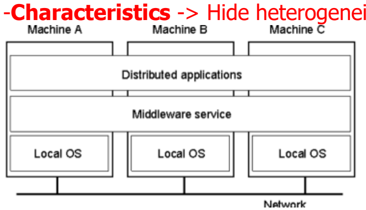
CHAPTER 1:

-**Distributed System** → collection of independent computers that appears to its users as single coherent system

-**Characteristics** -> Hide heterogeneity, easy to expand or scale, always up, layered design



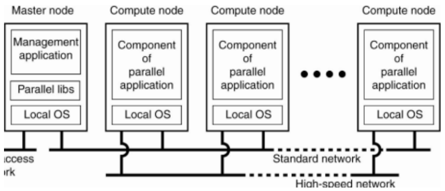
-middleware is above the OS in example to left

-**Design Goals ->** Make resources accessible (efficient sharing), Transparency, Openness (services describing syntax and semantics, TC/IP, IDL), Scalability (size, geo, admin: balance load to help) & security

-**Transparency Types ->** Access, location, migration, relocation (moved while in use), replication, concurrency (resource used by multiple), failure, persistence

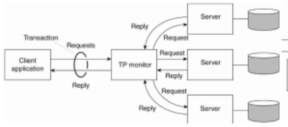
-**Never Assume ->** network is reliable, secure, homogeneous: topology no change, 0 latency, infinite bandwidth, 0 transport cost, 1 admin

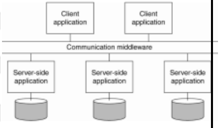
-**Types of D.S ->** High-performance (cluster (homo)/gird (hetero)), Information System (transaction, P2P), Pervasive Systems (sensor/ mobile battery powered)



-**Transaction ->** (atomic, consistent, isolated, durable)

-Nested: (apps don't talk to each other old)





CHAPTER 2:

-**System Architecture**-> instantiation of software architecture (where components placed)

-**Software Architecture->** Defines how various components should be organized and interact

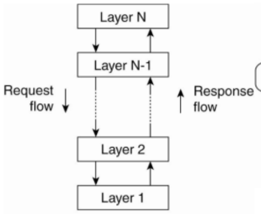
-**Layered** -> only talk to layer below, (networking), requests down & results up

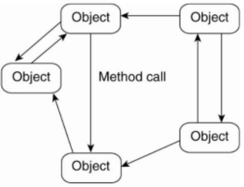
-**Object-Based->** looser than layered, components (objects) communicate through RPC, this is also what client server uses

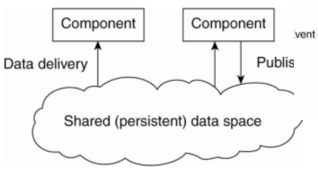
-**Data Centered ->** communicate through common repo, basically only through files, modern WWW is this by only exchanging files

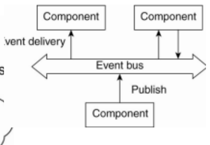
-**Event Based ->** events carry data, processes subscribe to events, middleware notifies, loosely coupled b/c no process refers to each other by name, just events in middleware

-**Shared Data Spaces** -> event based + data centered, both don't have to be active to communicate





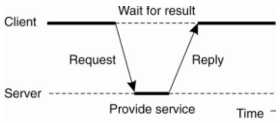




(ALL **ARCHITECTURAL STYLES** AIM TO ACHIEVE TRANSPARENCY)

-

-**Centralized Architecture->** (not a style) hard to know if request lost or it is taking a long time :(

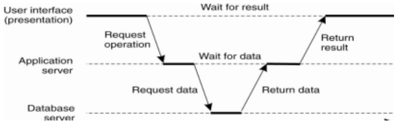


-**Idempotent ->** can be repeated multiple times without affecting the state of server (no harm)

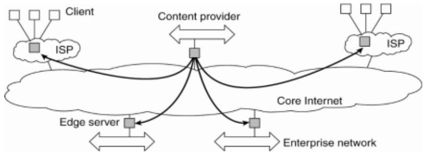
-**Application Layering**-> splitting up various UI, App, & DB (hard to know where line is between server and client)

-**Decentralized ->** Peer to peer is the only true decentralized

-**Multitiered ->**



-**Edge Server ->** hybrid between central/decentralized architecture, edge is boundary between enterprise network & actual internet (ISP, access points), serves content



CHAPTER 3

-multithreading in DS allow processing and communication (& multiple channels) @ same time

