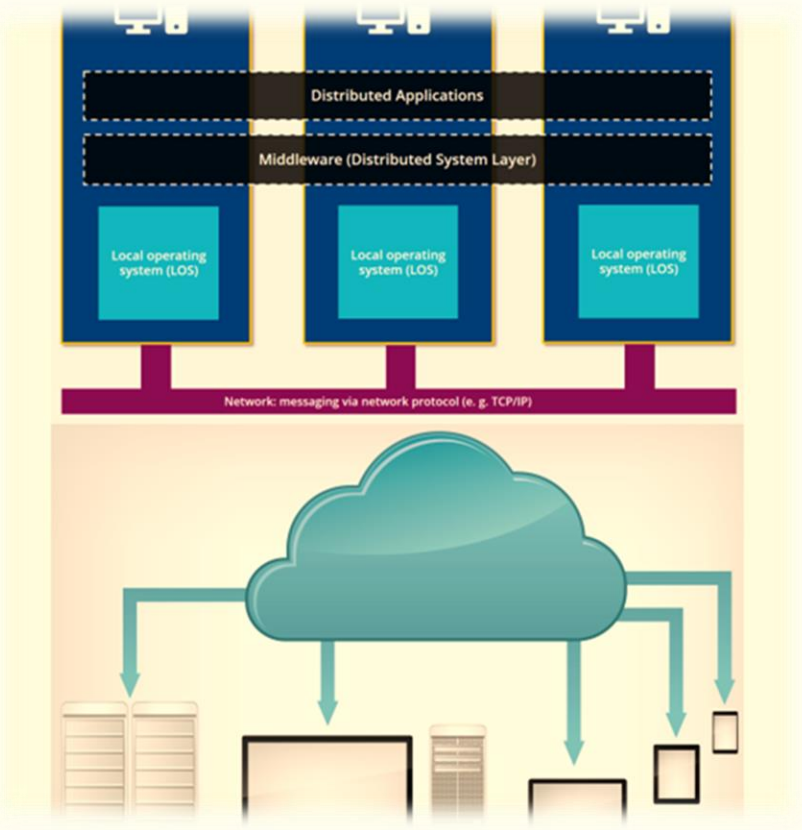
Faculty of Engineering & IT
Software Engineering Department

ITSE5336 – Distributed & Cloud Computing



Demonstrating distributed systems architectures, middleware, & cloud computing

Dr Mosab M. Ayiad

Spring 2022/2023

Module Overview

Module Description:

- In the mid of 1980s, microprocessors and high-speed networking were the two main advances that changed software systems. Approaches for parallel and distributed computing were introduced and continually evolved.

- Nowadays, powerful mini-computers (e.g., smartphones) are inter-networked and provide innovative services enabled by pervasive and cloud technology.

- This module takes students on an educational journey that begins with the concepts and models of distributed computing and systems and proceeds toward the paradigms of pervasive and cloud computing.

- The module covers several topics, ranging from the architecture of distributed systems, design issues, and common major problems to the recent cloud technologies, layouts, and services.

- The module is arranged into an incremental approach and the learning material is introduced in a gradual manner. The teaching involves practice occasions with chosen learning items for students to enhance their comprehension and skills.

- This module does not cover Security issues of distributed and cloud computing.

Aims and Objectives:

- The module aims to:
  1. Introduce students to distributed and cloud systems.
  2. Empower the understanding of the difference of systems architecture and application development.
  3. Introduce students to cloud computing environments, challenges and services.

- Module objectives:
  1. Understand the design challenges and implementation problems in distributed computing.
  2. Equip students with the knowledge to solve and handle distributed computing issues.
  3. Enable students to practice the concepts of parallel and distributed systems programming.
  4. Understand the design patterns of cloud applications.
  5. Demonstrate recent implementation issues in cloud computing.

Module Outcomes:

- By the end of this module students:
  1. Understand various system models such as: layered-based, event-based and object-based architectures.
  2. Understand and able to implement parallel processes and distributed applications.
  3. Being able to design and implement systems using remote procedure call and massage passing techniques.
  4. Being able to identify and solve distribution design issues such as: naming, agreement, scalability, availability and reliability.
  5. Understand the layouts of cloud computing and main could environments.
  6. Being able to design and implement cloud systems using centralised and decentralised approaches.
  7. Understand the design and implementation issues raised by ubiquitous computing and the engagement of IoT.

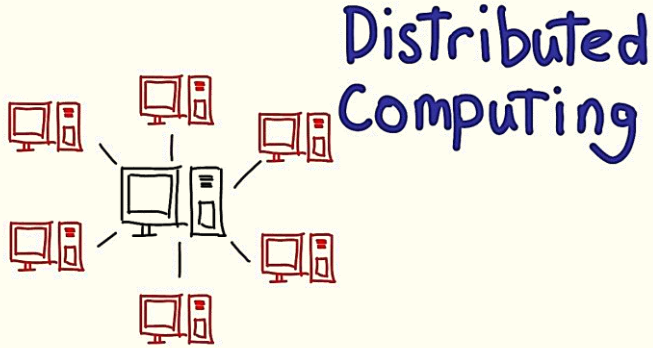
Module Structure:

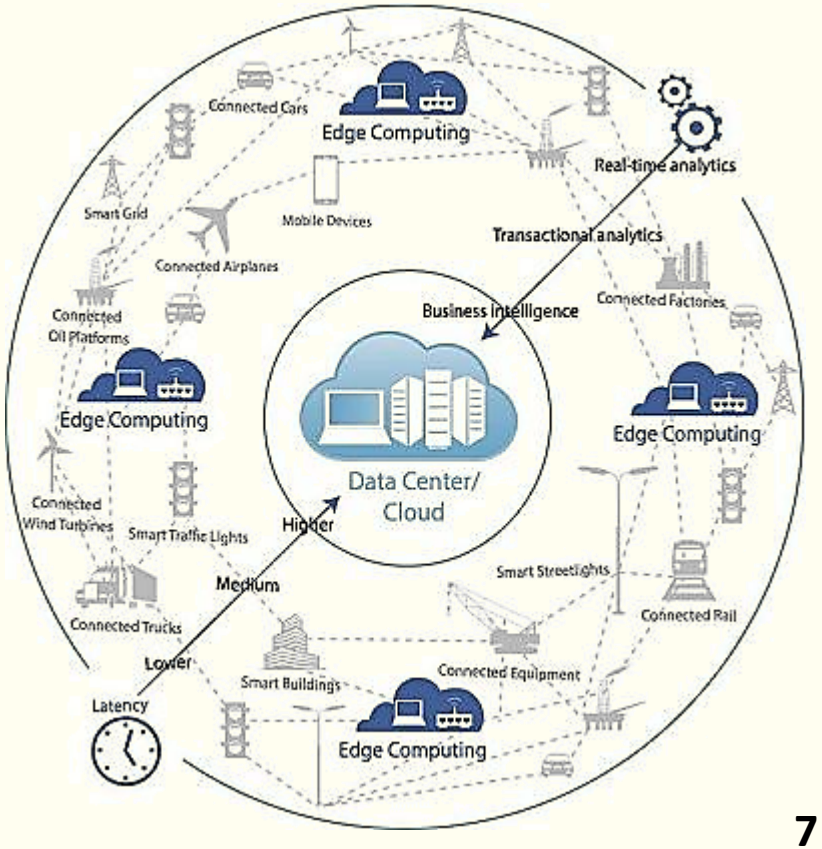
- The lectures involve:
  - Frontal lectures, paper presentations, in-class exercises, and team works.
  - The first portion of the module introduces basic concepts and approaches in distributed and cloud computing.
  - The next portion focuses on the concepts in distributed computing; middleware and major issues and solution approaches.
  - The final portion considers cloud technology. It involves the main cloud providers, models, and novel services.

Module Syllabus:

Weeks:

- **01**: Module overview.
- **02**, **03**: Introduction.
- **04**, **05**: Architectures of Distributed Computing (DC).
- **06**: Processes.
- **07**: Communications.
- **08**: Reading & mid-term examinations.
- **09**: Reading & mid-term examinations.
- **10**: Naming & Coordination.
- **11**: Replication & Consistency & Fault-tolerance.
- **12**: Orientation to Cloud Computing (CC).
- **13**: CC Environments.
- **14**: Virtualization in CC.
- **15**: Fog, Edge, & Ubiquitous Computing.
- **16**: Final examinations.





Practice sessions:

- There will be no in-lab practice sessions in this module, however, there are coursework assignments, in-class practices, and homework.

Grading:

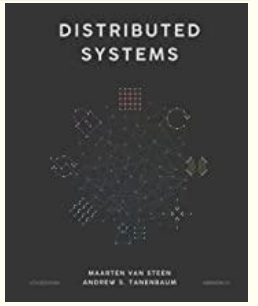
Pre-midterm assignments:	**20**%, to be submitted before Midterm exam (Week 3, 7).
Midterm exam:	**30**%, ~Week 8.
Third assignment:	**10**%, to be submitted before Final exam (Week 13).
Final exam:	**40**%.

Vital notes:

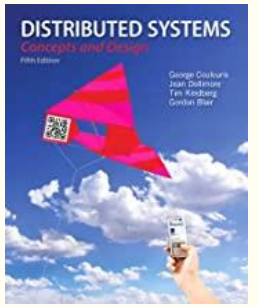
- ❑All assignments are per-student tasks.
- ❑The submission of all assignments will be via Moodle.
- ❑All exams are conducted in invigilated paper examinations.

Text Books and References:

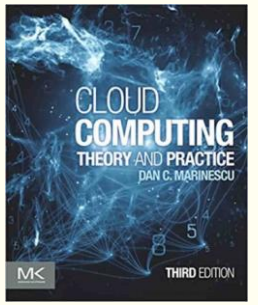
I. M. van Steen, A. S. Tanenbaum, "*Distributed Systems*", 4th ed, 2023, Pearson Education.



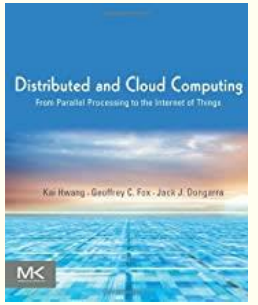
II. G. Coulouris, J. Dollimore, T. Kindberg, G. Blair, "*Distributed Systems Concept and Design*", 5th ed, 2012, Pearson.



III. D. C. Marinescu, "*Cloud Computing Theory and Practice*", 3rd ed, 2022, Morgan Kaufmann.



IV. K. Hwang, G. Fox, J. Dongarra, "*Distributed and Cloud Computing from Parallel Processing to the Internet of Things*", 1st ed, 2012, Morgan Kaufmann.

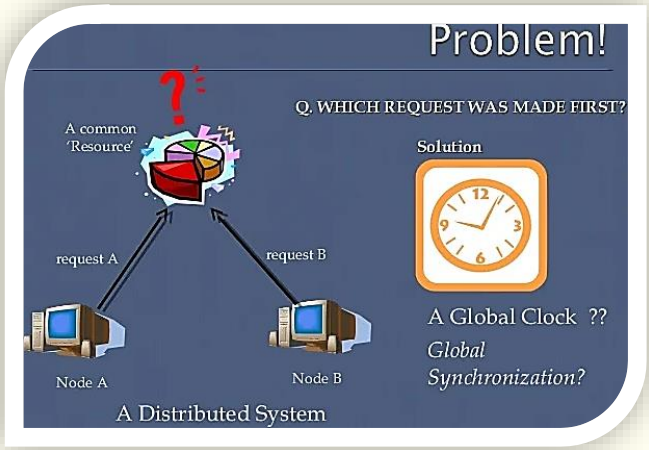


- Additional reading materials will be provided during the module.

Introduction

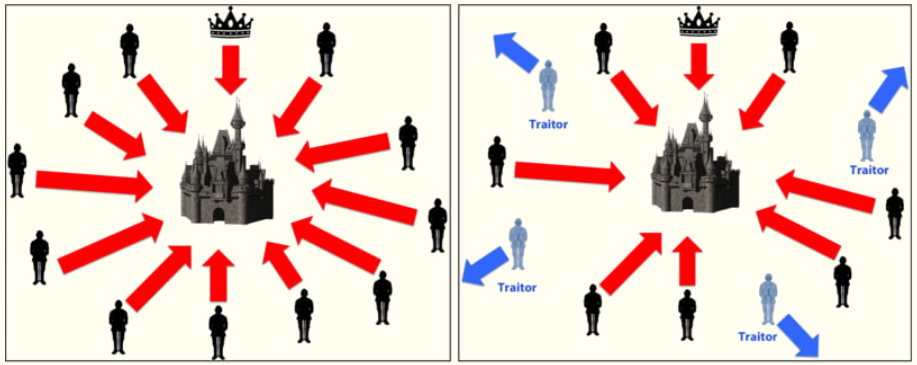
Definitions:

- Distributed System (DS) "is a collection of autonomous computing elements that appears to its users as a single coherent system" Van Steen & Tanenbaum.



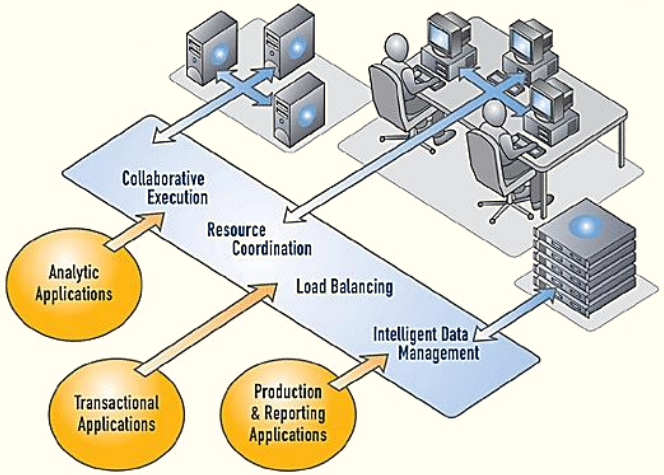
  - Autonomous computing elements (a.k.a nodes), e.g., hardware devices or software processes.
    - Preserve Independent behaviour, i.e., each node is autonomous and will thus have its own notion of time: there is no global clock, which leads to fundamental synchronization and coordination problems.

  - Single coherent system: users or applications perceive a single system ⇒ nodes need to collaborate.
    - All nodes operate the same. It doesn't matter where, when, and how interaction with the system takes place (Transparency), e.g., consider a banking system.
    - How to manage membership?
    - How to know that you are indeed communicating with an authorized member, e.g., consider the Byzantine Failure?



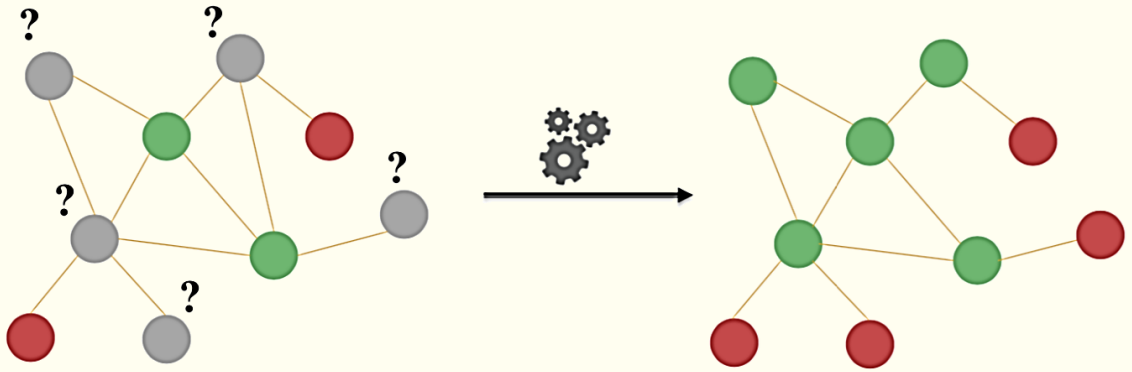
Coordinated Attack Leading to Victory | Uncoordinated Attack Leading to Defeat

- Distributed Computing (DC) refers to a software that solves a problem over distributed autonomous computers which communicate over a network.



- DC aims to achieve computational tasks more faster than using a single compute.
- It is so characterised by Remote Procedure Calls (RPC) and Remote Method Invocation (RMI) computations.
  - Nodes collaborate by message-passing.
  - A common language is a requirement, e.g., CORBA, SOAP.
  - Nodes must trust each other.

- Classification of DC systems:
  - Distributed Information Systems.
  - Distributed Pervasive Systems.
  - Middleware.



- Limitations: reliability and scalability issues, nodes and networking may fail.

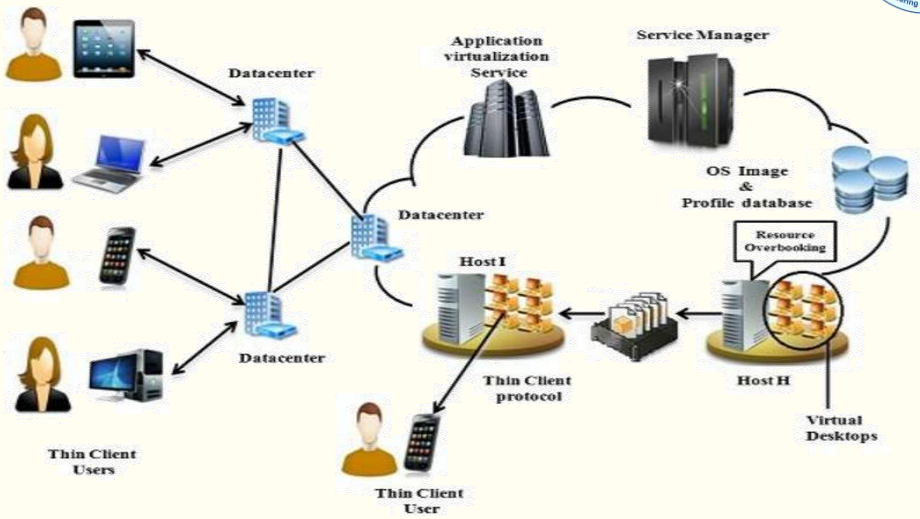
- Cloud Computing (CC) provides on demand resources/services, e.g., storage, database, analytics, software etc. over the Internet!!!

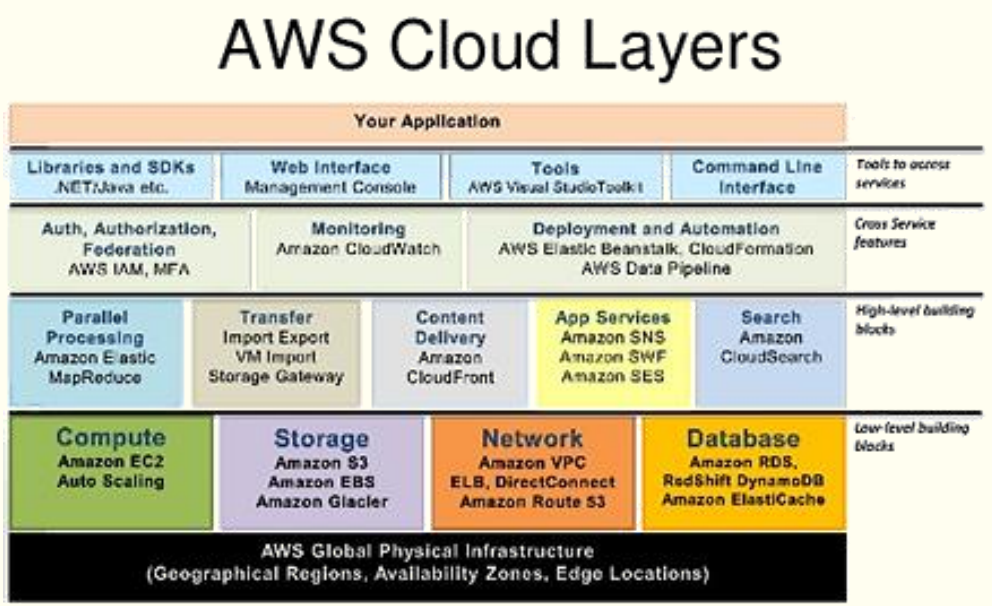
- CC implements various techniques to delivers hosted services over the Internet to cloud users/customers. It is characterised by providing a shared pool of configurable computing resources, on-demand services, and pay-per-use services, etc.

- Cloud systems are classified into different types, e.g.,
  - Public Cloud.
  - Private Cloud.
  - Community Cloud.
  - Hybrid Cloud.

- CC compromise some general requirements, e.g., being cost effective, fixable, reliable, scalable, globally accessible, secure, etc.

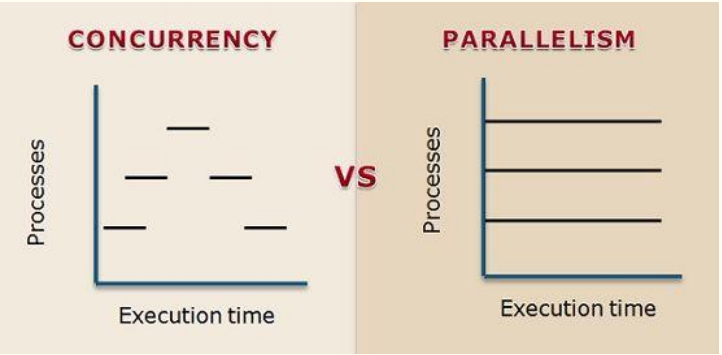
- Limitations: less control, security and privacy issues.



