CONCURRENCY AND REPLICA CONTROL :—

Handling large clients

Handling multiple servers

CONCURRENCY CONTROL :—

RPC → Remote Procedure call. → Abstraction for processes to call functions in other processes.

Ⓐ Code reuse

For object based settings
↳ RMI (Remote method invocation)

LOCAL PROCEDURE CALL → Call from one function to another within the same process.

Ⓐ Stack usage to pass arguments and return values. ⇒ Object access via pointers. (because same address space)

↳ Follows Exactly-once symantics.

RPC :— Caller and callee are in different processes.

⇓

Basically crosses process boundaries.

Ⓐ Object references via global pointers.
Ex. Obj. Address = IP + Port + Obj. Number

FAILED CALL :—

→ Request message dropped
→ Reply message dropped
→ Called process fails ↳ before execution / ↳ after execution

Ⓐ These cases are hard to distinguish

MULTIPLE CALL :—

→ Request message duplicated by the network

Possible Symantics

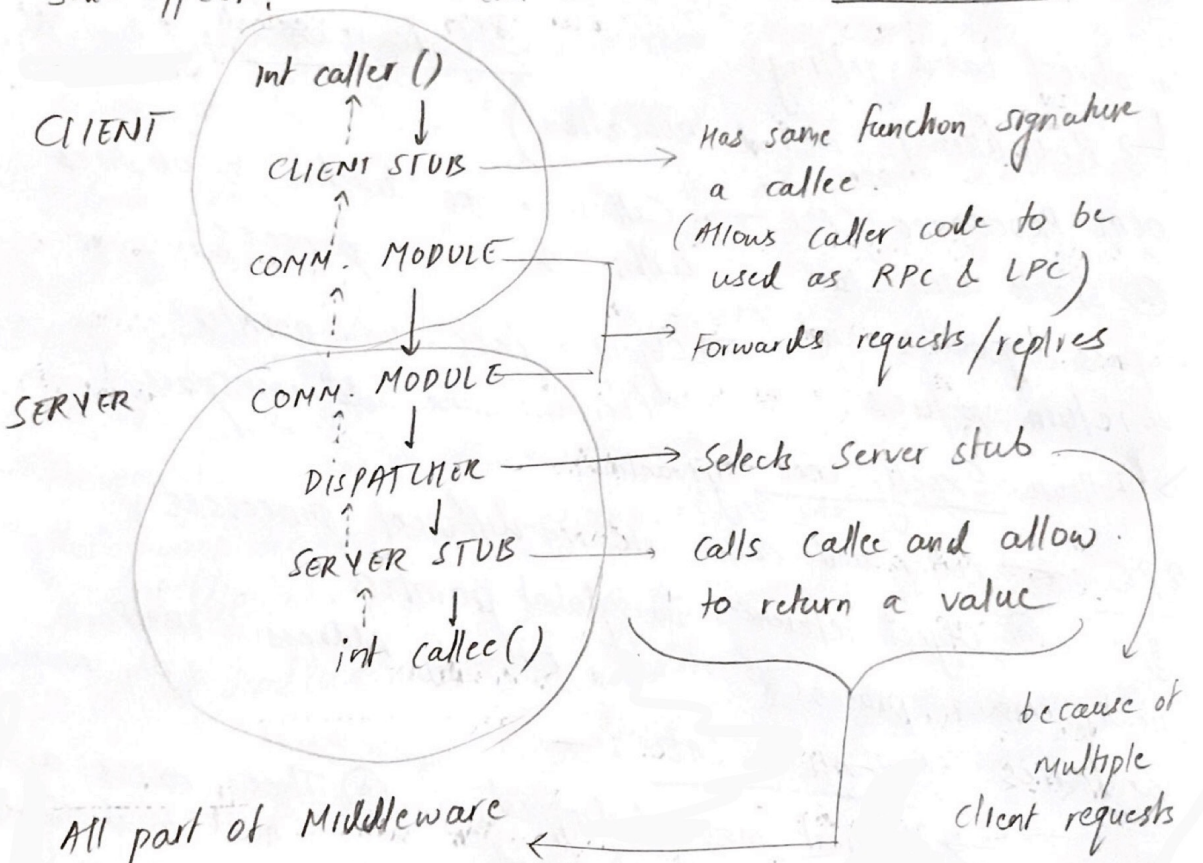
AT MOST ONCE → JAVA RMI

AT LEAST ONCE → Sun RPC

MAYBE → CORBA

	RETRANSMIT	FILTER DUPLICATES	RE-EXCUTE OR RETRANSMIT REPLY
AT LEAST ONCE	Yes	NO	Re-execute
AT MOST ONCE	Yes	Yes	Retransmit
MAYBE	NO	N/A	N/A

(*) Idempotent Operations can be repeated without any side effect. → can be used with AT LEAST ONCE



MARSHALLING → Caller converts arguments from platform dependent format to COMMON DATA REPESENTATI (CDR)

(*) CDR is used by Middleware to become Platform independent

REVERSE PROCESS. → Unmarshalling.