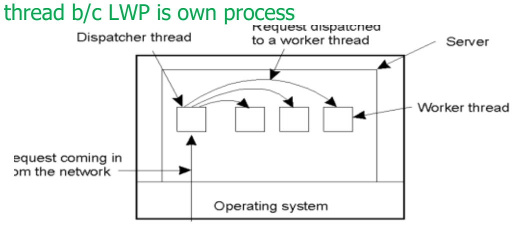
-**Thread ->** holds minimal info to allow for CPU to be shared between multiple threads, runs in shared memory space and resoures, protecting data is up to the app developer: has registers and stack still, not protected from eachother but processes are

-**User Level Implementation->** cheap to make, blocking thread will block process

-**Lightweight Process ->** hybrid between user and kernel, user level threads assigned to a LWP in kernel and has own stack; blocking call switches thread b/c LWP is own process

-**Dispatcher->** reads incoming requests, chooses worker to deal with, waits again

-**Worker->** typically a pool that does productive work assigned from dispatcher


Dispatcher thread
Request dispatched to a worker thread
Server
Worker thread
equest coming in om the network
Operating system

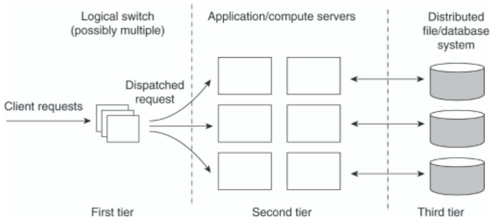
-**Server Design Issues->**Organization (iterative/concurrent), Binding (where clients connect), Interrupting (should be allowed? where), state vs stateless

-**Out-of-Band Data->** data processed by server before any other data used for interrupts, allows for only one connection unlike our project to terminate

-**Stateless Server ->** does not keep info on the state of its clients, think webservers, (do keep small amount of info like logs but if lost doesn't affect the client)

-**Soft-State ->** server promises to maintain state for period of time to client (notifying updates)

-**Stateful Servers ->** maintains persistent information on the client, info must be explicitly deleted by the server, performance gain w/ less read/writes, have to deal with crashes


Logical switch (possibly multiple)
Application/compute servers
Distributed file/database system
Dispatched request
Client requests
First tier
Second tier
Third tier

-**3 Tiered Cluster Server->** (image to left) the switch helps appear DS is a single unit as well as distributes the loads

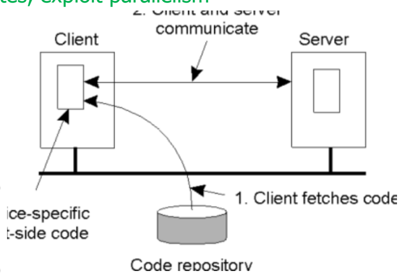
-**TCP Handoff->** client does not know that there is a handoff, switch determines the best server and forwards requests, inserts switch's IP on the way back

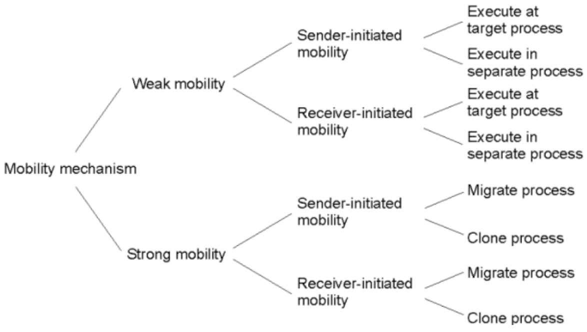
-**Reasons for Code Migration->** load distribution, reduces communication cost, Dynamic configuration of client to talk to server without updates, exploit parallelism

-**3 Program Segments->** Code, Resource, Execution

-**Weak Mobility->** only move the code segment (JS)

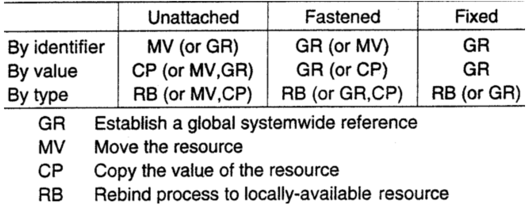
-**Strong Mobility->** moves all 3 so it can resume where it left of




Mobility mechanism
Weak mobility
Sender-initiated mobility
Execute at target process
Execute in separate process
Receiver-initiated mobility
Execute at target process
Execute in separate process
Strong mobility
Sender-initiated mobility
Migrate process
Clone process
Receiver-initiated mobility
Migrate process
Clone process

-**Process-Resource Bindings-** By ID (strongest, needs exact resource), by value (different resource can give same value), & by type (weakest, local devices like keyboards and monitors)