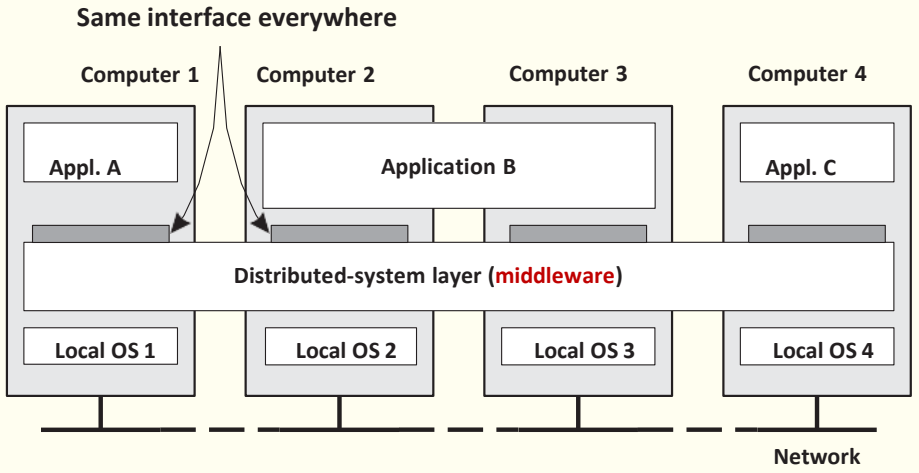


Engineering of DS:

- Intentional requirements of DS (Why):
  - Be cost-effective, via sharing data and computing resources.
  - Be high performance, via Parallelism & Concurrency.
  - Be scalable, through cloning of serving nodes (Replication).
  - Be reliable, by using Redundancy & Fault-tolerance techniques.

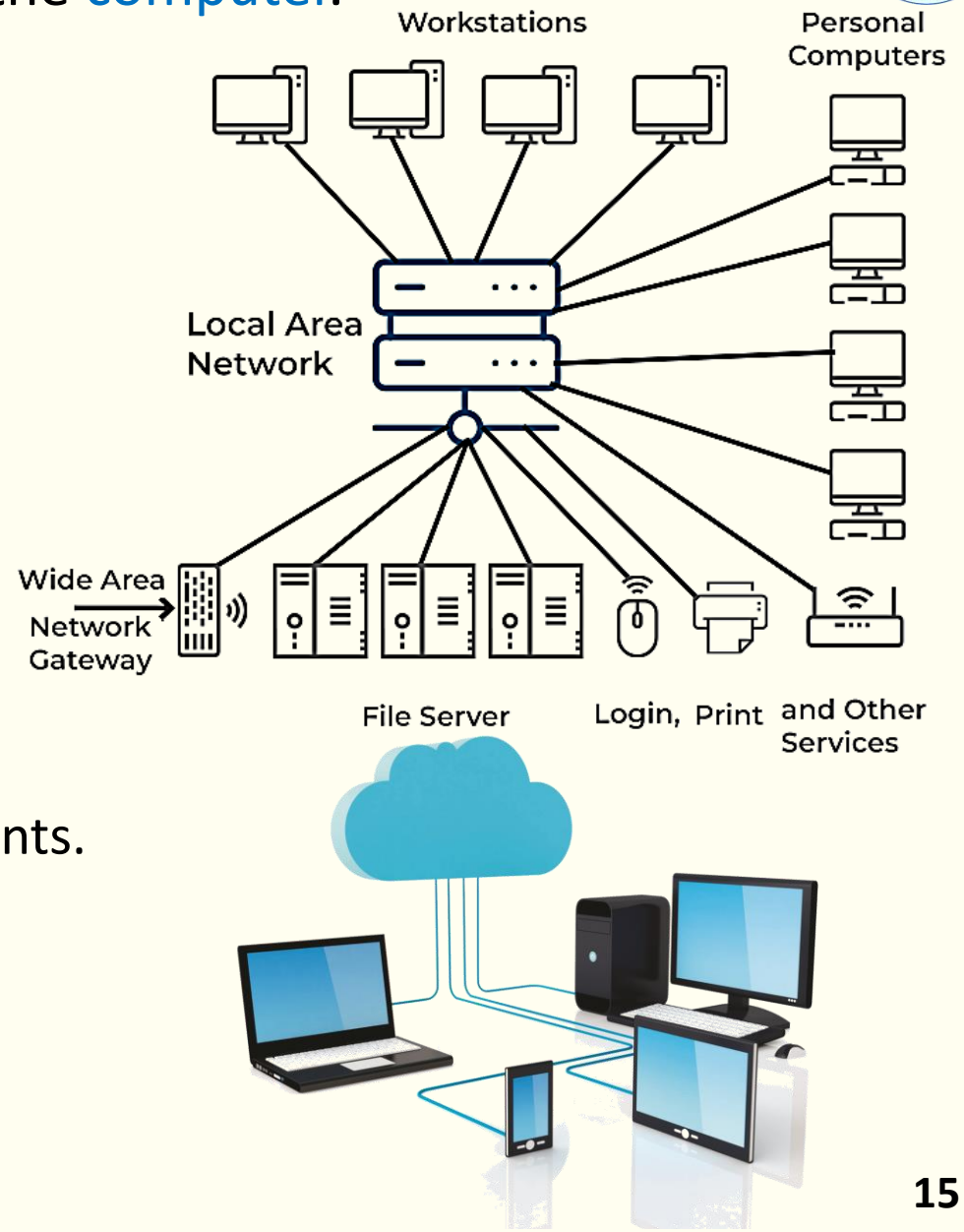


- Engineering requirements of DS (What & How):
  - Resource Sharing (Support Global Shares).
  - Transparency (Hide Complexity).
  - Openness (Use Interoperable Components).
  - Scalability (Enable Extensibility).
  - Availability (Ensure 99% up-time).
  - Maintainability (Make it easy to maintain & evolve).

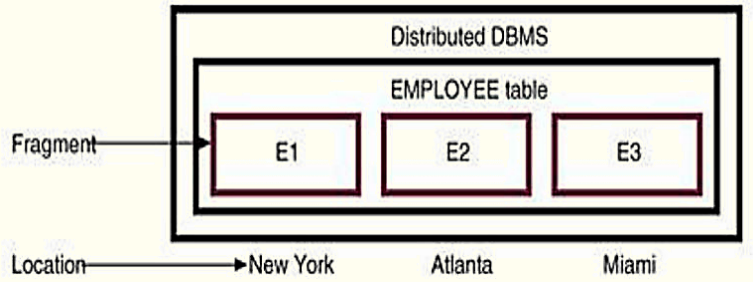


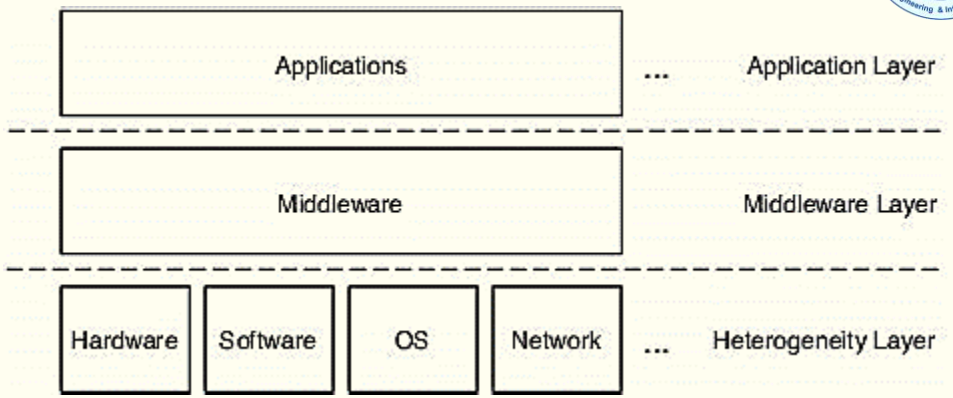
- DS requirements characterize the complications and diversity of the involved techniques, mechanisms, and approaches.

- Resource Sharing:
  - With stable resource sharing, the network becomes the computer.
  - Examples:
    - Internet connection, Printer, and File sharing.
    - Database & Database servers.
    - Peer-to-Peer (P2P) multimedia sharing.
    - Mail and web hosting servers.
    - Cloud computing and storage sharing.
  - Limitations:
    - Subject to local and remote failures.
    - May experience high latencies or delays.
    - Lack of standardization & agreement among participants.
    - Availability and formats issues.
    - Security and authentication issues.



- Distribution Transparency:
  - Examples:
    - Distributed DBMS.
    - Middleware.

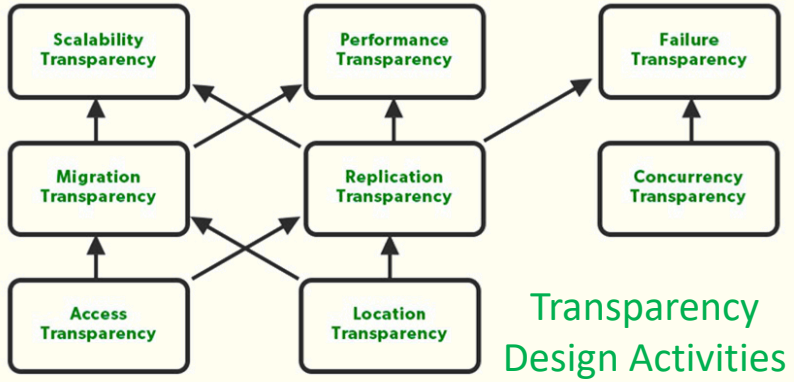




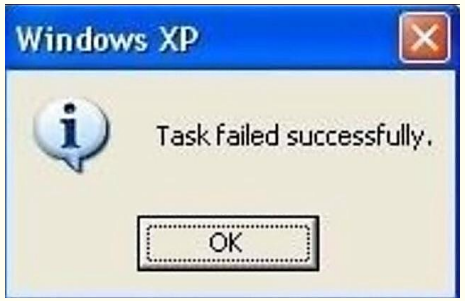
  - Transparency Types:
    - Access: hide differences in data representation and how an object is accessed.
    - Location & Migration: hide where an object is located; or hide that an object may be moved to another location while in use.
    - Replication: use an object or its replica as same object.
    - Concurrency: hide that an object may be shared by several independent users.
    - Failure: hide the failure and recovery of an object.

  - Follow the link for "*Transparency: Illusions of a Single System*".

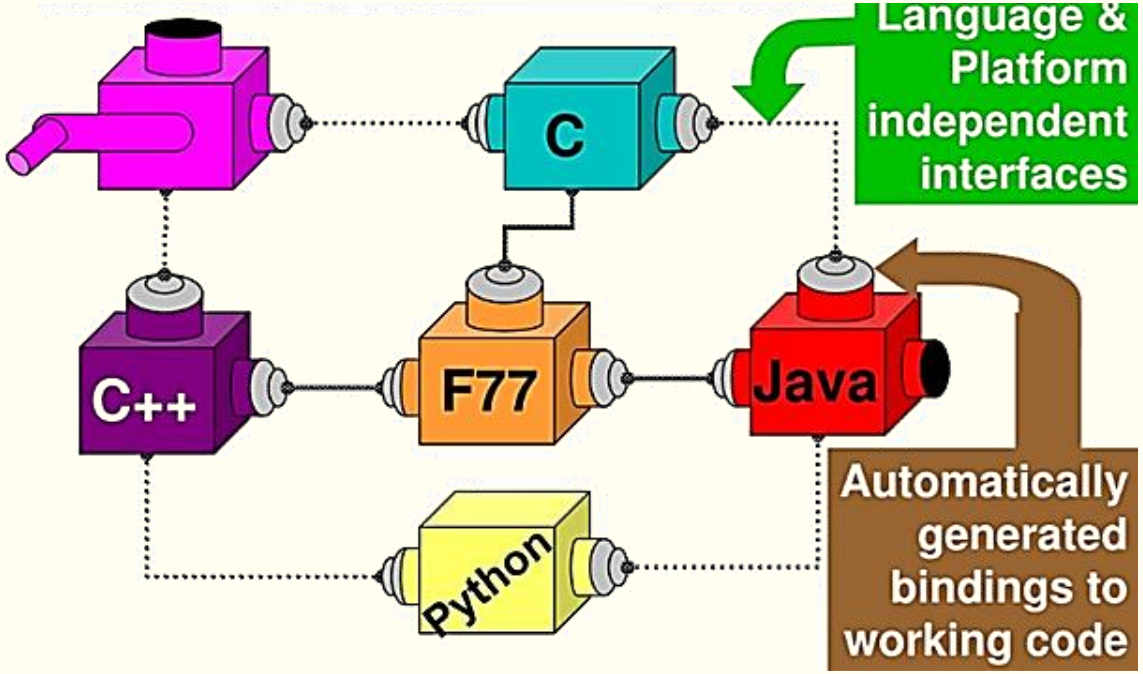
- The degree of Transparency:
  - Engineers aim at full distribution transparency, BUT,
  - There are communication latency issues and many covered failures either of networks or nodes.
    - ❖You cannot distinguish a slow computer from a failing one,
    - ❖You can never be sure that a server actually performed an operation before a crash,
    - ❖This a reality: engineers has no global view on the system.
  - Full transparency will cost performance and will expose system distribution:
    - Keeping replicas exactly up-to-date with the master takes time,
    - For fault-tolerance, write operations need immediate flushing to storage disk, meant, data may vanish due to a crash.
      - For example, a customer makes a deposit to his bank account a few seconds before the bank system crashes.
- Distribution transparency is an admirable goal, but achieving it is hard work.

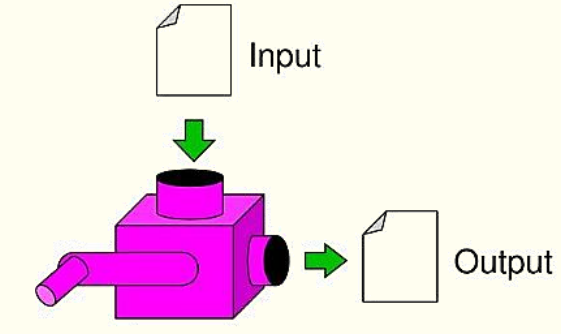


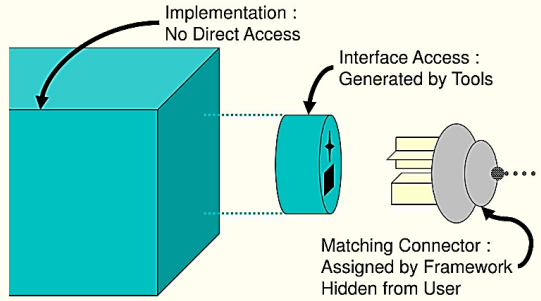
Transparency Design Activities

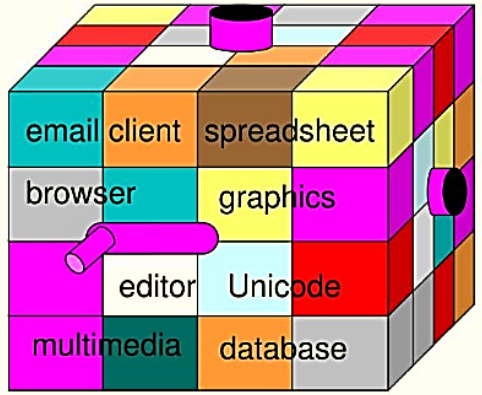


- Openness of DS:
  - Be able to interact with other open systems, irrespective of the underlying software/hardware components:
    - Systems should conform to well-defined interfaces!!!
      - Discussion: what are interfaces stereotypes?
    - Systems should easily interoperate.
    - Systems should support portability of applications.
    - Systems should be easily extensible.



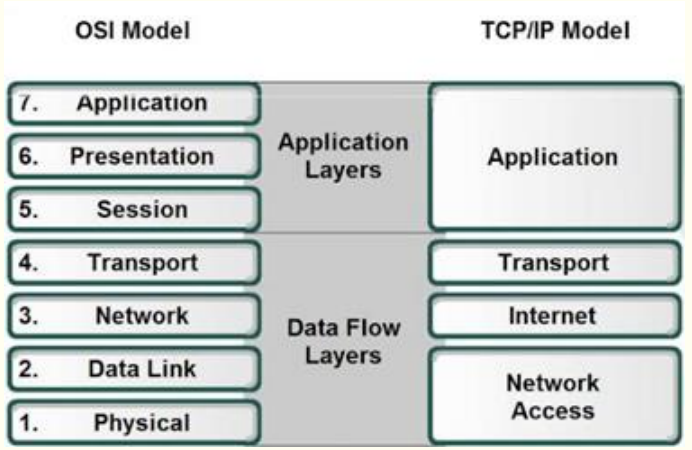






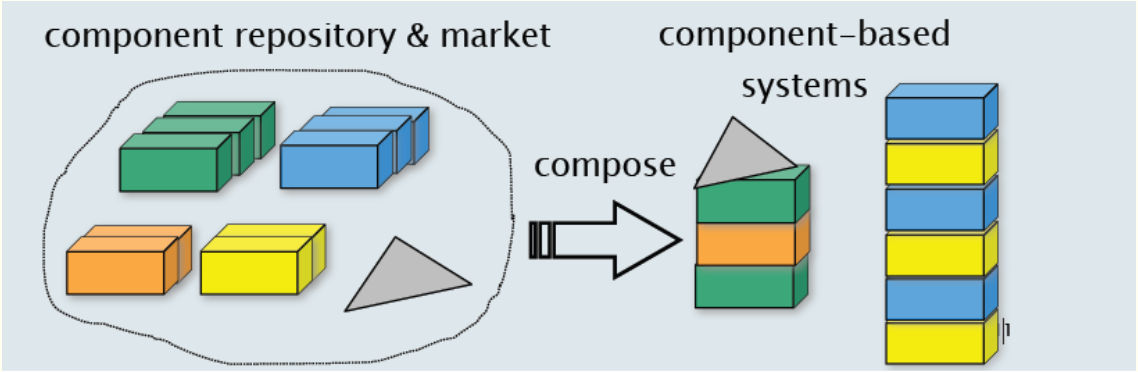
- **Interoperability across multiple languages**
- **Interoperability across multiple platforms**
- **Incremental evolution of large legacy systems (esp. w/ multiple 3rd party software)**

- Typically, system interoperability is implemented as policies and mechanisms.
- Engineers elicit for DS requirements using some questions such as:
  - What level of consistency do we require for client-cached data?
  - Which operations do we allow downloaded code to perform?
  - Which QoS to adjust for the varying bandwidth?
  - What level of security do we require for communication?

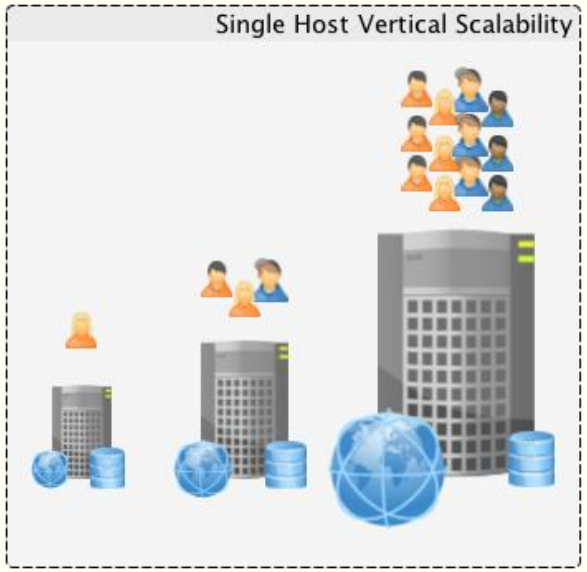


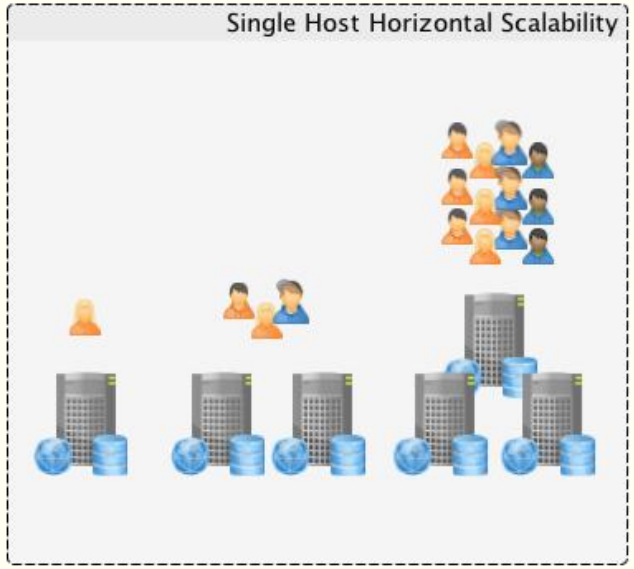
- The using of strict policies leads to many configuration parameters and complex management, thus, balance openness is needed.

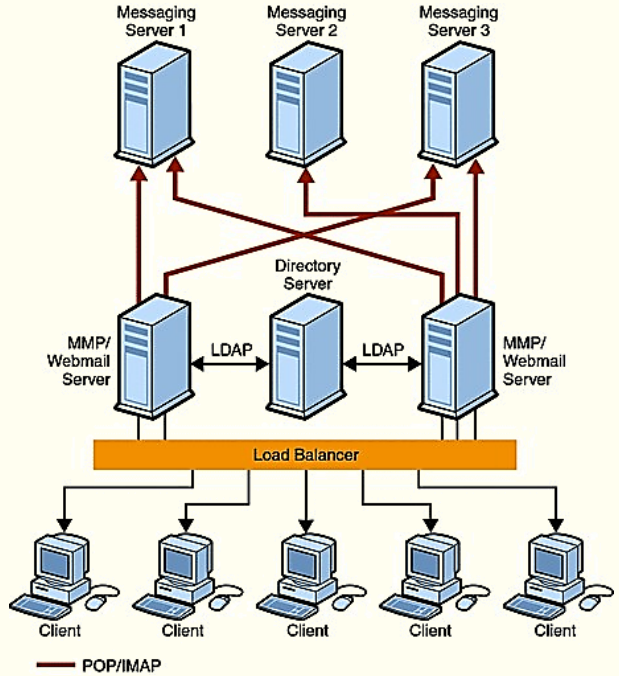
- On recent approach is to build component-based systems.



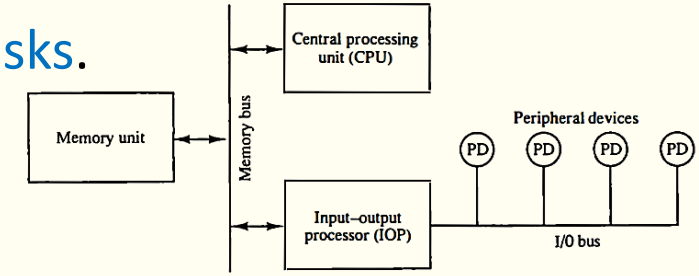
- Scale of DS:
  - Why DS actually scales?
  - Often engineers refer to an increase in size by the term scalability, which is handled by multiple powerful servers operating independently in parallel. However, there are
    - An increase in the number of users and/or processes (size scalability),
    - An increase in distance between nodes (geographical scalability),
    - An increase in number of administrations (administrative scalability),
    - An increase in functionalities or loads (load scalability),
    - An increase in using components from different vendors (heterogeneous scalability).