TRANSACTIONS :

→ Series of operations executed by client
→ Each operation is an RPC to a server

Either commits all operations at server

Or Aborts all operations and has no effect

REQUIREMENTS :—

- → Atomicity
- → Isolation → Indivisible from point of view of other transactions (Basically free from interference)
- → Consistency
- → Durability

PROBLEMS :—

① Lost update Problem (solved by Isolation)

② Inconsistent Retrieval

SERIAL EQUIVALENCE :—

An interleaving O of transactions is serially equivalent iff there exists ordering O' of those transactions one at a time which gives same end-result (for all the objects and transactions)

- Read → Affects client
- Write → Affects server.

CONFLICTING OPERATIONS

→ Combined effect depends on order of execution

EX. Read(x), write(x)

Ⓡ Serial Equivalence achieved when pairs of conflicting operations are in same order for all the objects

Ⓐ Must abort one of the transactions when conflict in 2 transactions detected.

2 APPROACHES : ① PESSIMISTIC ② OPTIMISTIC

① Prevent access → Use exclusive locks (Read-write locks)

PROBLEM: Reduces concurrency.

② check at commit time

TWO PHASE LOCKS :- (PESSIMISTIC CONCURRENCY CONTROL)

→ Transaction cannot acquire any locks after it has started releasing locks.

PHASE ①: Growing Phase
PHASE ②: Shrinking phase
] → Guarantees serial equivalence

DISADVANTAGE : Deadlocks

- ★ Expensive
- ★ Might not even be a deadlock
- ★ Wasted work

FIX :

① Timeout : Abort transaction

② Deadlock Detection :
- ★ Keeping track of weight graph
- ★ Find cycles
- ★ Abort one or more transactions to break cycle

③ Deadlock Prevention :
1. Allow read-only access to objects
2. Allow pre-emption of some transactions
3. Compound locks → Lock all objects at the beginning

② OPTIMISTIC CONCURRENCY CONTROL

1. BASIC APPROACH: Check for serial equivalence, abort if not satisfied → Leads to cascading abort

↓

Transactions who read dirty data of aborted transaction needs to be aborted

2. TIME STAMP ORDERING :

↓

Transaction id determines its position in serialization order (Abort if rule violated)
- ★ Read only if write by lower id in past
- ★ Write only if read/write by lower ids in past

③ MULTI-VERSION CONCURRENCY :—

→ Per-transaction version of object is maintained and marked as tentative versions (alongside committed version at server).
↙ Has timestamp

→ On read-write, mark correct version to read or write from
↳ based on transaction id

Eventual consistency → Similar to Optimistic concurrency

RIAK KEY-VALUE STORE → Vector clock implements Causal ordering.

Ⓐ Sibling value resolved by user or application

REPLICATION CONTROL :— ↗ Fault tolerance ↳ Load balancing

Higher availability? $1-f$ per server ⇒ $1-f^k$ any 1 server is working

CHALLENGE : → Transparency (Replication must be invisible) ⇒ Frontend
↳ Consistency (Sees single consistent copy)

Ⓐ Concept of Replicated State Machines

① PASSIVE REPLICATION

→ Elected master in the system ⇒ Determines total ordering of all updates.
(On leader failure, run election)

provides replication transparency

② ACTIVE REPLICATION :

Frontend multicasts the request to read/write to entire replica group
↳ Can use any ordering

Total order on multicast ⇒ same inputs to replicas.

Ⓐ Failures in active replication dealt by Virtual Synchrony

ONE-COPY SERIALIZABILITY :-

A concurrent execution of transaction in a replicated database is one-copy-serializable if it is equivalent to serial execution of these transactions on a single logical copy of database

FINAL OBJECTIVE = Serial equivalence + One-copy Serializability

Ⓐ A transaction may touch different servers for different objects. Commit must commit to all or no servers (ATOMIC COMMIT).

Ⓐ ONE-PHASE COMMIT

⇒ special server Coordinator initiates atomic commit

PROBLEM :- ① Problems at a single server like corrupted object
② Server may crash before receiving commit msg.

Ⓐ TWO-PHASE COMMIT

① Coordinator sends PREPARE message to servers. → retrievable after crash.
② Server saves updates to disk and reply YES or NO
③ If all votes received are YES within the timeout, it sends commit message. Otherwise abort.
④ On commit, server commits and sends ACK. → server can poll if no decision received

To deal with COORDINATOR crash, it logs all messages, decisions on disk. After recovery, new election.