-**Resource-Machine->** Unattached (data files associated with process), Fastened (can move but at high costs, like DB or large file), Fixed (bound to specific machine, can't move, local port)



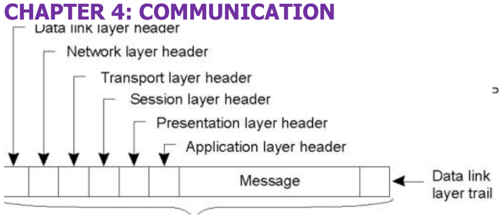
	Unattached	Fastened	Fixed
By identifier	MV (or GR)	GR (or MV)	GR
By value	CP (or MV,GR)	GR (or CP)	GR
By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

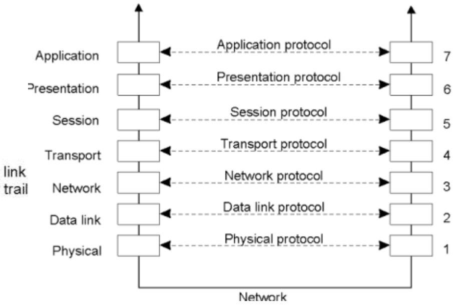
GR	Establish a global systemwide reference
MV	Move the resource
CP	Copy the value of the resource
RB	Rebind process to locally-available resource

-**Memory Migration Ways ->** push all pages, send modified pages, stop VM, migrate all, start VM, On demand paging (solution is combine 1&2)

-**Main points of Code Migration ->**
1) performance gains
2) flexibility
-handle heterogenous with VM's

CHAPTER 4: COMMUNICATION





-**When A talks to B->** A makes msg in own space, tells OS to send over network to B

-**OSI Model->** allow open systems to talk

-**OSI Layers ->** Physical (standardizes voltages 0's1's), Data Link (detects errors w/checksum), Network (routing), Transport (turns layers before useful for app), Session (sync & dialog), Presentation (bit meaning & formatting), App (app specific, HTTP)

-**Middleware Protocols->** middle ware lives in Application layer (authentication, commit protocols (Atomic), locks), this is the presentation & session layers (with a little in transport)

-**Persistent Comm->** msg stored by middleware as long as it takes to deliver, receiver doesn't have to be online, think email

-**Transient Comm->** msg stored only as long as both systems executing/running, if interrupt or receiver not online, msg dies, most transport layers this, think phone calls/ home routers

-**Synchronous Comm->** blocked until middleware notifies it got it or until delivered or response

-**Remote Procedure Call->** allow programs to call routines on other machines without the programmer or client knowing its happens (transparency)

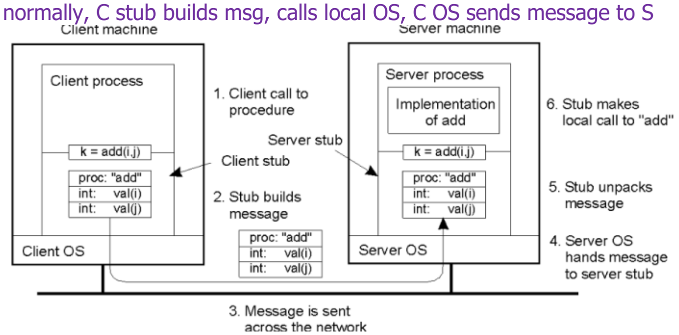
-**Conventional Procedure Call ->** params pushed to stack, procedure runs, return value in register, remove return address, transfer control back to caller, caller clears stack

-**Call by Copy/Restore**-> variable copied to stack, manipulated, & copied back; essentially has the same effect as copy by reference

-**Client Stub->** proxy code put into client, calls the local OS but no ask for data, instead packages parameters, and request it be sent elsewhere (Client doesn't know but stub does), once msg back, OS puts message in buffer, resumes control and reads from it like nothing happened

-**Server Stub->** same thing as client stub, makes network msg into local code, after it unpacks, server has no idea its RPC (params on local stack), processes it, sends msg back

-**RPC Steps**-> C calls C stub normally, C stub builds msg, calls local OS, C OS sends message to S OS, S OS gives msg to S stub, S stub unpacks params on stack & calls S call, S does work, results back to S Stub, S stub packs msg & calls local OS, S OS sends message to C OS, C OS gives msg to C Stub, C stub unpacks result, returns to client