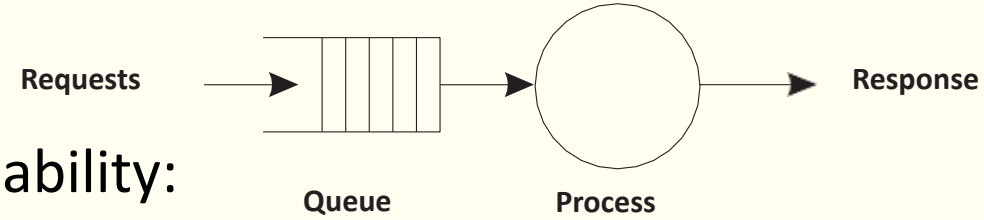




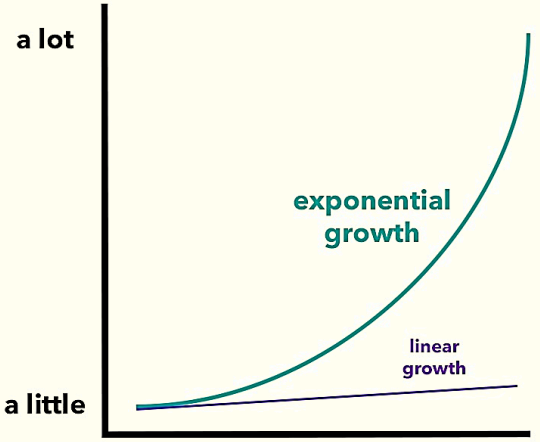


- Size scalability:
  - Main issue for centralised systems,
  - Computational capacity is limited by the CPUs,
  - Storage speed is bounded by the transfer rate between CPUs and disks.
  - Response time involves network delays and computational delays.



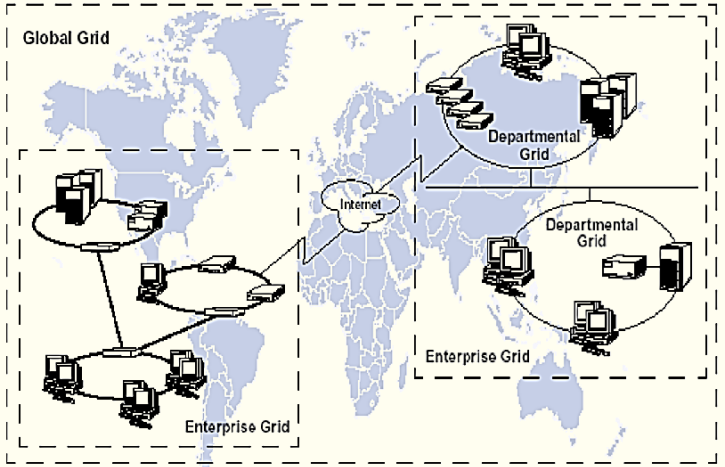


- Formal analysis for size scalability:
  - How do engineers check whether to scale up their systems or not?
  - A centralized service can be modelled as a simple queuing system:
    - The queue has infinite capacity,
    - λ is the arrival rate of requests per time unit,
    - μ is the service rate of requests per time unit (processing capacity).
    - $\overline{N}$ is average number of request in the system $\Rightarrow \overline{N} = \frac{\lambda}{\mu - \lambda}$.
    - S is average time to process λ requests in the system $\Rightarrow S = \frac{1}{\mu - \lambda}$.

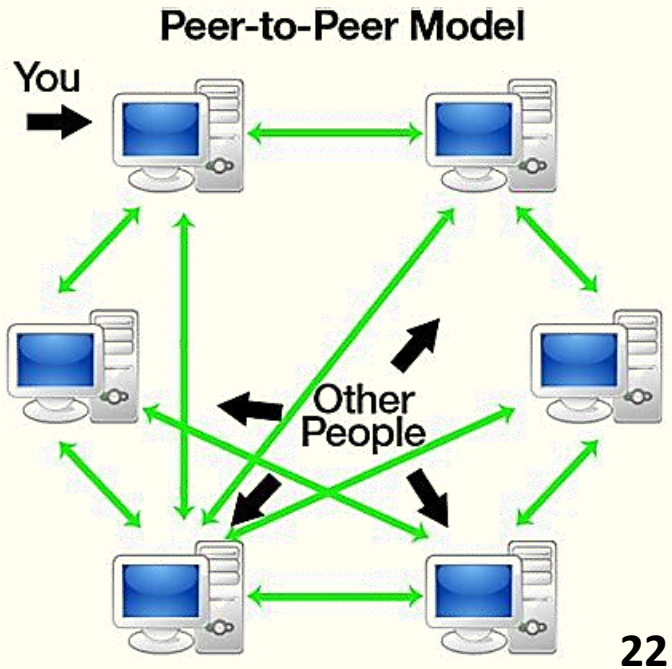


- As the number of requests increases linearly the service/response time increases exponentially.

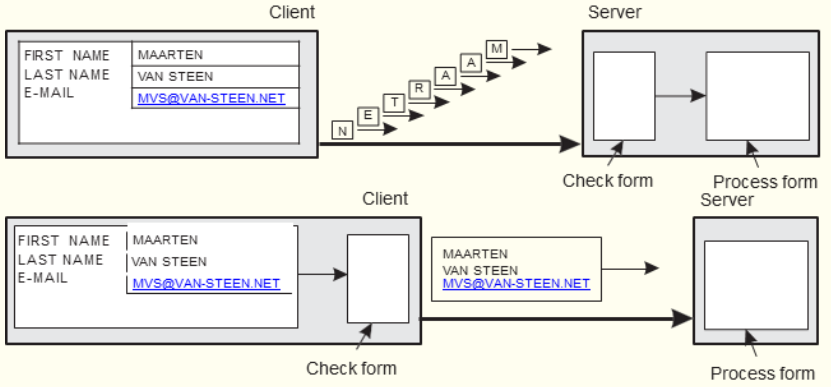
- Geographical Scalability:
  - Go from LANs to WANs is not simple, issues like synchronous client-server interactions may be prohibited due to network latency and reliability.
  - Requires global naming services to support scalability and global broadcasting.
  - Has special issues in CC (Edge & Grid).



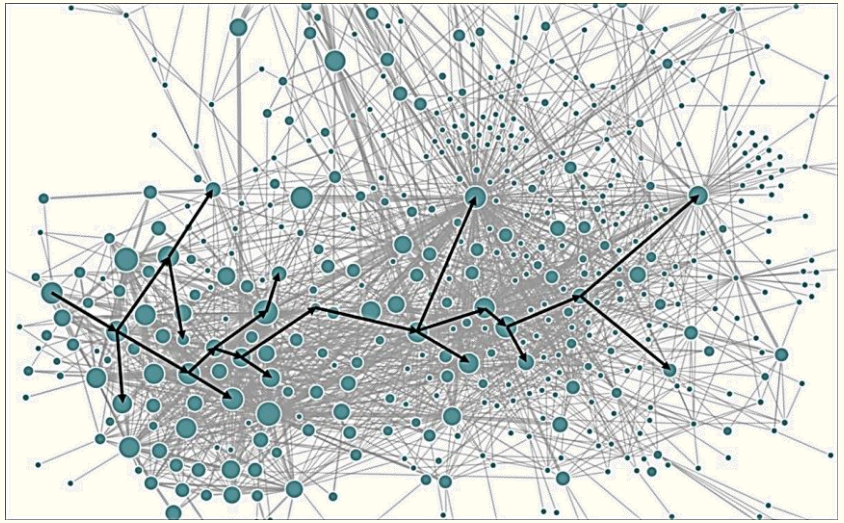
- Administrative Scalability:
  - A typical result of sharing expensive resources between different systems (usually centralised).
  - Involves conflicting policies concerning usage, payment, management, and security.
    - Example: the control, manage, and use of a shared radio telescope constructed as large-scale sensor network.
  - It, however, is not present in decentralised systems (Why?), i.e., peer-to-peer networks:
    - File-sharing systems (BitTorrent).
    - Telephony (Skype).
    - Assisted audio streaming (Spotify).
    - Cryptocurrencies (Bitcoins).



- Scalability Engineering:
  - Use asynchronous communication to overcome communication latencies.
  - Use independent services for incoming responses.
  - Move some computations to the client side.
  - Implement replication and caching, e.g.,
    file servers, databases, mirrored sites, and web caches.



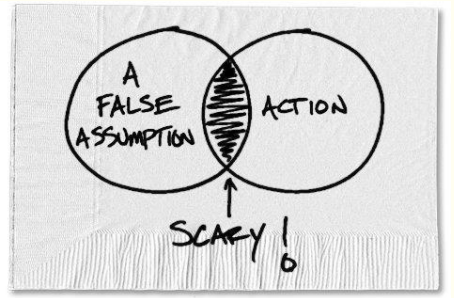
- Limitations:
  - One size doesn't fit all.
  - Applying replication is problematic:
    - Inconsistent copies, despite the frequent updates.
    - Requires global agreement and synchronisation
      on each modification.

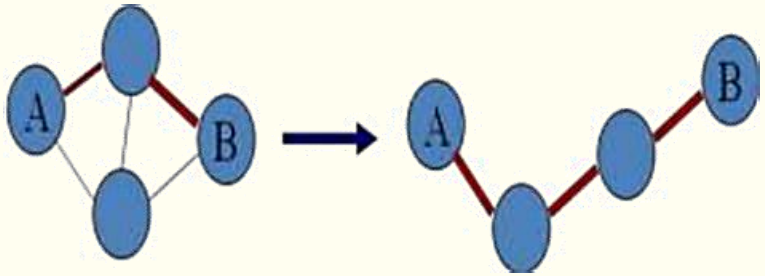


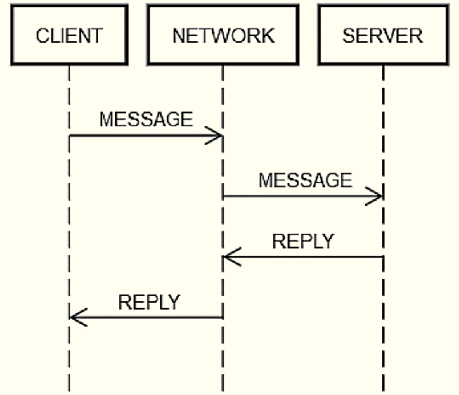
- Notes:
  - Global synchronisation blocks large-scale solutions.
  - Tolerating consistency can reduce global synchronisation but this is an application dependent.

Pitfalls in DS Engineering:

- Often, engineers make systems heavily complex and error-borne by applying false assumptions. Here is the Common false and hidden assumptions:
  - The 8 fallacies, by L. Peter Deutsch, Sun Microsystems, 2015:
    1. The network is reliable,
    2. The network is secure,
    3. The network is homogeneous,
    4. The topology does not change,
    5. Latency is zero,
    6. Transport cost is zero,
    7. Bandwidth is infinite.
    8. There is one administrator.



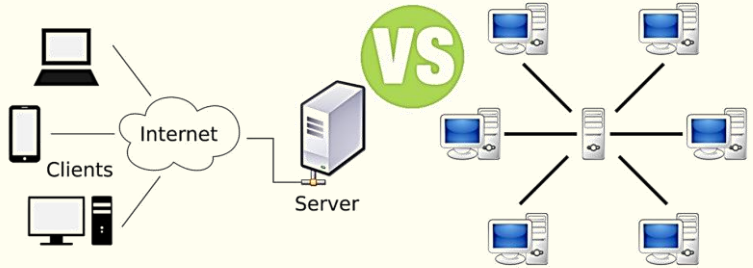


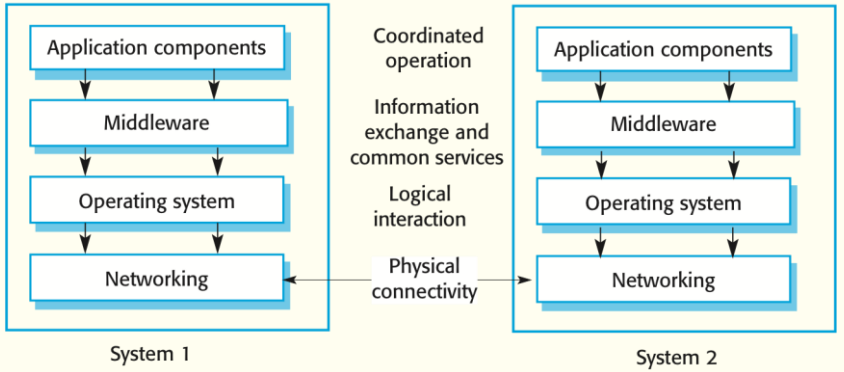


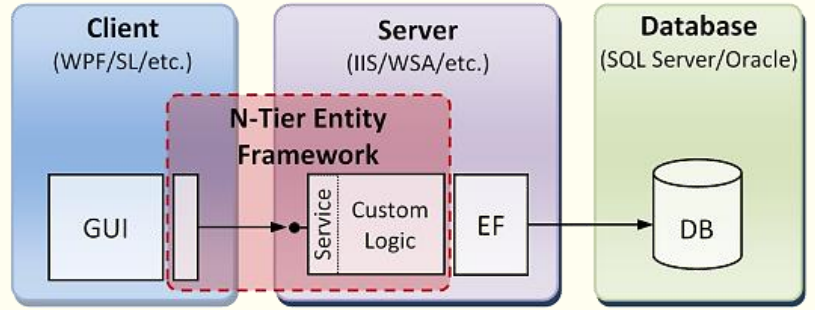
  - Additional fallacies:
    - All failures have been considered in the development stage.
    - The server hardly fails, it is configured to restart itself no failure.
    - Engineers always have a global view of the distributed system.
    - A client can distinguish network failure from server failure.
    - All computing elements/nodes are trustworthy.
    - Using formal analysis and modelling is time-consuming and expensive, so, just skip it.

Classification of DSs:

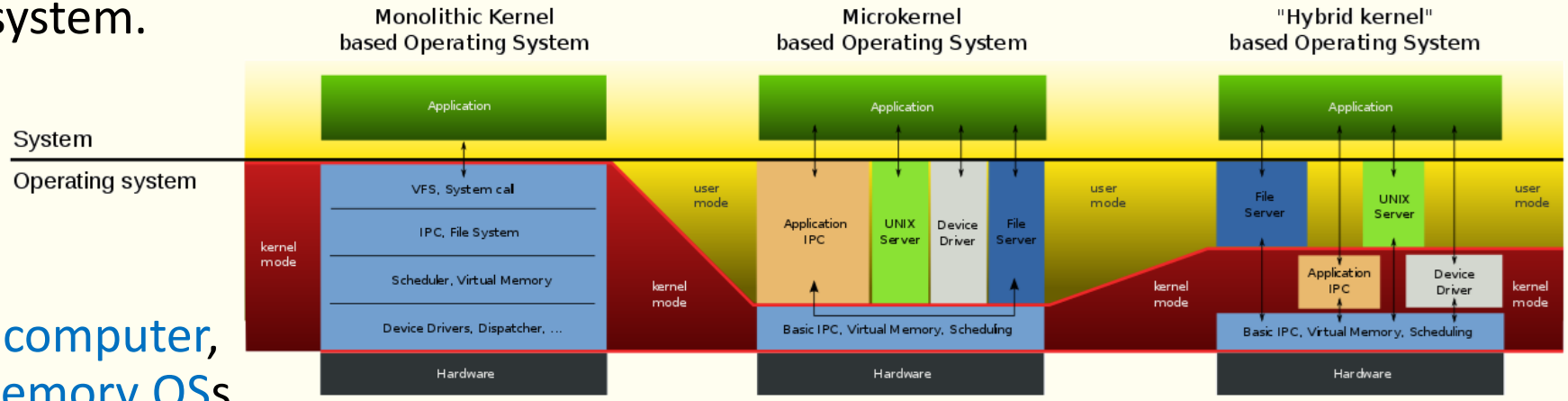
- Based on the architecture:
  - Client/Server systems (Centralised).
  - Peer-to-Peer systems (Decentralised).
  - Middleware systems.
  - Three-tier/N-tier systems.
- Based on the distribution of computing:
  - Operation Systems (OSs).
    - Distributed Operating Systems (DOS).
    - Network Operating Systems (NOS).
  - High-Performance Computing systems (HPC).
    - Cluster systems.
    - Grid systems.
    - Cloud systems.
  - Distributed information systems (DIS).
    - Distributed transaction processing.
    - Enterprise Application Integration (EAI).
  - Distributed pervasive systems.
    - Ubiquitous systems.
    - Mobile systems.
    - Sensor systems.



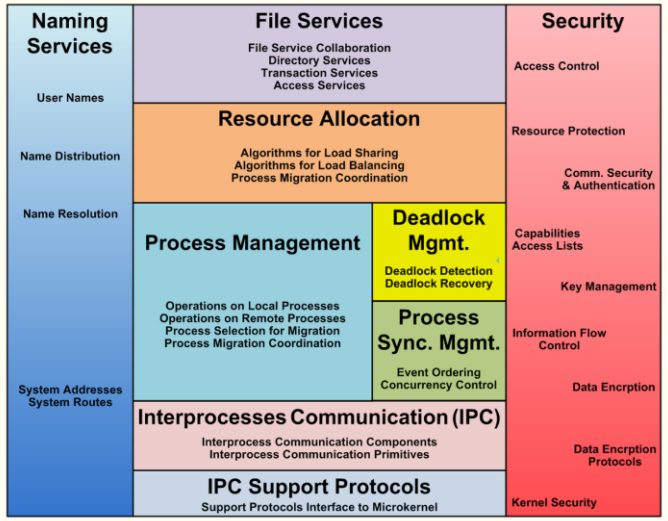




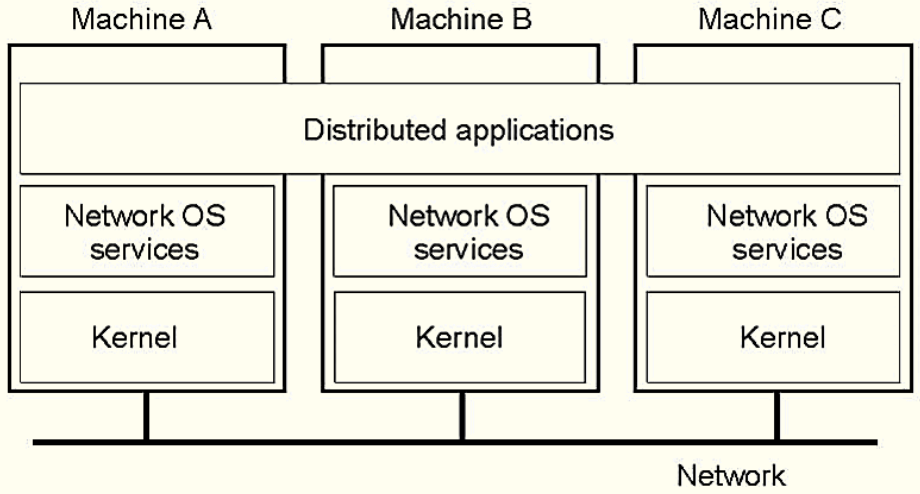
- Distributed Operating System (DOS):
  - DOS is a system software that comprises a collection of independent, networked, and physically separate computational elements/nodes (Wikipedia).
  - Jobs/loads are serviced by multiple CPUs. Each node holds a specific software subset of the global aggregate operating system.

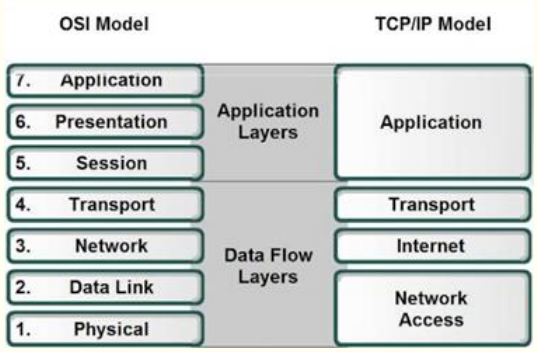


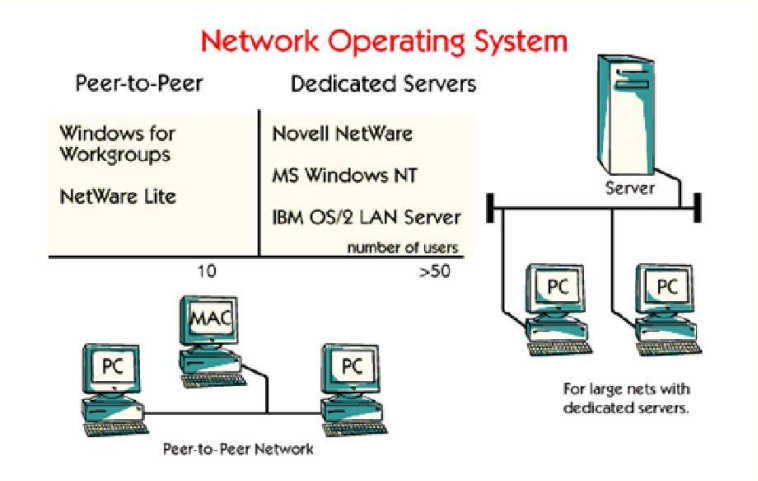
  - DOSs Types: Multiprocessor, Multicomputer, Distributed Shared Memory OSs.
  - DOS Requirements:
    - Performance & Scalability (Intentional).
    - Distribution Management (Complexity Issues),
    - Transparency: Location, Migration, Replication, Concurrency, Scale,
    - Openness (Interoperability),
    - Reliability & Availability (Fault-tolerance),
    - Synchronisation (Atomicity, Consistency, Deadlock),
    - Flexibility (Services, Components).



