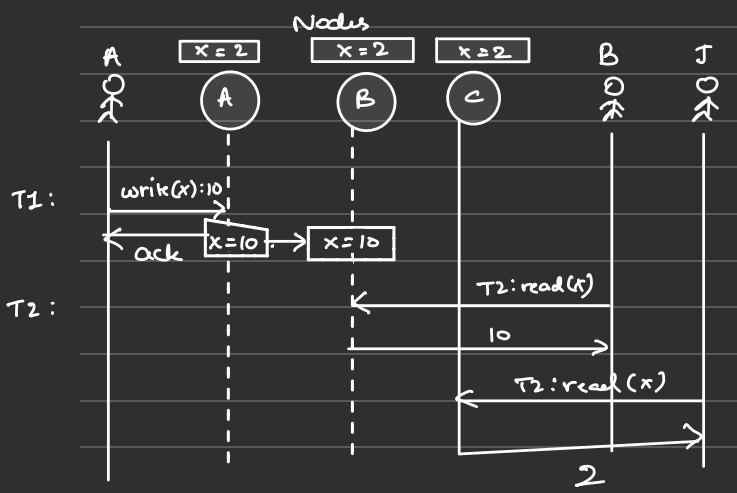
eg: commenting.

Sequential : stronger than causal

- preserves order
- same order as global clock ✗

Linearizability aka Strict consistency:

- read = latest values
- acknowledge.
- challenge in Distri system



- worse performance
- Strong assurance
- seq : reals time order

④ FAILURE SPECTRUM :

easy → difficult

Fail Stop | Crash | Omission | Temporal | Byzantine

Fail stop:
- Halts permanently
- other nodes detects

Crash :
- node halts silently
- other nodes cant detect non working

Omission :
- node cant send/recieve messages.
- send omission failure
  or
  receive omission failure

Temporal :

- too late to be useful
- bad algo / bad design
- loss of synch clocks

Byzantine :

- random behaviour

eg : transmit arbitary msg
wrong results, stop midway