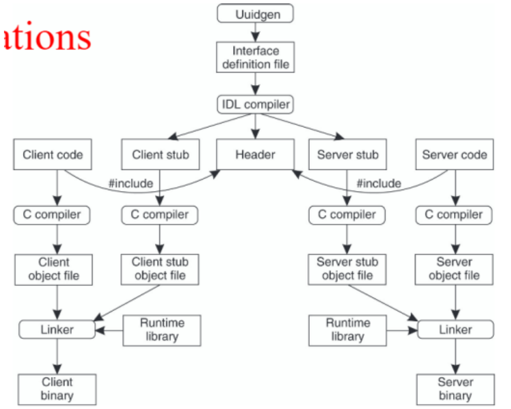
-**RPC w/ Reference Params->** either don't allow it or send copy as separate msg and send pointer to that msg, then copy it back client side; essentially call by copy/restore; if know buffer is input/output you do not need to send it twice (doubles speed), still high cost, no work for Objects

-**Param Spec & Stub Generation ->** make standard method to turn msg to function on C&S

-**Async RPC ->**still blocks for acknowledgement unless its One-way-RPC (not reliable)

-**Distributed-Computing-Environment RPC->**true middleware, offers distributed file, security and time services, add to existing machines w/o messing up current apps (make distributed now)

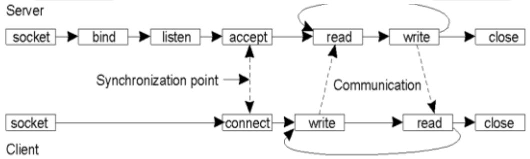
-**Interface-Definition Language->** glue that holds everything together, define syntax not semantics, Global id (uuidgen) to find it, IDL makes the client and server stubs

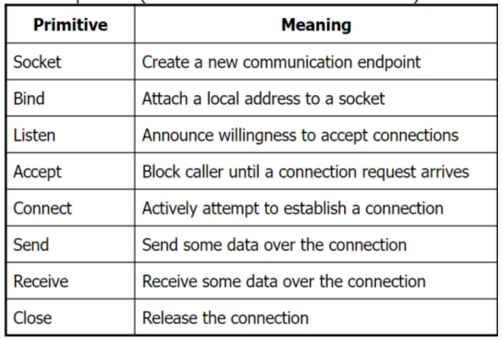


-**Client Binding ->** locate machine, DCE daemon to locate correct port (in table)

-**At-most-Once ->** no call repeated out of fear of messing up (opposite idempotent)

-**Berkley Sockets ->** RPC and RMI not always appropriate, sockets abstraction of OS endpoints (execution bottom left of this)





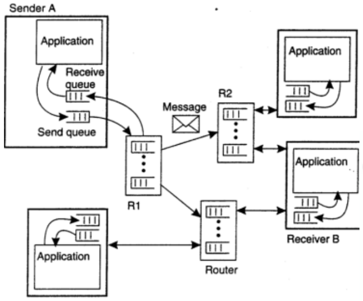
Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

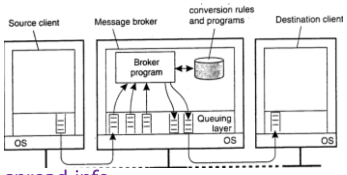
-**Message Passing Interface ->**sockets primitive & generic, transient, gID and ID, high performance with, local and remote buffers, loosely coupled b/c not referring by name (4)

Message Oriented Persistent Com (MoM)-> persistent async, msgs take minute not sockets

-**Msg Interface->** Put(append msg to specific Q), Get(block until Q !empty, remove 1st msg), Poll(check specific Q for msgs, remove 1st, nvr block), Notify(callback when msg put in Q)

-**Architecture->**queues must be local/nearby, DB is kept of these locations like DNS, unique



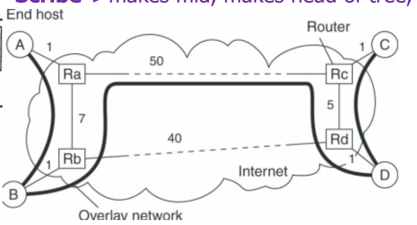


-**Msg Broker->** conversions, can match machines, rules DB

-**Multicasting->** at middleware level not IP (no ISP support)

-**Overlay->** Tree based(unique path, failure, simple), Mesh(many paths, and offers robustness because of this)

-**Scribe->**makes mid, makes head of tree, other nodes join



-A forms tree, costs are #s, more efficient to go AC, not AD

-**Multicast Tree Measurements-** Link Stress(often packet comes across same link, > 1 means not good, smaller #'s on chart i.e 7), Relative Delay Penalty(delay between nodes in ratio, divide #'s on edge), Tree cost(fixing the other 2)

-**Gossip/Epidemic->**spread info like disease, rapidly spread info