- Network Operating System (NOS):
  - NOS is a specialised OS for a network device such as a router, switch or firewall (Wikipedia).
  - NOS enable computers nodes to participate and share files and printers access within the network.
  - Proprietary & Open Source.
  - Client/Server & Peer-to-Peer.
  - NOS Requirements:
    - Resource Sharing & Cost Saving (Intentional).
    - Transparency: Location, Platform.
    - Openness (Interoperability),
    - Reliability & Availability (Fault-tolerance),
    - Synchronisation (Consistency).
    - Security.

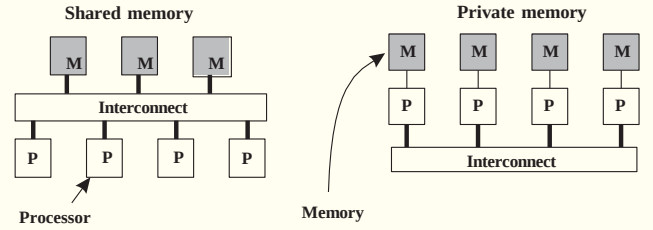




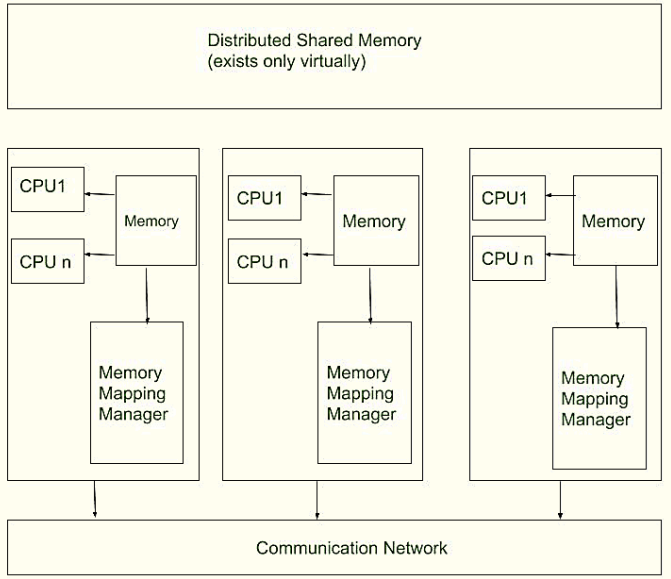


- Limitations:
  - DOS, complex & non-flexible,
  - NOS, not a coherent computer,
- For the best of both, consider the Middleware approach.

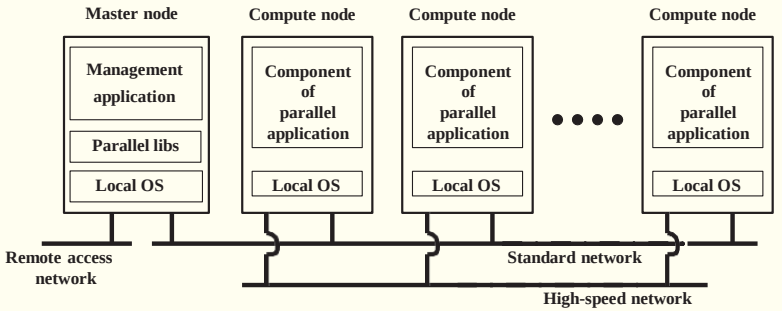
- High-performance distributed computing (HPC):
  - Typically is characterised by parallel computing, multiprocessor/multicore, and high-end multicomputer.
  - Typically, multiple processes/threads run at the same time;
  - All threads have access to shared data; access is controlled via synchronisation mechanisms (e.g., Semaphores).



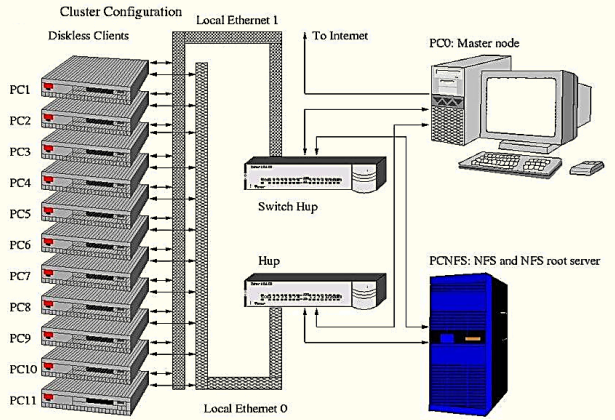
  - However, this approach does not easily scale due to hardware limitation (i.e., Mainboards have limited sockets for CPUs & RAMs) .
  - A solution was to use shared-memory model on top of multicomputer.
  - The shared-memory model suffers from synchronisation (i.e., concurrent access) and message-passing (i.e., latency) issues.
  - Later, the Distributed Shared-Memory (DSM) model is introduced. In the DSM, a processor can address a memory location as if it were local memory (Virtual-Memory Technique). However, DSM has been abandoned due to performance issues.



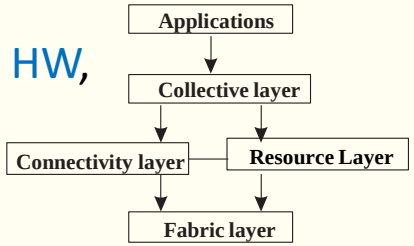
  - Implementations of the HPC:
    - Cluster computing.
    - Grid computing.
    - Cloud computing.

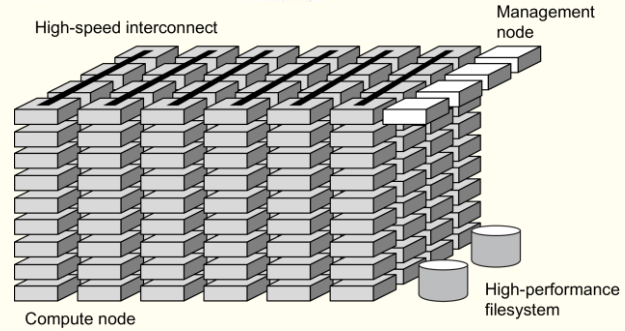


- Cluster computing:
  - A group of high-end computers/servers connected via a LAN,
  - Homogeneous, nodes use the same OS (i.e., DOS) and have identical hardware, There are a single managing node.

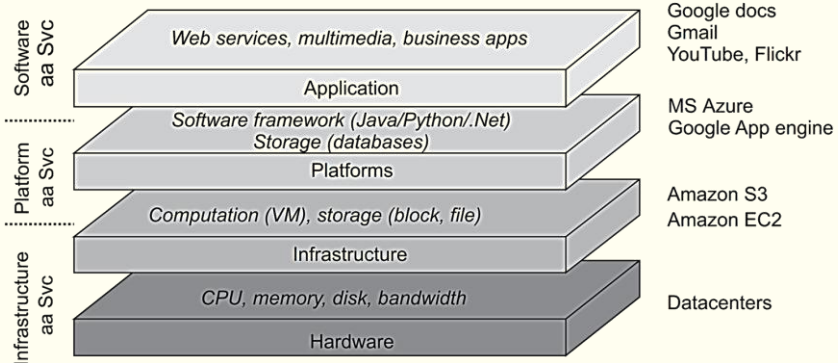


- Grid computing:
  - Heterogeneous, lots of nodes of different OSs and hardware,
  - Dispersed across several organizations and may span a wide-area network,
  - Architecture for grid computing:
    - Fabric layer provides interfaces to local resources, i.e., OS & HW,
    - Connectivity layer runs communication protocols,
    - Resource layer manages a single resource,
    - Collective layer handles multiple resources, e.g., discovery, scheduling, & replication,
    - Application layer is actual grid applications in a single organisation.



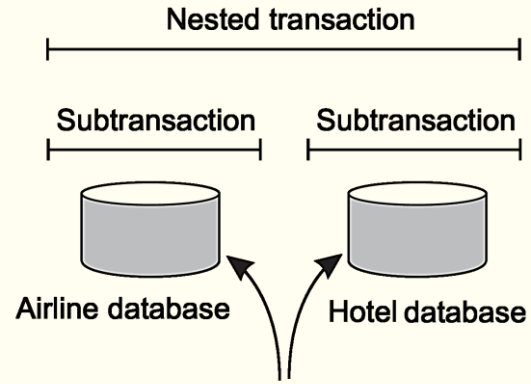


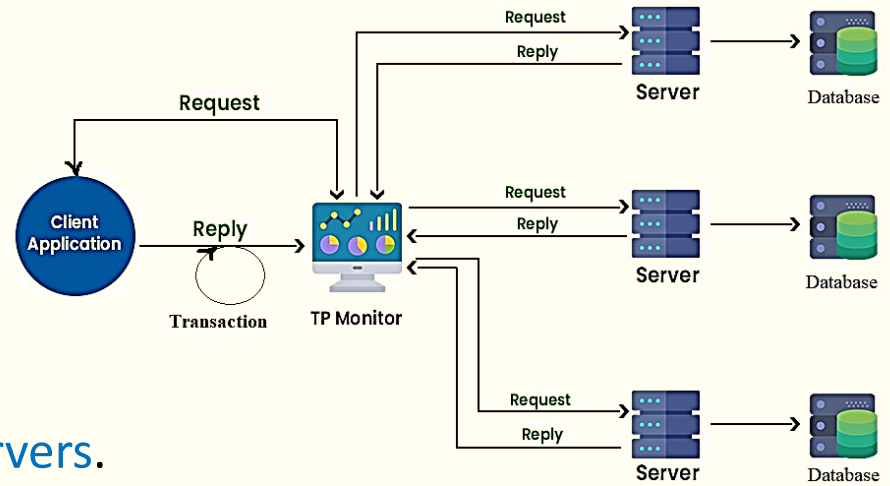
- Cloud computing:
  - General architecture:
    - Hardware layer includes processors, routers, and power HW.
    - Infrastructure layer deploys virtualisation techniques.
    - Platform layer provides higher-level abstractions for storage & so (i.e., Middleware).
    - Application layer.

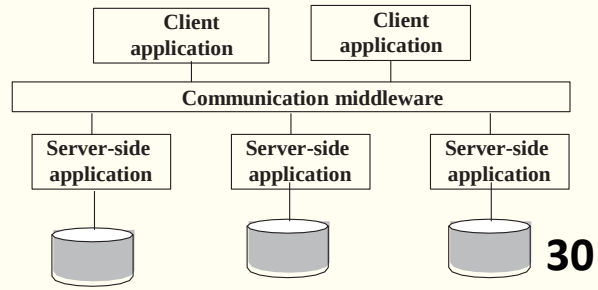


- Discussion: what engineering requirements do HPC systems need?

- Distributed information systems (DIS):
  - At present, organisations management relies on many networked applications, e.g., DBMS, CRM, HRM, FMIS, … etc.
  - DIS diversity makes achieving interoperability painful!!!
  - But, Integration in the DIS is needed because a single transaction may execute over many applications/machines.
  - Distributed transaction processing:
    - Transaction Issue: all-or-nothing,
    - Atomic: happens indivisibly,
    - Consistent: does not violate system invariants,
    - Isolated: not mutual interference,
    - Durable: commit means changes are permanent,
    - TPM: Transaction Processing Monitor.
  - Enterprise Application Integration (EAI):
    - Data involved in a transaction is distributed across several servers.
    - A TP Monitor coordinates the execution of a transaction (Middleware).
    - Middleware offers facilities for integration: i.e., Remote Procedure Call (RPC)! & Message Oriented Middleware (MOM).

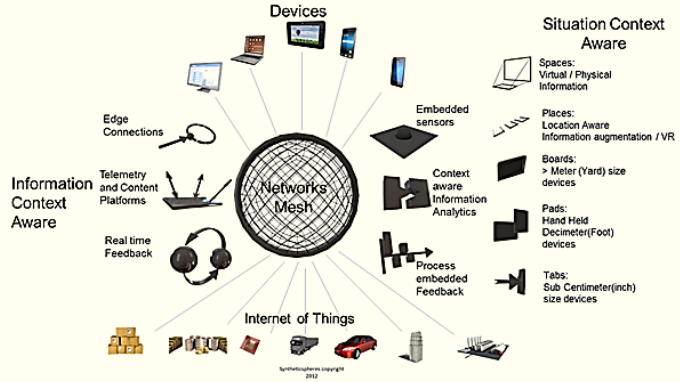


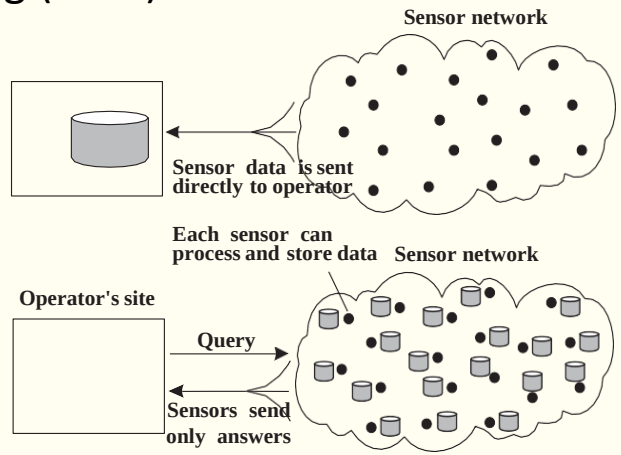




- Distributed pervasive systems:
  - Systems in which nodes are small, mobile, embedded in a larger system, and naturally blend into the user's environment.
  - The distinction is the usage of wireless networks, sensors , actuators, and IoT.
  - Ubiquitous computing systems:
    - Distribution: devices are networked, distributed, and accessible transparently.
    - Interaction: interaction between users and devices is appreciable & continuous.
    - Context awareness: system is aware of a user's context.
    - Autonomy: devices operate autonomously and are self-managed.
    - Intelligence: system handles a wide range of dynamic actions and interactions.
  - Mobile computing systems:
    - Consider systems for smartphones, tablets, GPS devices … etc.
    - They are mobile, location changes over time, thus services change too,
    - Got issues like connectivity, reachability, and discoverability… etc.
    - Known architecture: Mobile Edge Computing (MEC) and Mobile Cloud Computing (MCC).
  - Sensor computing systems:
    - A collaborative system that senses and actuators.
    - Consists of many (10s-1000s) simple nodes.
    - Data aggregation either centralised or distributed.
    - Known issues like power, failure, connectivity.
    - Decentralised dissemination techniques are used for fault-tolerance.

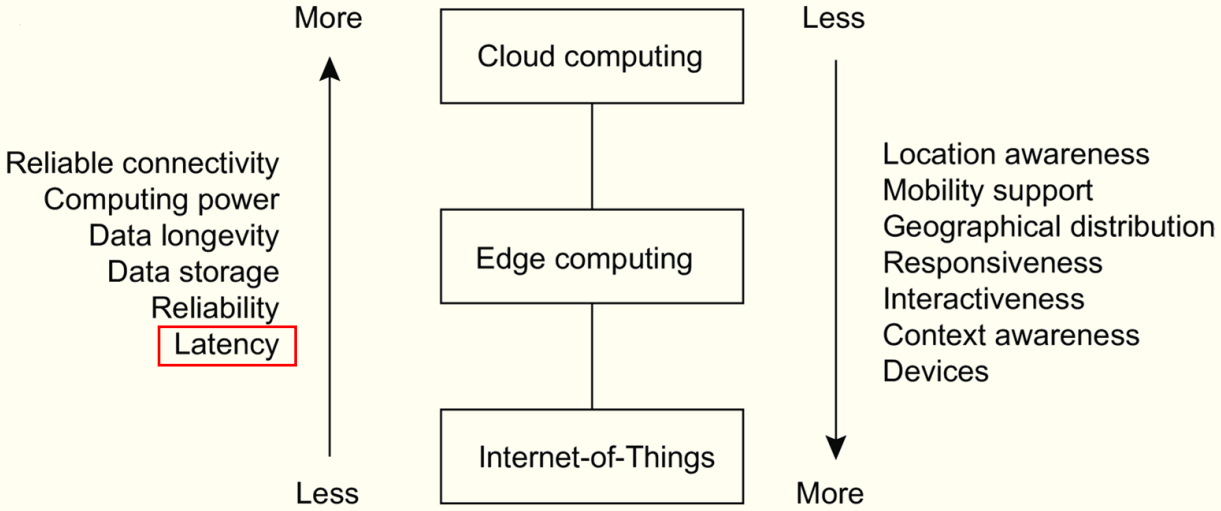
Ubiquitous Systems

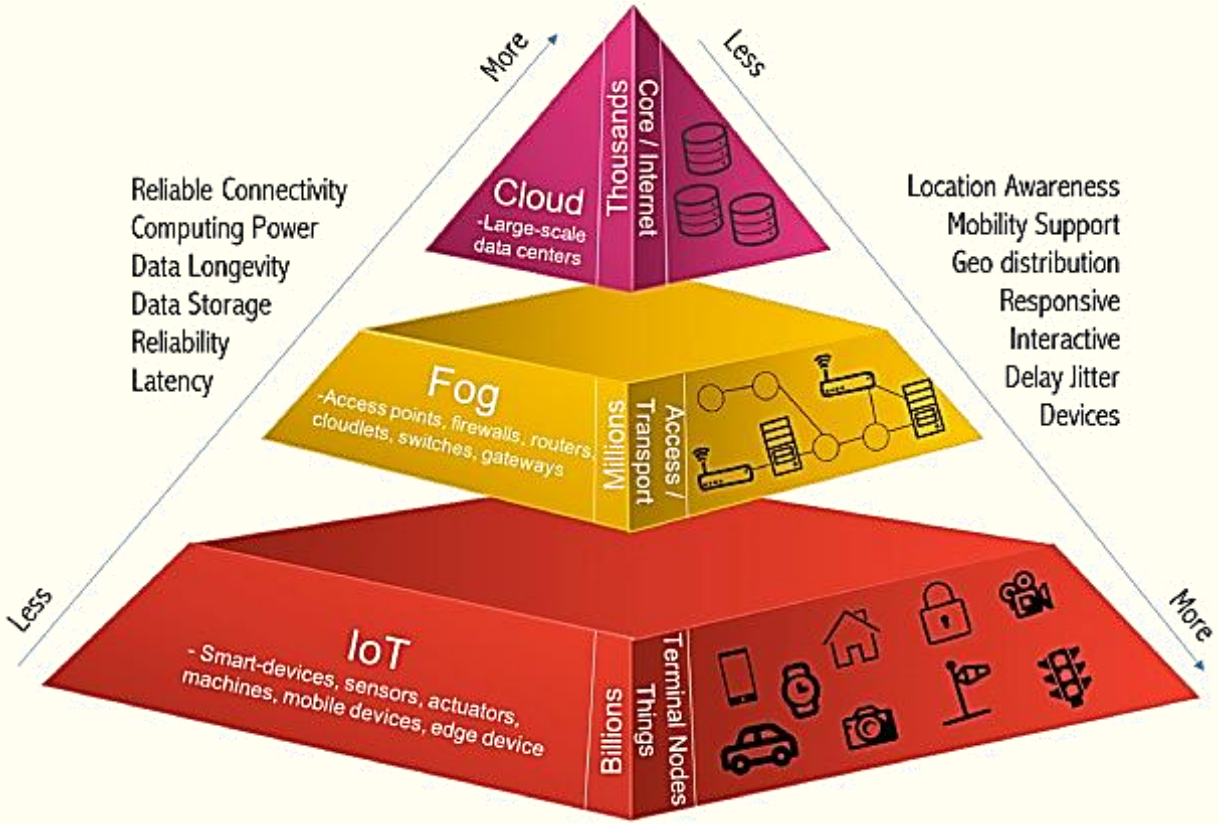




- Discussion: what requirements does pervasive systems engineering need?

- A hierarchical view from clouds to devices:
  - Clearly, DSs span a wide diverse range of networked computer systems, typically connected through a LAN or recently WAN (i.e., IoT).
  - Higher up the hierarchy, the qualities improve with a less number of computers;
  - Lower in the hierarchy, there are location-related and latency issues. Also, the number of devices increases.



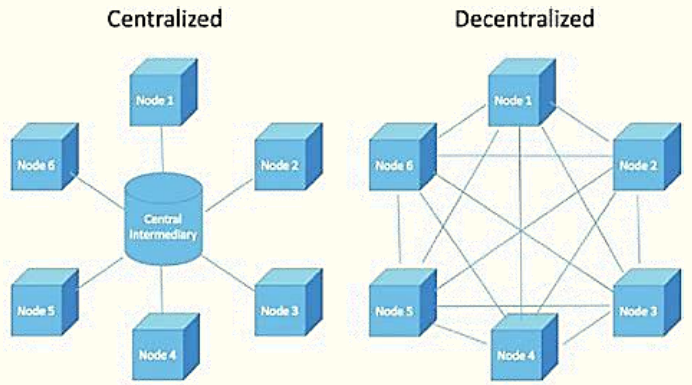


Yousefpour A., et al. All One Needs to Know About Fog Computing and Related Edge Computing Paradigms: A Complete Survey.

Distributed Systems Architectures

Architectural styles:

- A DS is built of interconnected computing units (i.e. services & components). Units are meant to be replaceable.



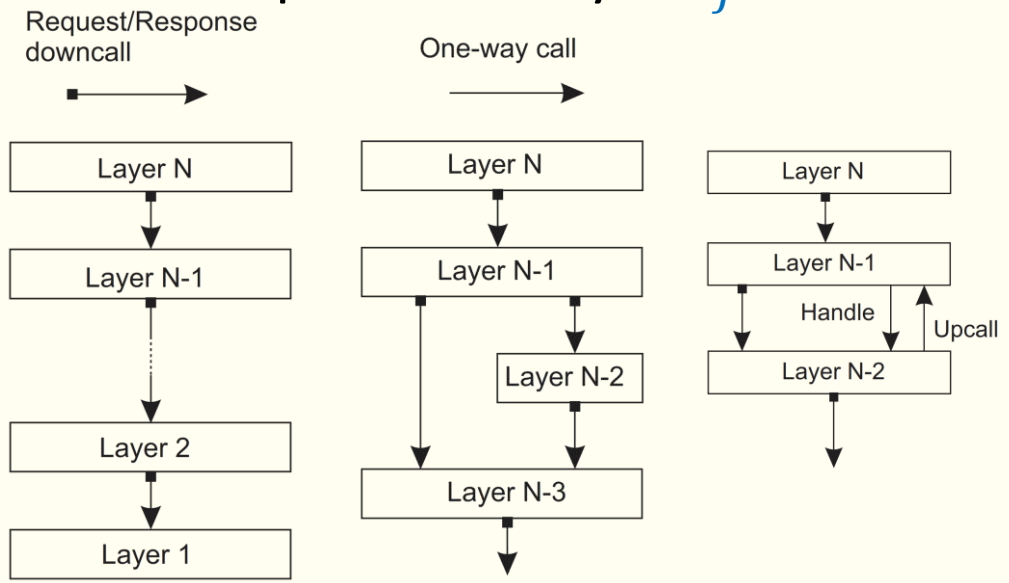
- A DS style is formulated in terms of:
  - components with well-defined interface,
  - the way that components are connected to each other, (i.e. Connectors),
  - the data exchanged between components,
  - how these components and connectors are jointly configured into a system.

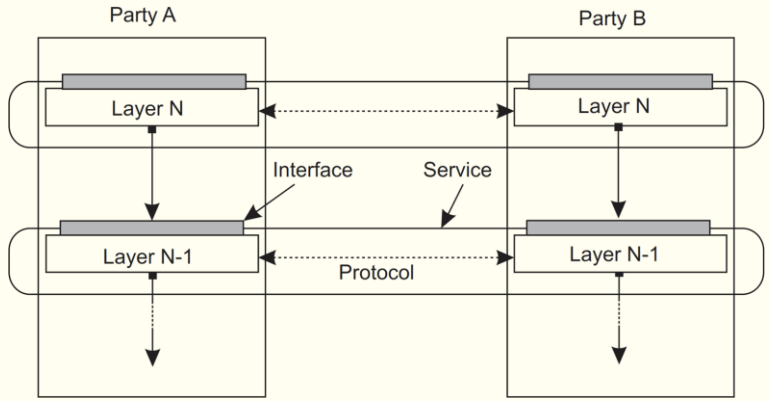
- Styles:
  - Layered & Multitiered Architectures.
  - Service-oriented Architectures (SOAs).
  - Publish-subscribe Architectures.
  - Middleware Architectures.
  - Symmetrically DSs Architectures.
  - Hybrid System Architectures.

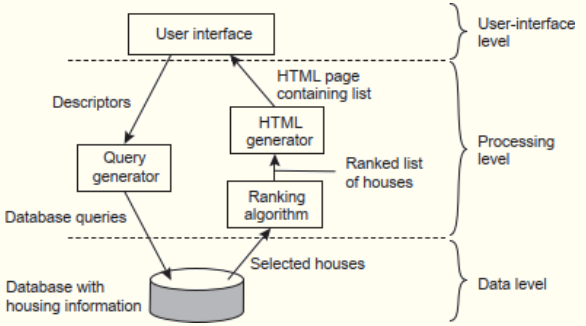
- Discussion: Is adopting the right architecture crucial for the success of large software development?

Layered Architectures:

- Simply, systems units are organised into logical layers where a component at layer L_j can make a down call to a component at L_i, with $i < j$.
- Layered communication protocols:
  - Connect parties via protocol stack,
  - Characterised by interfaces and services,
  - e.g. TCP/IP, HTTP, Web systems …
- Application layering:
  - The Interface Layer consists of input/output elements, and elementary data manipulations.
  - The Processing Layer handles business logic and major data manipulations.
  - The Data Layer is the data storge.
- Drawbacks:
  - Strong dependency among layers.

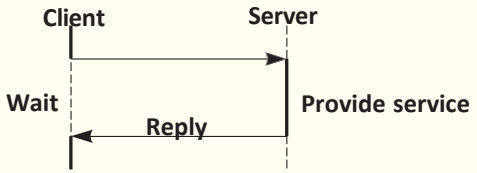


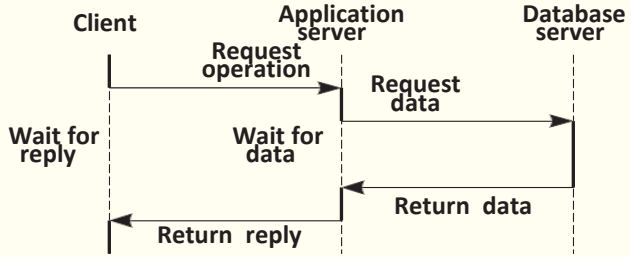


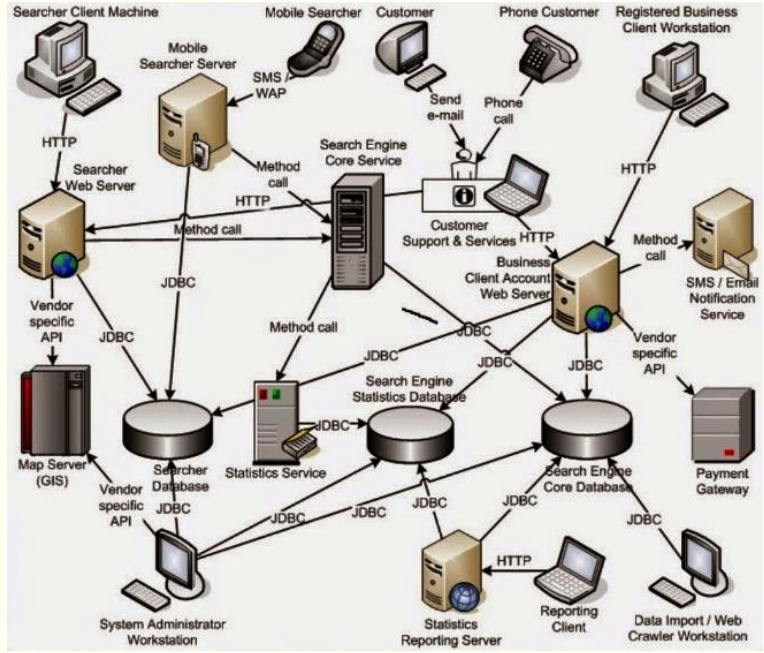


Layered-system (Multitiered) Architectures:

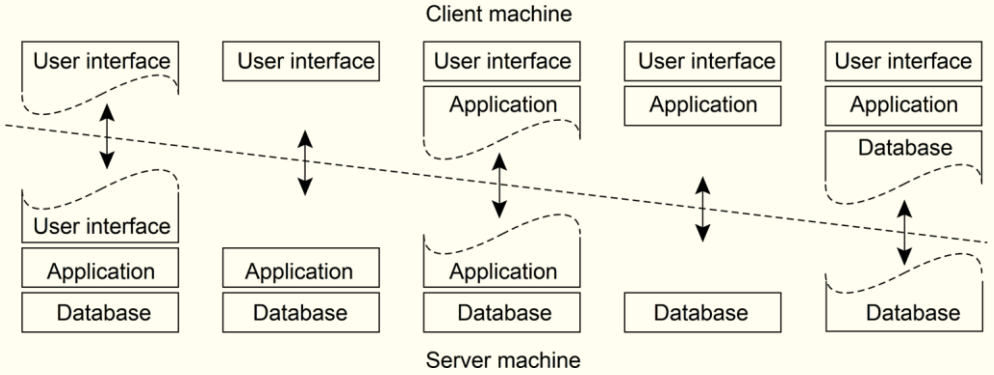
- This style is classified by considering where software components are located.
- Client/Server Architecture:
  - There are processes offering services (servers) and processes that use services (clients),
  - Interaction follows request/reply model with a connectionless or connection-oriented communication.
  - Distinction between client and server is problematic (Why?).







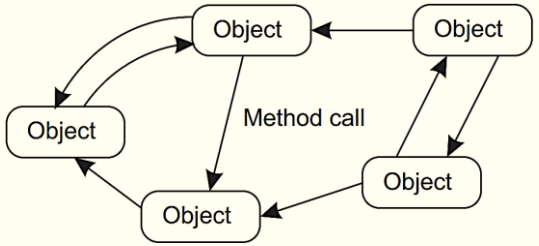
- Multi-tier Architecture:
  - Tiers number impacts performance.
  - It can have various organisations.
  - Example: Computing Terminals, NFS, Web.



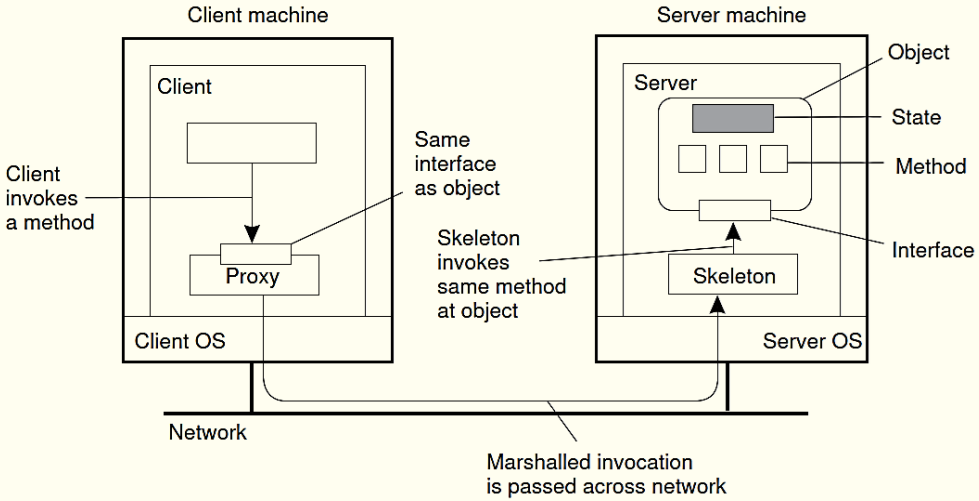
- Discussion : what factors to consider in choosing a multi-tier style?

Service-oriented Architectures (SOAs):

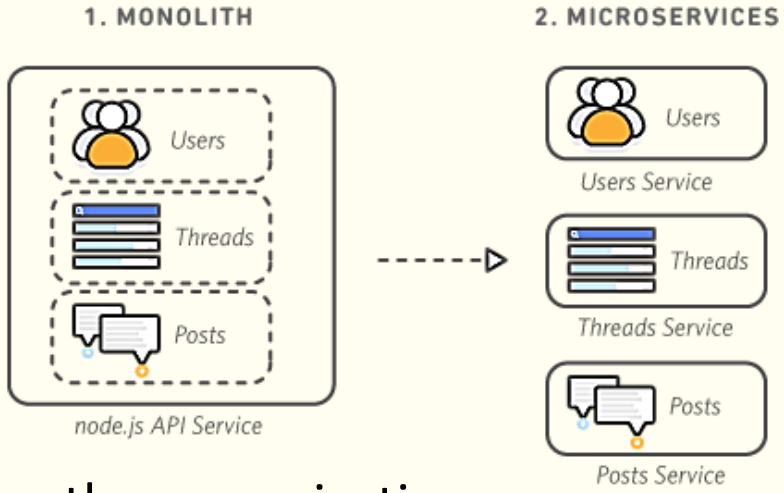
- More loose organisation into a collection of separate, independent entities (Objects, Components, Services).



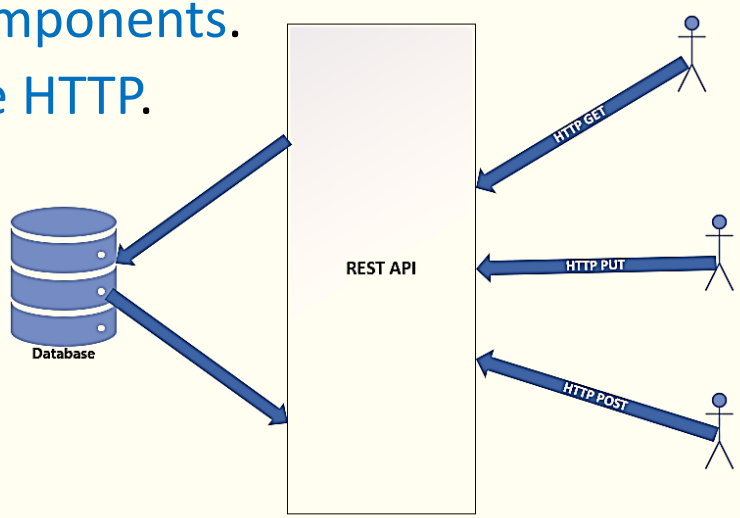
- Object-based style:
  - Distributed Independent objects connected through a procedure-call.
  - Objects encapsulates data and offer methods without revealing the internal implementation.
  - At client-side, to bind a remote object:
    - Load Proxy (Stub) object to memory,
    - Proxy object packs remote invocations (Requests).
  - At server-side, a skeleton object is always up,
    - Skeleton object unpack requests, pass them to server objects, packs and sends responses.



- Microservice style:
  - In a service-oriented architecture, a DS is constructed as a composition of many services.
  - A service is a self-contained entity.
  - Provides a specific functionality (service).
  - Can be queried and access remotely, but has complexity and integration issues.
  - e.g., e-shops: interface, orders, & database, Also, payment & delivery; which typically are services belonging to other organisations.



- Resource-based style:
  - A collection of resources that are individually managed by components.
  - Stateless, fire-and-forget client/server entities with cacheable HTTP.
  - It is widely used for the Web and is known as Representational State Transfer (REST).
  - Single naming scheme, uniform interfaces, and self-described messages.
  - Simple, but stateless.
  - Example: sharing a cloud file using a URI link.

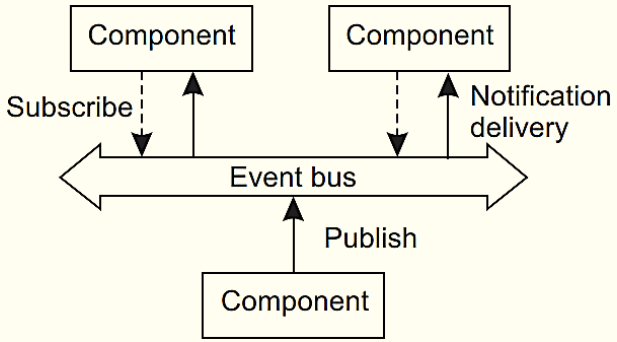


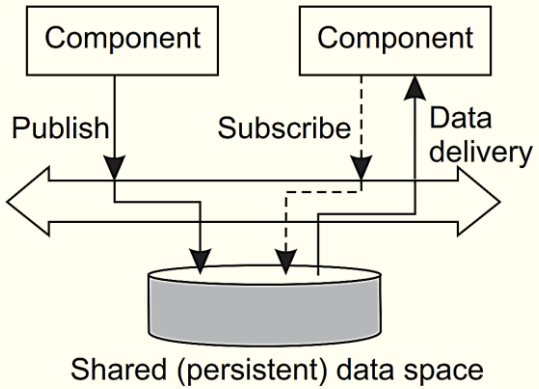
Publish-Subscribe Architectures:

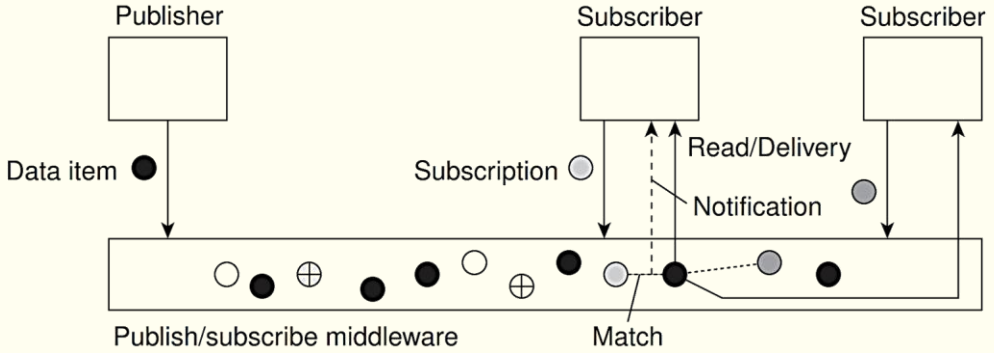
- Some large DS demand architecture with dependencies as loose as possible.
- So, a strong separation between processing and coordination is needed, this has led to:
  - Direct coordination, when processes are temporally and referentially coupled,
  - Event-based coordination, when processes are temporally or/and referentially decoupled.
  - Processes do not know each other explicitly! So, processes communicate via publish-notification fashion.
  - Processes may need to subscribe to receive notifications.

	Temporally coupled	**Temporally decoupled**
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

- Also, there is content-based publish-subscribe systems.

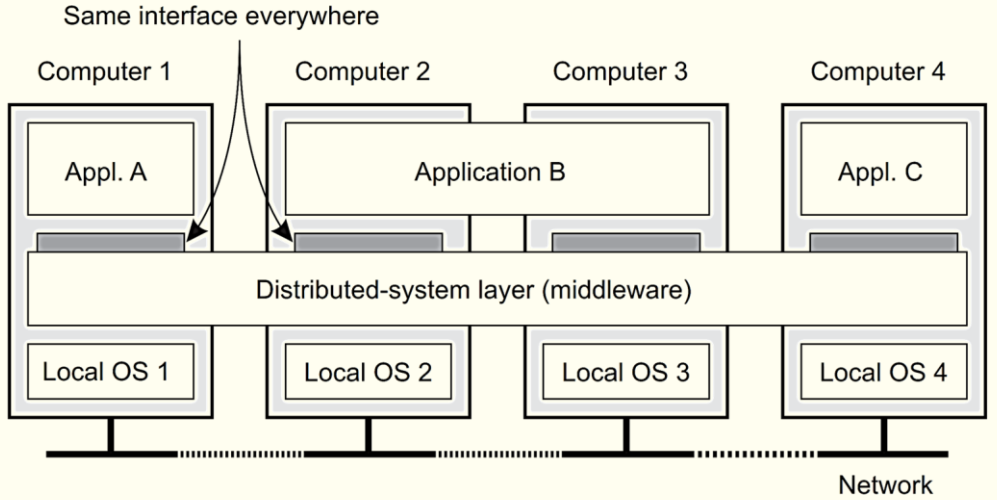




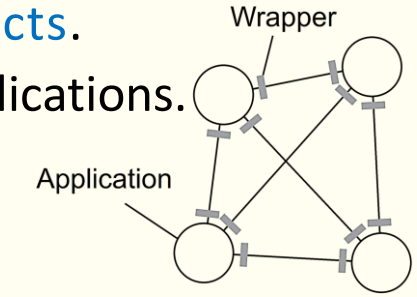


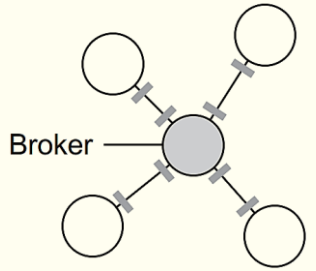
Middleware Architectures:

- In 1990s, to ease the development of DS, a separate layer of software is logically placed on top of the respective OS/platform known as Middleware.
- Middleware facilities inter-communication, security, accounting, masking OSs, platforms, failures, and recovery.
- Middleware must be portable, scalable, interoperable, transparent, and fault-tolerant.
  - Discussion: OSs are a sort of mindware, aren't they?



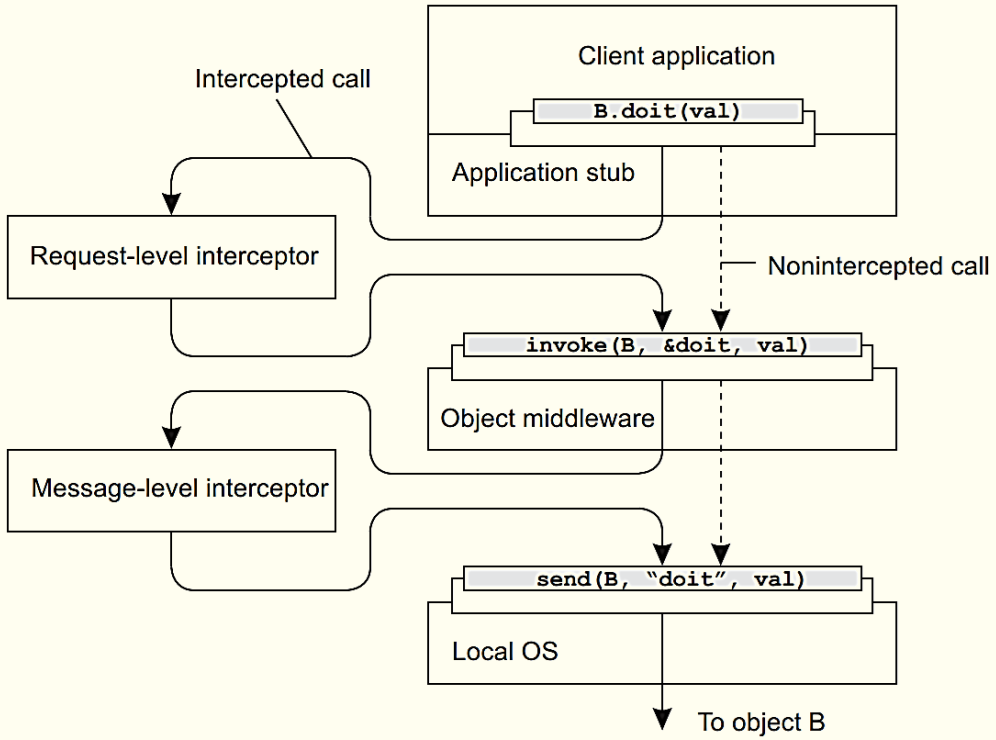
- Middleware construction involves one or more:
  - Wrapper is an interface component/object to a certain set of client applications.
  - Adapter is a component that allows applications to invoke remote objects.
  - Broker is a centralized component that handles all the accesses among applications.
  - Interceptor is a component that break the flow-of-control to adapt middleware to a need of an application.





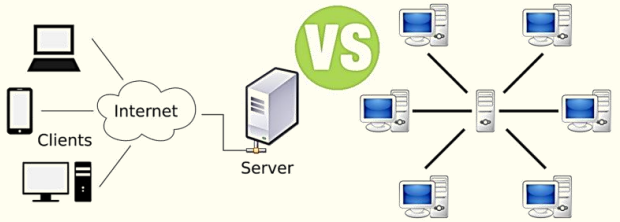
- Modifiable Middleware:
  - Computing environment evolves rabidly, e.g., Mobile and Pervasive systems.
  - Advanced middleware should handle and hide context changes and events on-the-fly.
  - Organizing middleware to be modifiable requires more research work.

- Component-based design focuses on supporting modifiability through composition. A system may either be configured statically at design time, or dynamically at runtime.

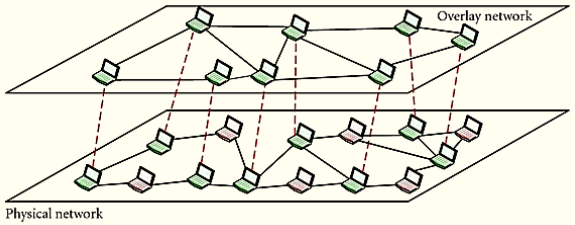


Symmetrically DSs Architectures (Decentralisation):

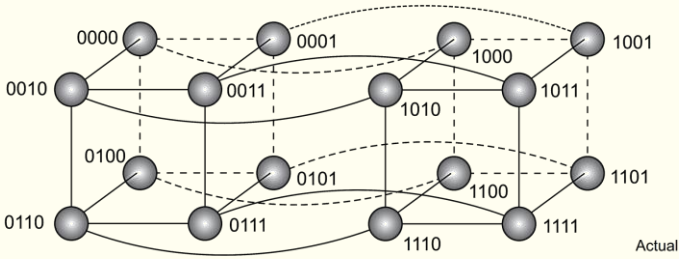
- The distribution of computation in DSs takes two forms:
  - Vertical distribution refers to splitting the computations into Master-Slave models corresponding to the application organisation i.e., client/server and multitiered architectures.



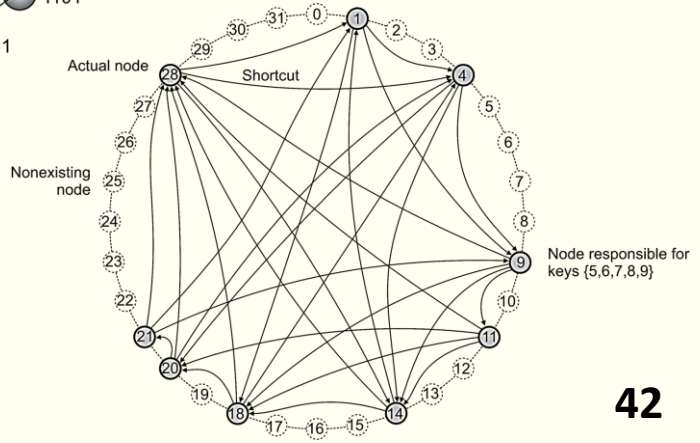
  - Horizontal distribution refers to equivalent computations models of client/server architecture corresponding with application organisation, e.g., peer-to-peer systems.
    - The interaction between processes/nodes is symmetric, "each process/node will act as a client and a server (a.k.a servant) at the same time.
    - Due to the symmetric style, peer-to-peer architectures revolve around how to organize the processes/nodes in an overlay network.



- Structured P2P systems:
  - Processes/nodes are organised in an overlay that adheres to a deterministic topology.
  - Make use of a semantic-free index to uniquely identifies nodes with a key and a global hash function:

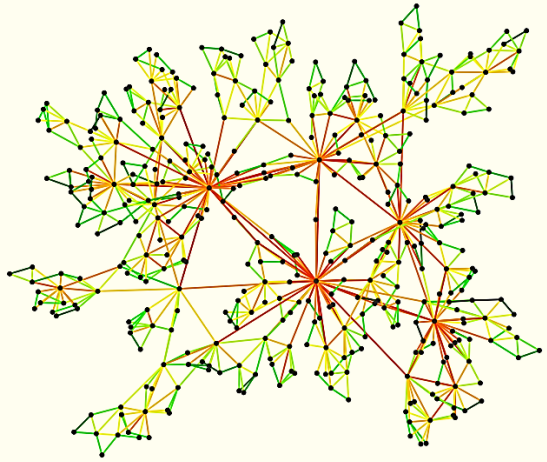


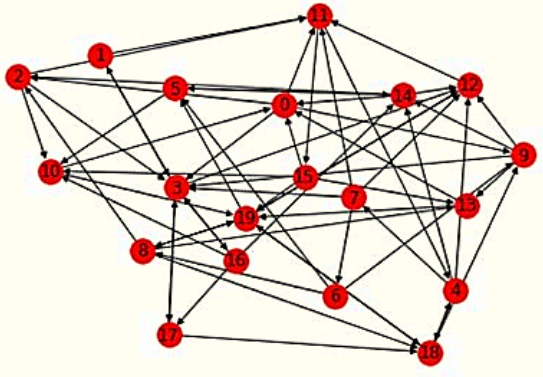
$$key(data\ item) = hash(data\ item's\ value)$$



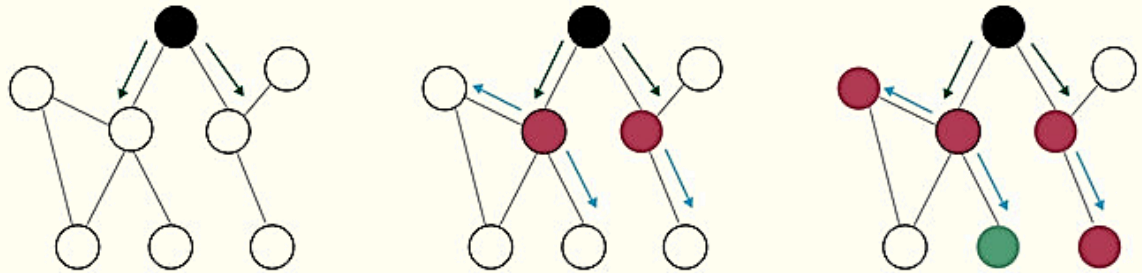
  - The Distributed Hash Table (DHT) is vital for distributed computation, data access and lookup in the overlay network (IP might be present in the underlying network!), e.g., Chord system.

- Unstructured P2P systems:
  - Each process/node maintains an ad-hoc/stochastic list of neighbours.
  - The overlay network forms a random graph with an edge (u, v) exists only with a certain probability $\mathbb{P}[\langle u, v \rangle]$.
  - P2P overlay can be static but commonly it forms dynamic topologies.
  - Exhibit several issues:
    node joining, neighbours membership, and search/lookup, routing, … etc.



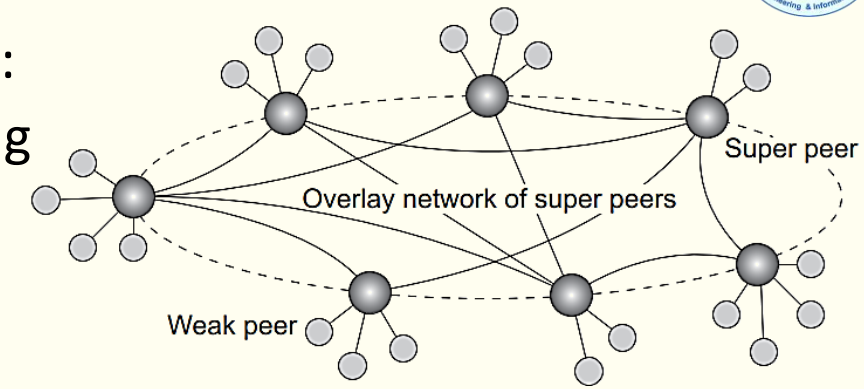


  - Diffusion/Dissemination techniques in P2P overlays:
    - Flooding: node u requests all neighbours, node v ignores if seen, forwards if not applicable, responds directly to u or to precedence node.
      To avoid expensive flooding, limit search by Time-To-Live (TTL) value.
      Random walk: node u requests a random neighbour node, forwards randomly if not applicable, or responds directly to u.
      To degrease lookup time, issuer node start n random walks simultaneously.
      To stop random walk,
      use TTL or policy-based methods.



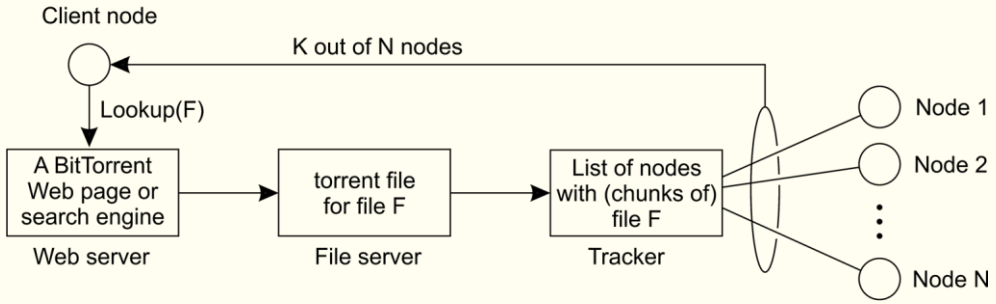
(1) (2) (3)

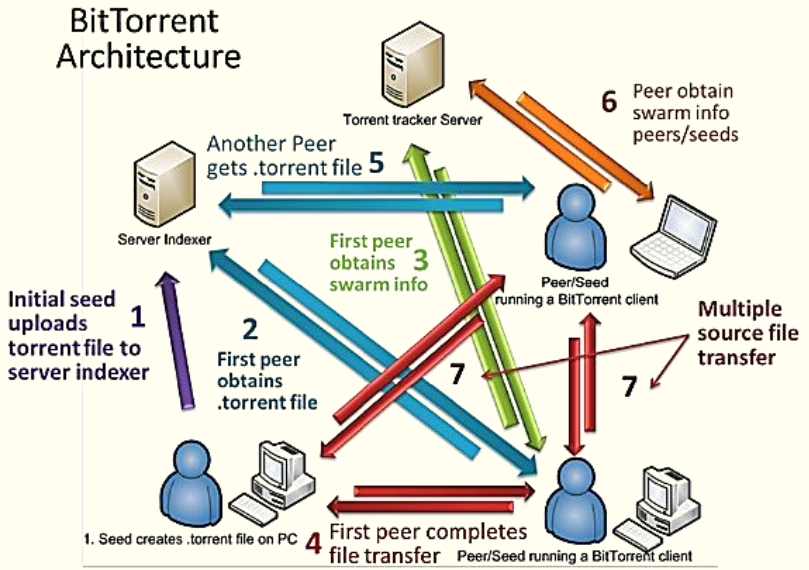
- Hierarchical P2P networks:
  - A.k.a Super-peer overlay topologies.
  - Sometimes, it is sensible to break the symmetry in P2P systems:
    - Having index servers improves the performance of searching in unstructured P2P systems, e.g., Content Delivery Network (CDN),
    - Brokers provide efficient data access.



  - Super-nodes are always fix or eligible to be chosen by some leader-election technique.
  - Weak peers can be associated with a fixed super-peer or deterministically or randomly.

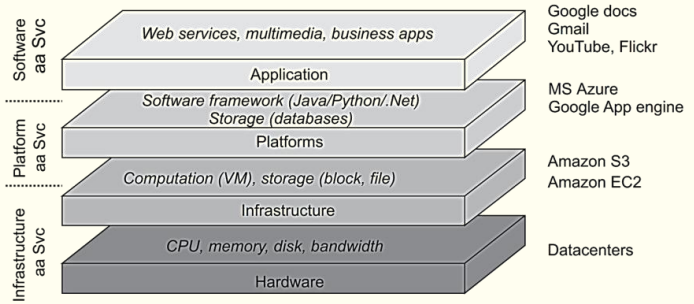
  - Examples:
    - BitTorrent.
    - Skype

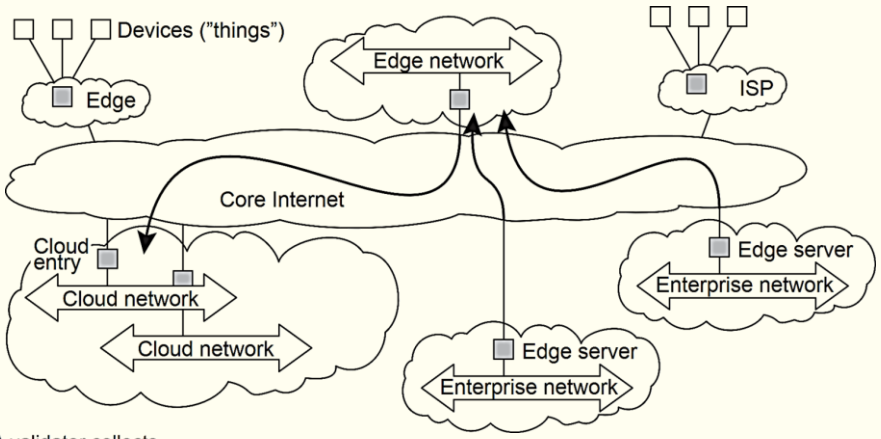


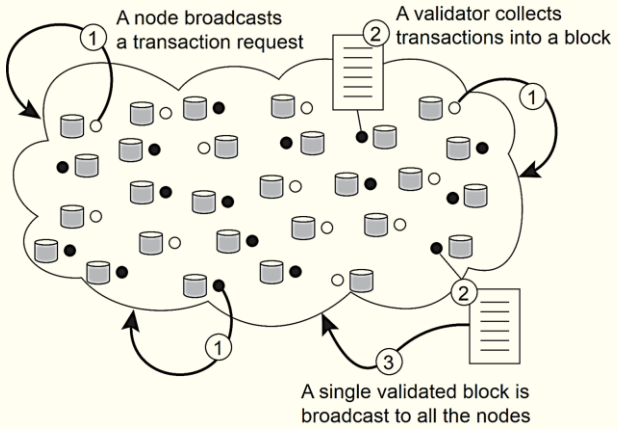


Hybrid System Architectures:

- Real-world DSs are complex, they combine hundreds of architectures to achieve the best features from all involved styles.
- Cloud Architecture:
  - Provides configurable microservices across dedicated infrastructure,
  - Highly advanced client/server architecture with completely hidden servers (Function-as-a-Service (FaaS)).
- Edge-cloud Architecture:
  - Servers are placed at the edge of the network (Hierarchical Style).
  - Better utilisation of latency & bandwidth.
  - Issues: reliability, security, privacy, resource allocation, service placement, edge selection.
- Blockchain Architecture:
  - Decentralised P2P distributed ledgers.
  - Validator requires distributed consensus:
    - Centralised!, distributed! (permissioned),
    - Decentralised! (permissionless).



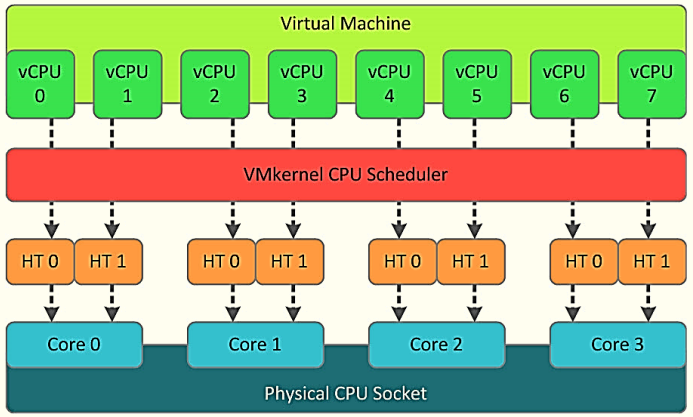




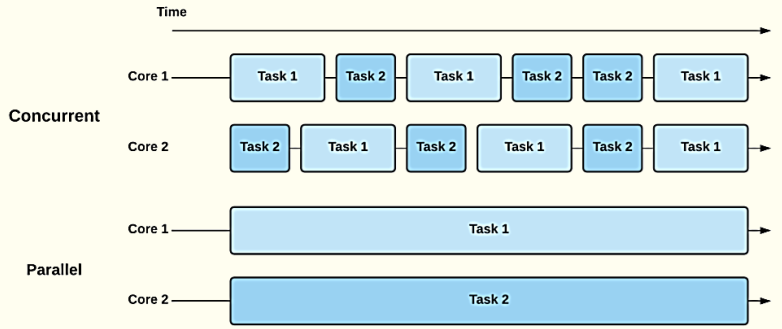
Processes

Basics:

- Software processes originated in the OSs, they play crucial roles in DSs.
- In recent OSs, there are virtual processors built on top of physical processors:
  - Multiprocessing: processes are executed simultaneously in a multi-processors system.
  - Multicore is a processor that has more than one independent processing units (core).
  - Multithreading: threads of a process are executed simultaneously.
  - Parallelism is the ability to execute independent processes in the same instant of time.
  - Concurrency is to execute multiple processes/threads in an overlapping time period, i.e., not running at the same time.

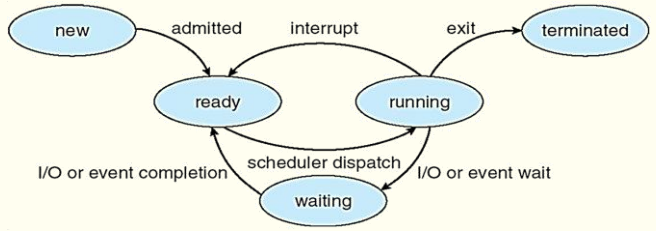


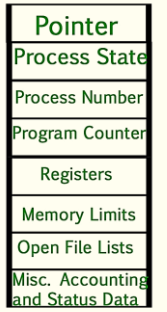
- This topic discusses many parallel and concurrent computing foundations:
  - Processes & Threads.
  - Zoom in Client/server Architecture.
  - Virtualisation.
  - Code Migration and Agents.



Processes & Threads:

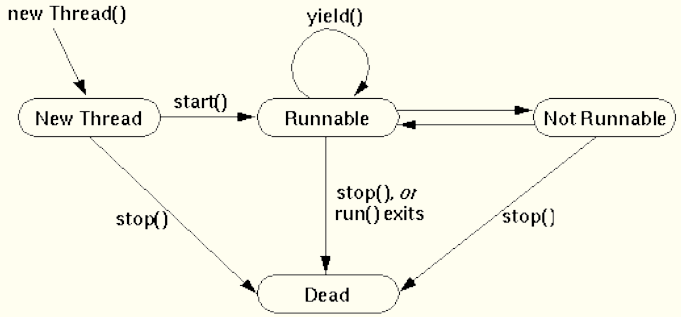
- An OS/DS process is a software program dispatched from the ready state and scheduled in the CPU for execution in an OS's virtual processor:
  - Each process has independent address space provides by the OS,
  - OSs identify and tracks processes via the Process Control Block (PCB),
  - It is heavyweight, takes time to create, terminate, and switch contexts!
  - It is isolated (HW protected) and preserves less efficient communication.



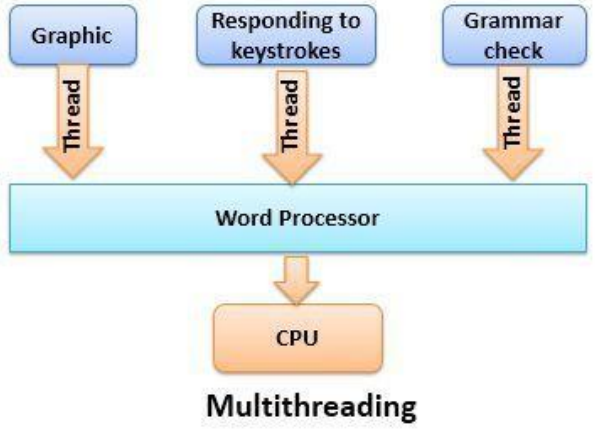


Process Control Block

- A thread is the segment (finer granularity) of a process, i.e., a process can contain multiple threads:
  - Threads are cheaper to create, terminate and switch!,
  - They share the same address space, not isolated (i.e., not protected against each other) and provide more efficient communication,
  - A multithreaded application performs highly better than its single-threaded counterpart.



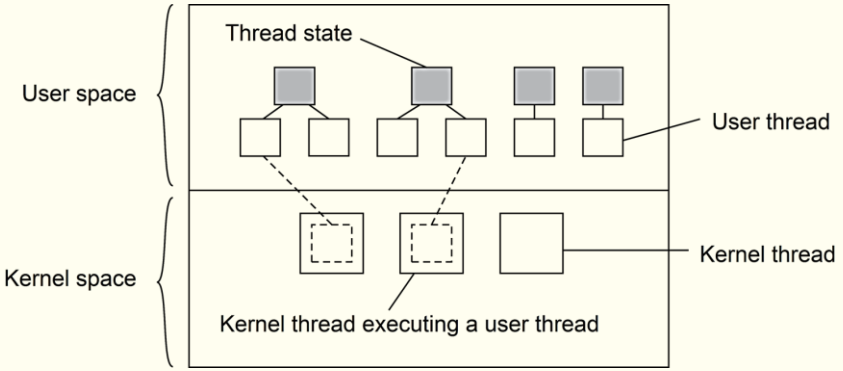
- Why use threads?
  - Avoid needless blocking in a multi-threaded process, e.g., when doing I/O.
  - Support parallelism as threads can run in parallel on a multi core/processor.
  - Avoid process switching as thread switching is faster, why?



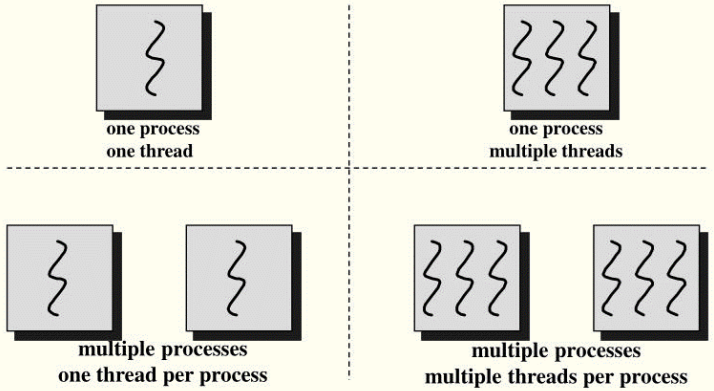
Multithreading

- Discussion: is an android/java thread similar in structure to a windows thread?

- Threads Implementation:
  - Typically, threads are provided in programming package/library, which contains operations to create, destroy and synchronise threads (e.g., mutexes and condition variables).
    - User-level threads (many-to-one-model): implemented by the user, cheap switching but single schedulable entity.
      - A blocking system call blocks the entire process!
    - Kernel-level threads (one-to-one-model): implemented by OSs, transparent to the application.

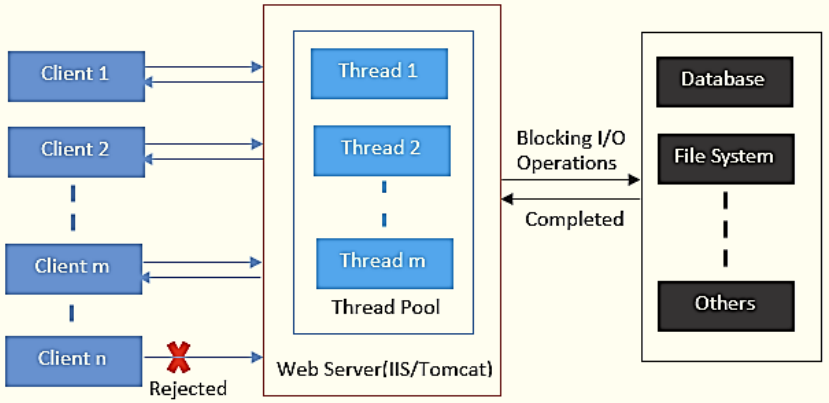


  - User-level and Kernel-level threads are combined into a single concept (Hybrid Threads), leading to performance gain but with increased complexity (Lightweight Processes).
  - Kernel-level threads are runnable threads that executes User-level threads.
  - In practice, applications are a collection of concurrent processes, and make use of the inter-process facilities in the OS, e.g., Web-Servers.
  - Processes have isolated memory space and HW protected; but heavyweight.

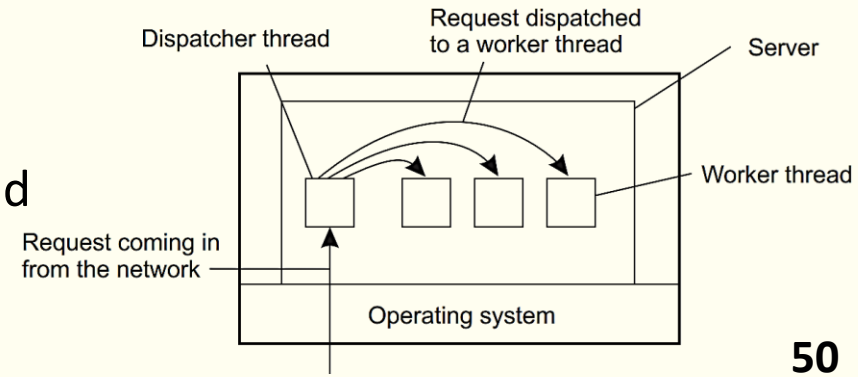


  - Threads are faster and lighter but difficult to implement as developers are responsible for managing concurrent access to data.

- Threads in DSs:
  - Client-side uses multi-threaded computing with high degree of transparency, e.g., to:
    - Hide network latency in Web-browser (multiple downloads).
    - Make simultaneous calls each by a different thread, then each thread waits for the response.
  - Server-side use multi-threaded computing to achieve high performance, e.g.,
    - By using cheap threads rather than processes.
    - By hiding network latency with multithreaded connections.
    - By using optimised I/O blocking calls.
    - By adopting Dispatcher/Worker model
    - Sever construction options:

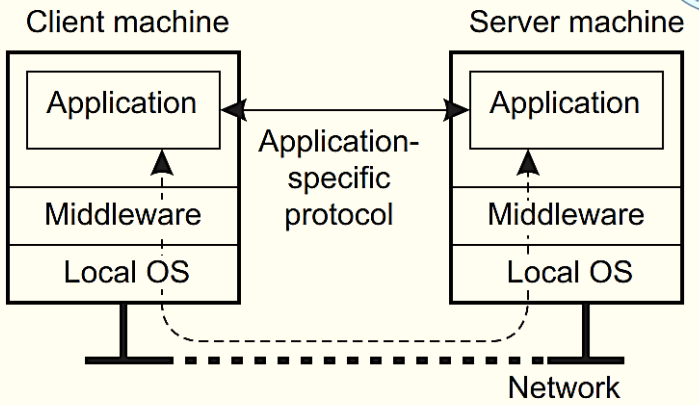


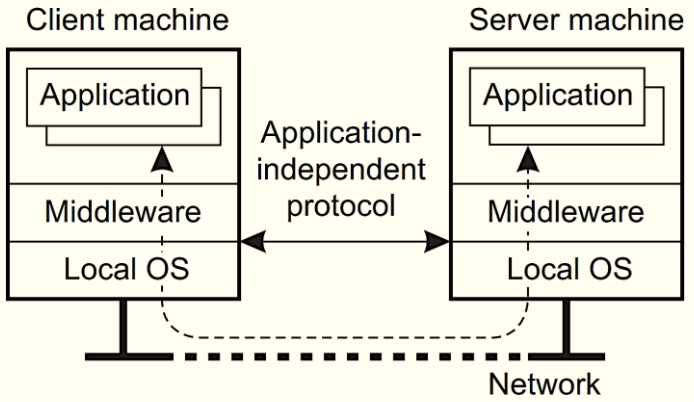
1. Single-threaded process: provide no parallelism and preserve blocking system calls,
2. Threads (no supervisor): support parallelism with blocking system calls, i.e., in a single-processor machine, a thread may cause I/O or HW blocking.
3. Finite-state (with supervisor) machine: on an event/request, a dispatcher thread (supervisor) activates a thread to react; the dispatcher saves threads' state on the context-switching and switches context for block-causing threads; provides no parallelism and nonblocking system calls.

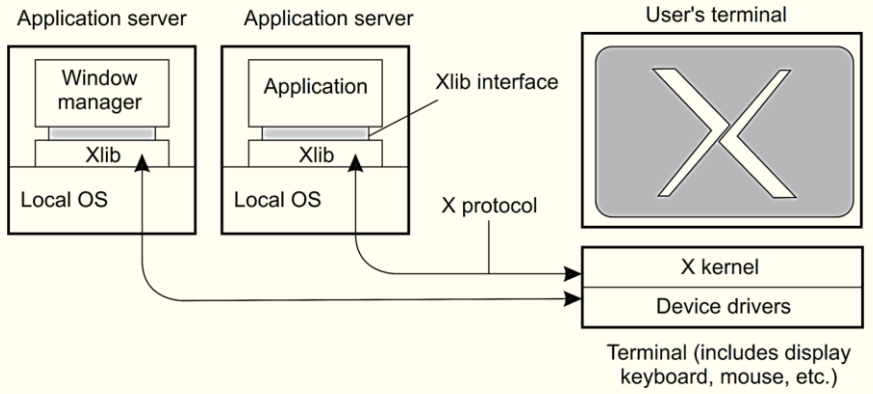


Anatomy of Clients:

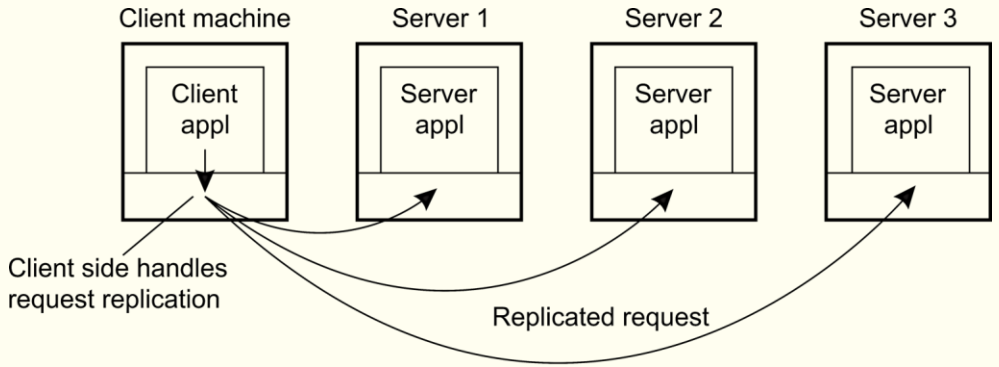
- Clients typically are means for users to interact with the DS.
- There are two interaction ways:
  1. A client machine has a separate counterpart/component that communicates with a remote service over a network, e.g., Calendar Apps.
     - Requires an application-level protocol to handle the communications/synchronisation,
  2. A client machine provides direct access to remote services by offering only a convenient user interface, e.g., command line terminals.
     - There is no local data storage or processing.
- Examples:
  - X-window system acts as a client to the X-kernel running as a server.
  - Virtual Network Computing (VNC).
  - Victual Desktop Environment (e.g., RDP, AnyDesk, TeamViewer).
  - Web Browsers and Document Object Model (DOM).
  - Progressive Web Apps (PWA).







- Client-side software:
  - Client software comprises more than user interfaces; some computing in a client-server application takes place on the client side.
  - Generally tailored for distribution transparency:
    - Access, hiding data/resources representation and access; e.g., clients RPC stubs handle remote objects access,
    - Location/migration /relocation, hiding server/resources location; clients can track actual/new server/resources location via naming services and brokers,
    - Replication, hiding server/resources replicas; the client stub may issue multiple invocations to replicas,

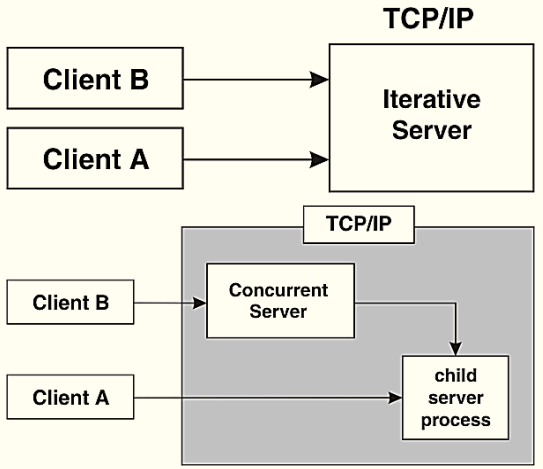


    - Failure, through masking server/resources and communication failures; recovery via retransmission and multiple connection techniques.
    - Concurrency, typically, transaction monitors distribute loads for load-balancing; enabling server/resource sharing among clients; require less support from client software.

- Discussion: what roles do multithreading & virtualisation play in client distribution transparency?

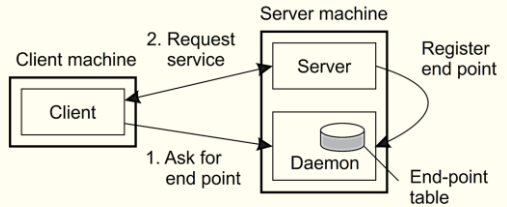
Servers Organisation:

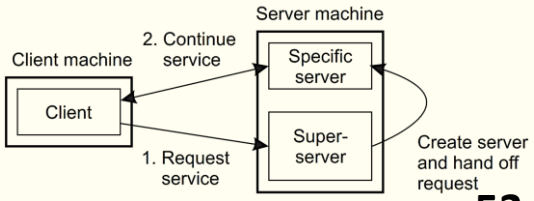
- Generally, servers wait (Passive Process) for an incoming request from a client, process the request, wait for the next incoming request.
- Basic server types:
  - Iterative: a server handles the request before attending a next request.
  - Concurrent: via a dispatcher that picks up an incoming request that is then passed on to a separate thread/process.
    The norm, notably in the presence of blocking.



- Issues in contacting a server:
  1. Clients send requests to an end-point (port) where a service is listening.
     - How do clients know the end-point of a service?
     - ➢ One approach is to globally assign end-points for well-known services, typically, assigned by the Internet Assigned Numbers Authority (IANA).
     - ➢ Another approach is dynamically assigned end-points via a hidden service (daemon) that tracks services end-points. However, this approach is resource-draining.
     - ➢ More efficient approach is to implement a single super-server listening to all end-points associated with alive services

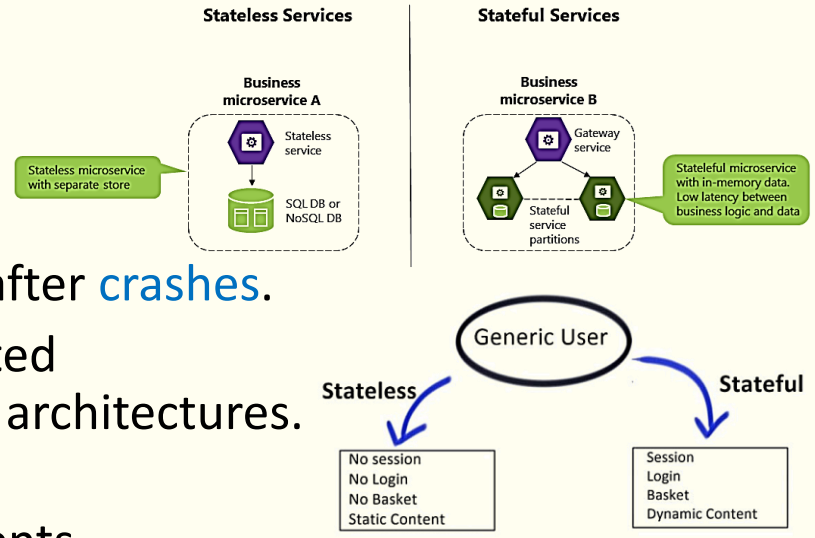
ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
smtp	25	Simple Mail Transfer
www	80	Web (HTTP)





2. Interrupting a server:
   - Is it possible to interrupt a server once it has accepted a service request?
   - Examples: cancelling a file upload, or closing a playing stream.
   - The server will eventually tear down the old connection, thinking the client has crashed.
   - A better approach, use a separate thread/process (port) for urgent requests (out-of-band data),
   - Another approach, use facilities of the transport layer, e.g., TCP and OS signalling techniques.

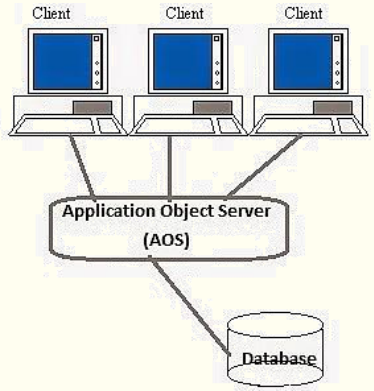
3. Stateless and stateful servers:
   - A stateless server does not keep information about its clients, and can change its own state without informing clients, e.g., HTTP requests to Web servers. So: clients and servers are completely independent and have minimum state inconsistencies after crashes.

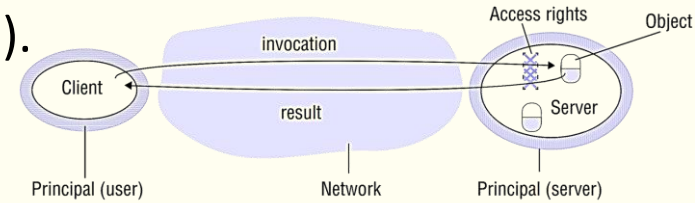


   - A soft-state server maintains state on behalf of the client for a limited time (Session State)., e.g., maintained in three-tiered client-server architectures.
   - A stateful server generally maintains persistent information (Operations, Crashes) on its clients. It improves performance as clients keep local copies, but preserve complex recovery on failures due to recovering all client's tracks.
   - States in a stateful server can be Sessional or Permanent: vital information are stored in DB.
   - The choice for a stateless or stateful design should not affect the services provided by the server, e.g., cookies and Web servers.

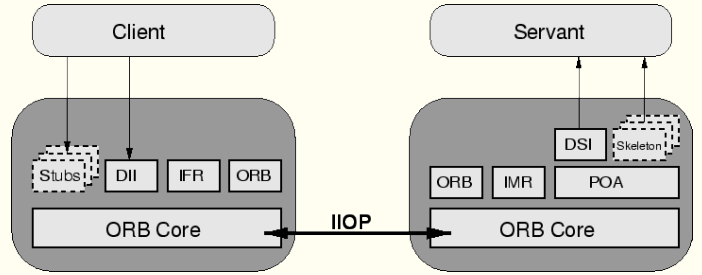
- Discussion: does connection-oriented communication fit into a stateless design?

- Object Servers:
  - Common servers are object servers.
  - Object server by itself does not provide a specific service. Specific services are implemented by the objects that reside in the server.
  - Alive remote object reside in a server consist of two parts: data representing its state and the code for executing its methods.
  - Object server can either:
    - Keep callable objects alive all the time (resource-draining), or
    - Activate callable objects as long as they really needed (Performance Issue!).
    - Assign a thread for each callable object, or
    - Assign a separate thread for each invocation request.



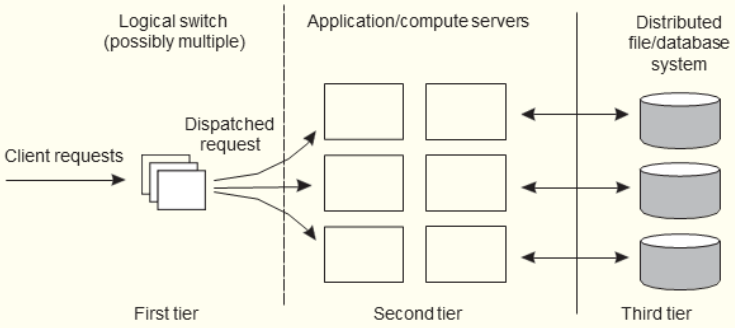


  - Object server use generic object adapter/wrapper to attach objects to a server application following a specific activation policy.
  - An object adapter supports different policies via runtime configuration and can operate in single-threaded or multithreaded mode.
  - A servant is the general term for a piece of code that forms the implementation of an object.

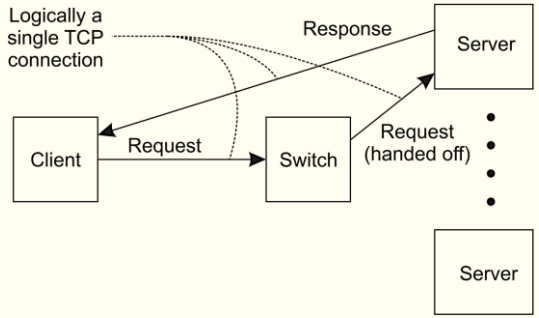


  - For examples of servers supporting activation policies, see CORBA-compliant systems.
  - For examples of servers that balance the separation between policies and mechanisms, study the Apache Web server.

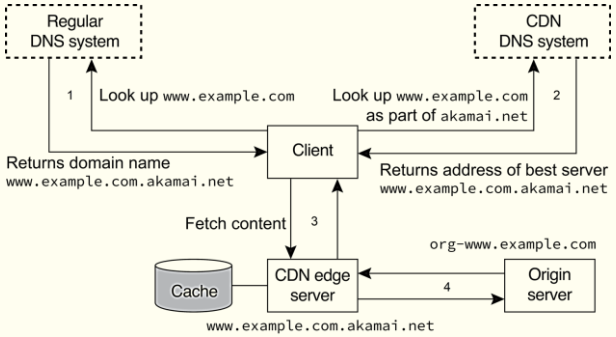
- Server Clusters:
  - A collection of machines connected through a network, where each machine runs one or more servers.
  - Often, a server cluster is logically organized into three-tiers, not all clusters are 3-tiers, e.g., media streaming are 2-tiers.
  - The first tier is responsible for passing requests to an appropriate server: request dispatching (Access Transparency).



  - Access transparency is maintained by address/content request switching:
    - Transport-layer switches (TCP), e.g., NAT, IPv6, or
    - Application-layer switches, e.g., URL.

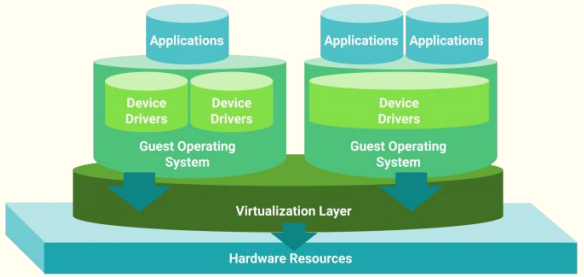


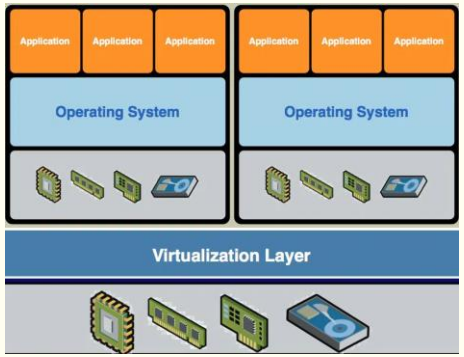
  - Wide-area server clusters (Grid Servers) aim to offer data and services that are close to clients, e.g., Amazon, Google, PalnetLab, and Content Delivery Networks (CDN).
  - For scalability and availability, a cluster may have multiple access points,
  - Origin server maintains a home address, but hands off connections to address of collaborating peer, origin server and peer appear as one server.



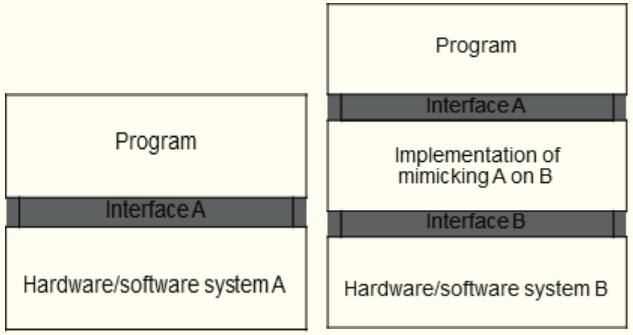
Virtualization:

- Virtualization is applied since 1990s, but has received attention due to the recent DSs.
  - To handle hardware changes that is faster than software,
  - To ease of portability and code migration,
  - To help in isolation of failing or attacked components.
  - It is vital concept for DSs, it helps achieve portability and handle failures

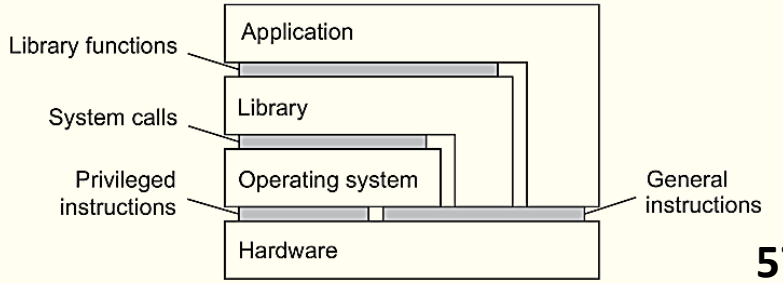




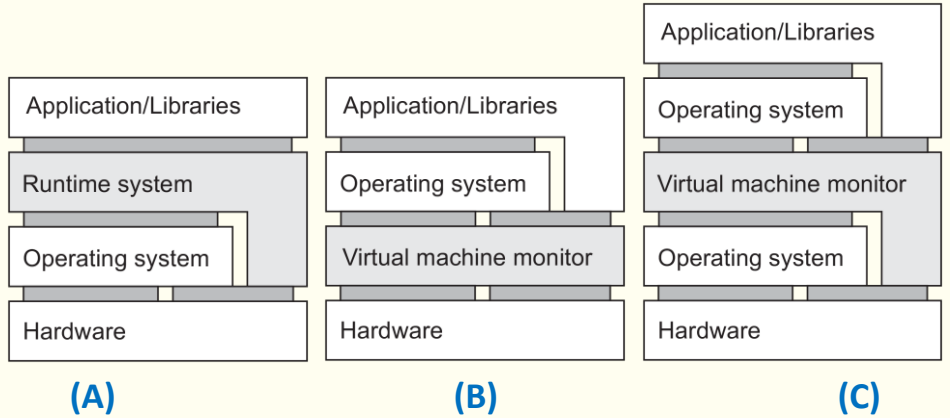
- In recent OSs, there are virtual processors built on top of physical processors:
  - A native service of an OS that allows an application/program to run concurrently with other applications, but highly independent of the underlying hardware and platforms.



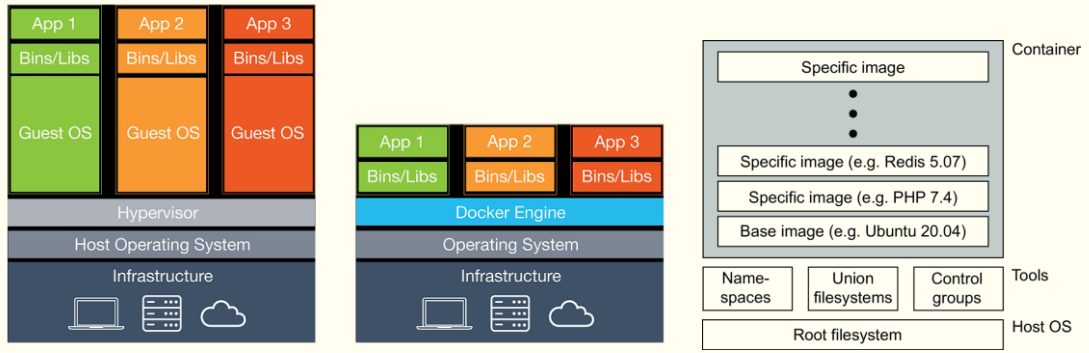
- In practice, every DS offers a programming interface to higher-levels:
  - Instruction Set Architecture (ISA): between hardware and software, with two subsets:
    - Privileged instructions: allowed to be executed only by the OS.
    - General instructions: can be executed by any program.
  - System calls as offered by an operating system.
  - Library calls, a.k.a Application Programming Interface (API).

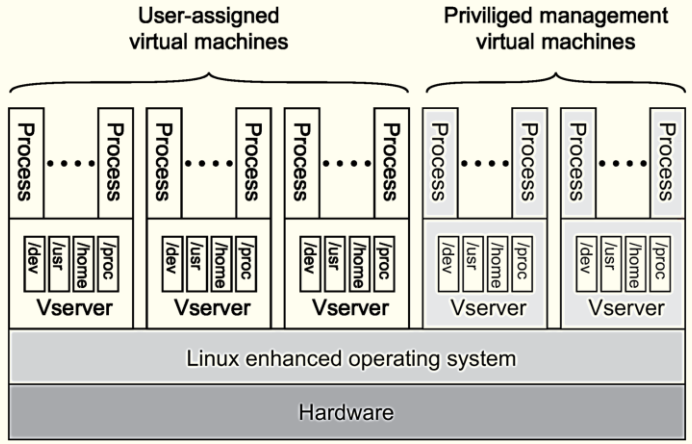


- Virtualization can take place in different ways:
  - What examples for each scenario (a),(b), and (c)?
  - Common issues: portability, transparency, performance, fault-tolerance, and security,
  - Mian issue is the implementation of control-sensitive and behaviour–sensitive instructions,
    Solution is modify guest OS (Paravirtualization).



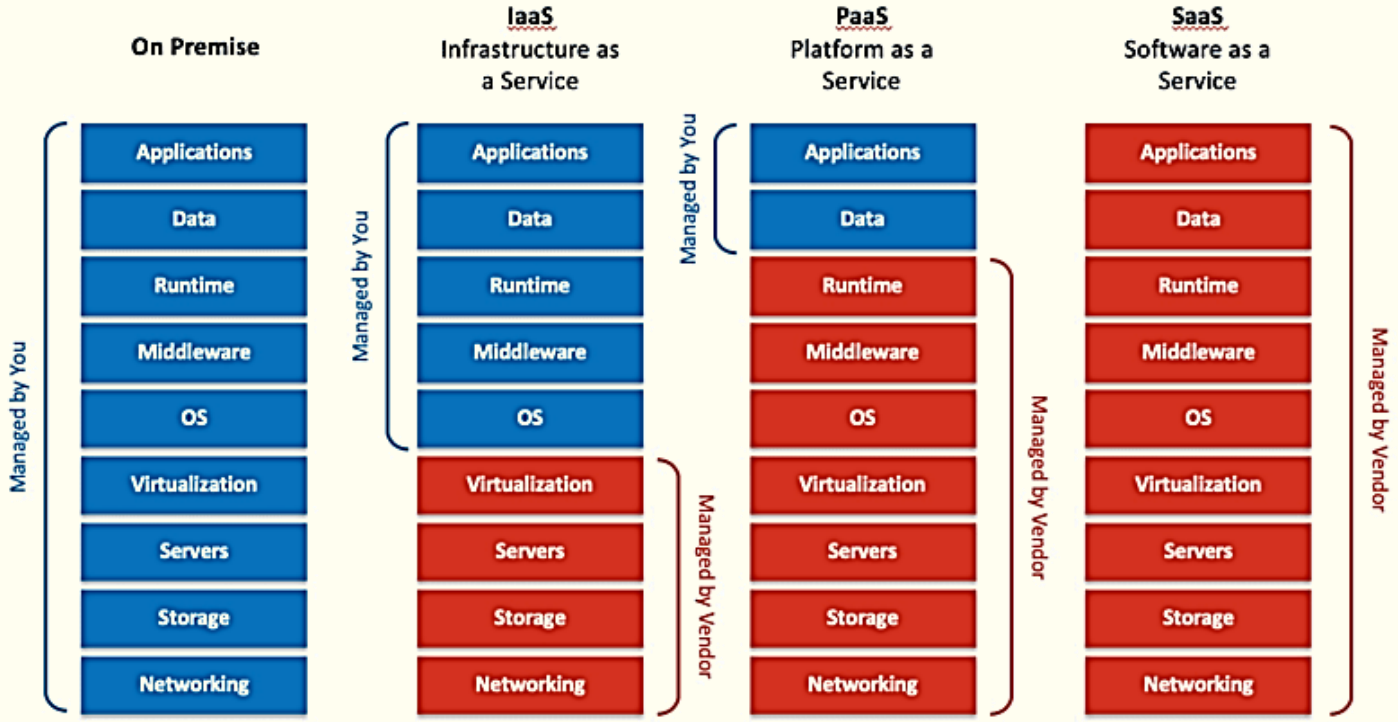
- Containers:
  - They are virtual machines run applications side-by-side but independently across different platforms.
  - They require significant efforts to ensure portability and performance.
  - A container is a collection of binaries/platform images that constitute the environment for running applications.
  - Applications operating in different containers need to be isolated.
  - Copying an entire platform is not efficient.
  - Hosting platform need some control over containers.
  - Very common in Linux platforms.
  - Example: PlanetLab is a DS system for collaborative research.





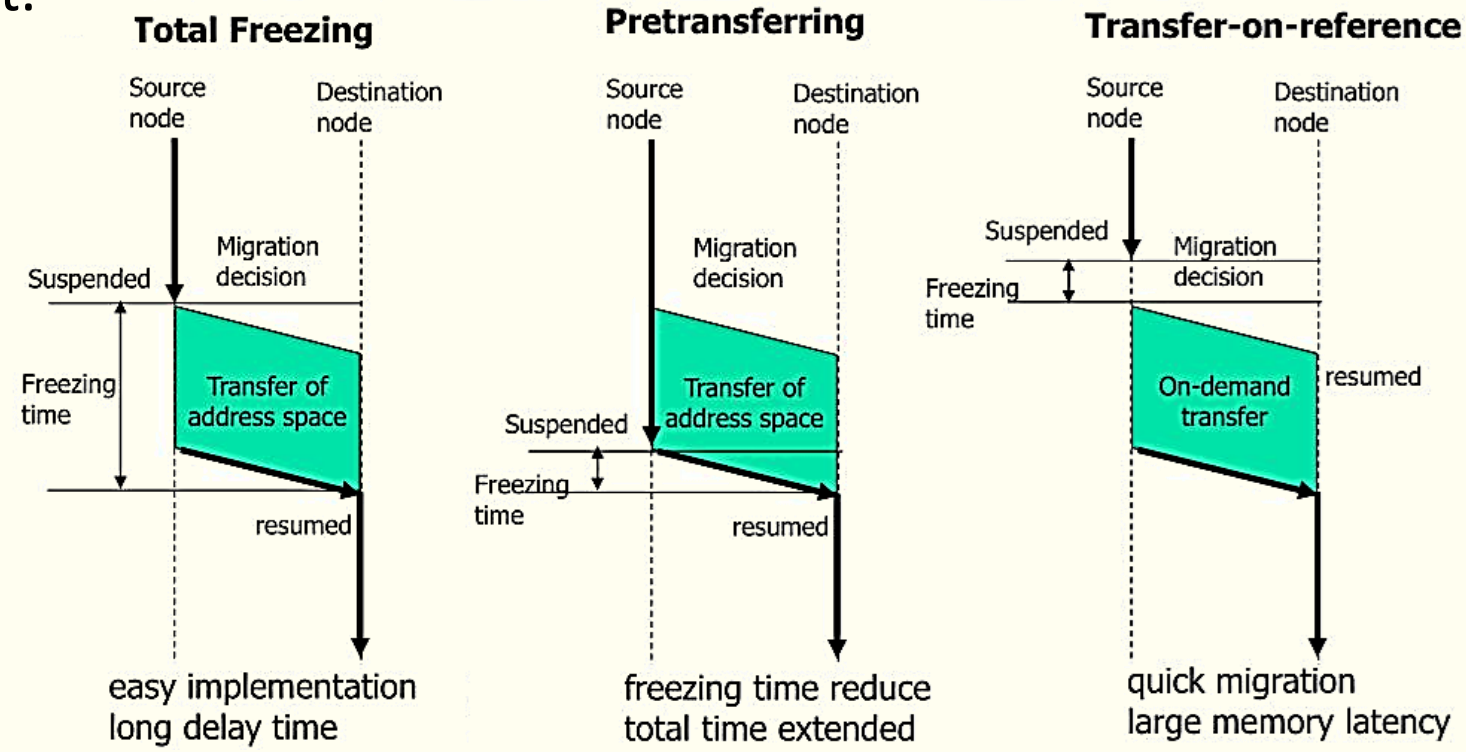
- Discussion: what is better: virtual machines or containers?

- Application of virtual machines to DS (Cloud Computing):
  - A cloud provider rent-out a virtual machine and shares a physical machine among customers.
  - A cloud provider ensures virtual machines' isolation and performance.
  - Clouds provide three types of services:
    - Infrastructure-as-a-Service (IaaS) covering the basic infrastructure,
    - Platform-as-a-Service (PaaS) covering system-level services,
    - Software-as-a-Service (SaaS) containing actual applications.

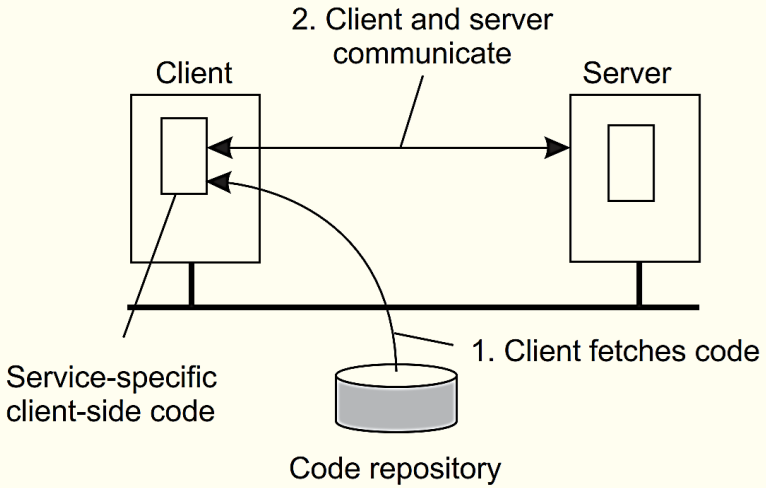


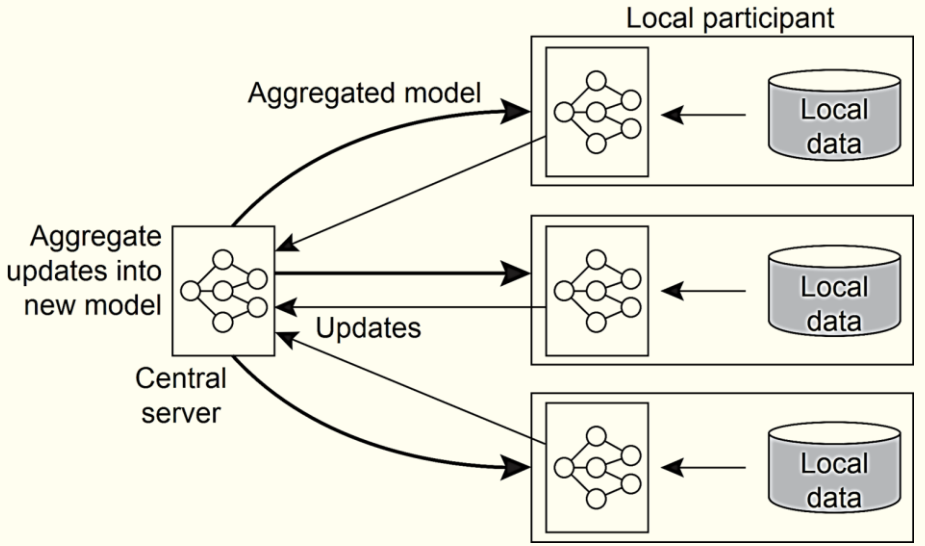
Code Migration:

- So far, distributed objects/components/services communicate to pass data.
- There are situations in which passing programs/codes simplifies the DSs.
- Historically, code migration meant process migration, i.e., An entire process was moved from a node to another.
- In DSs, optimisation of computing power is aless pressing issue than minimising communication cost.

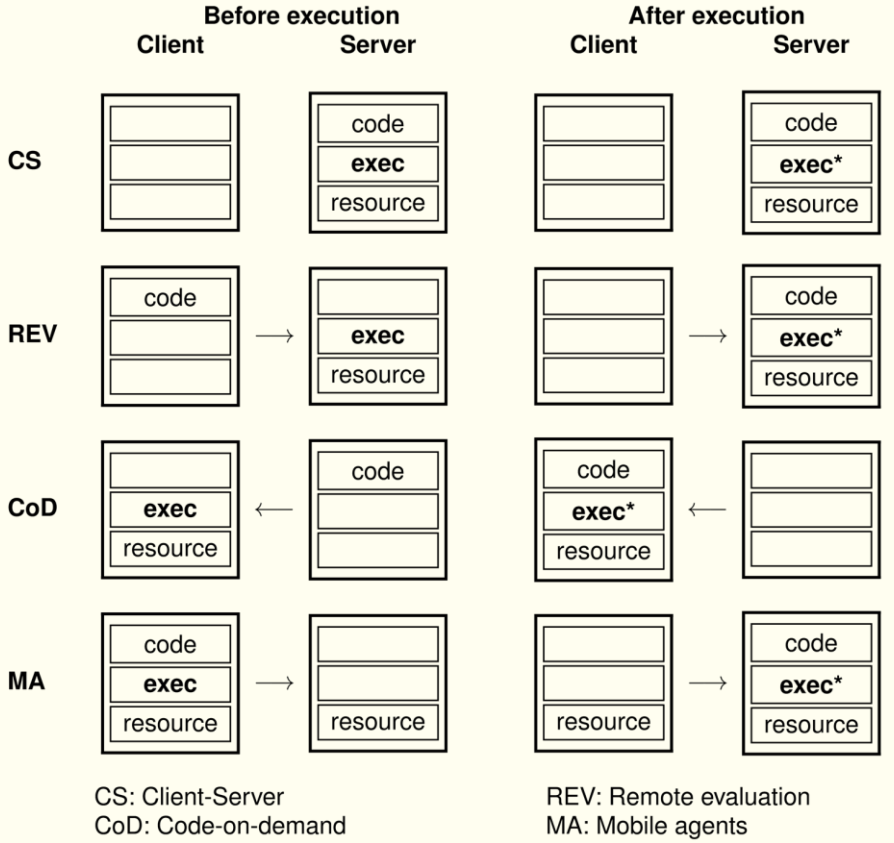


- Why migrate code?
  - Often migration decision is based on qualitative reasoning instead of mathematical models, e.g. migrate DB queries to the server rather than sending them over the network.
  - Performance: move processes to lightly loaded! nodes (i.e., offloading), Also, via parallelism and agents, e.g., search mobile agents.
  - Load distribution:
    - To ensure that servers in a data-centre are sufficiently loaded, e.g., to save energy,
    - To minimize communication by that computations are close to where the data is, e.g., data training & mining (federate learning).
  - Privacy and security: move process where data is for security.
    Flexibility: migrate code to a client when needed, e.g., moving an interface to access server's file system.





- Models for code migration:
  - To understand the model for code migration, refer to a framework proposed by Fuggetta et al.
  - Code segment contains the set of instructions of the program being executed.
  - Resource segment contains references to external resources needed by the process, e.g., files, printers.
  - Execution segment is used to store the current execution state of a process.
  - Sender-initiated migration.
  - Receiver-initiated migration.
- The four different paradigms for code mobility,
  - Client-server (CS): segments are in and run at the server.
  - Remote Evaluation (REV): code migrates for execution at the sever.
  - Code-on-Demand (CoD): code migrates for execution at the client.
  - Mobile Agents (MA): segments migrate to server for execution.
    - Weak mobility: migrate code, data, and reboot execution.
    - Strong mobility: migrate code, data, and continue execution.
      - Either by migrating the whole object or by cloning it.



- For code migration, the problems coming from heterogeneity!
  They are in many respects the same as those of portability.

Review Questions:

1. Resources sharing is an objective of DSs, give examples of hardware, data, and software resources that might be shared in a DS. Explain the most needed requirements for efficient resource sharing.
2. Discuss two of the following DSs challenges: heterogeneity, openness, fault handling, concurrency, quality of service, scalability, and transparency.
3. Clock synchronisation is problematic in DSs, explain the factors that limit the accuracy of the clocks. How could organise a global clock in a large system? Discuss the accuracy of global clock synchronisation mechanisms .
4. What is novel about cloud computing as a concept in comparison to traditional client-server computing?
5. A server program written in C++ provides objects that are intended to be accessed by clients. Clients are written in Java. Both server and client objects have access to the internet. Describe the problems due to heterogeneity that need to be solved to make it possible for a client object to invoke a method on a server object.
6. Discuss the sharing, scalability, and anonymity in different peer-to-peer architectures.
7. Discuss the query and data dissemination in the peer-to-peer architectures. Consider centralised, structured, unstructured, and hierarchal topologies.
8. Computing components may fail when a client process invokes a method in a server object, discuss with examples the expected failures. Suggest how components can be made to tolerate such failures.
9. Resources in the WWW and other services are named by URLs. Discuss? Give examples of web resources that can be named by URLs.
10. What problems are foreseen in the direct coupling between communicating entities that is implicit in remote invocation approaches? Consequently, what advantages can be anticipated from a level of decoupling as offered by referential and time uncoupling?
11. List one typical criterion for selecting a node to be promoted to a super node. Justify why such a choice would improve network stability.
12. Choose a major Internet application of your favourite, and discuss with illustrations the client-server architecture of that application..
13. A search engine is a web server that responds to client requests to search in its stored indexes and concurrently runs several web crawler tasks to build and update the indexes. discuss and explain the requirements for synchronisation between these concurrent activities.
14. Most peer-to-peer systems are often simply personal computers in users' offices or homes. What are the implications of this for the availability and security of any shared data objects that they hold and to what extent can any weaknesses be overcome through the use of replication? .
15. Assume a transaction processing system that maintains ACID. Briefly describe one serious shortcoming of each of the following implementations: (a) The database is updated on disk with each transaction. (b) The database is kept in memory and on disk, with the copy on disk updated every 50 transactions. (c) The database is kept in memory. A log file is maintained on disk recording every transaction.
16. Discuss the Middleware architecture styles.
17. Consider any business company. Sketch out a three-tier solution to the distributed service of the company. Use this to illustrate the benefits and drawbacks of a three-tier solution considering issues such as performance, scalability, dealing with failure and also maintaining the software over time.
18. For each of the factors that contribute to the time taken to transmit a message between two processes over a communication channel, state what measures would be needed to set a bound on its contribution to the total time. Why are these measures not provided in current general-purpose distributed systems?
19. What is a case where it is preferable to use multiple threads instead of multiple processes? What is a case where the reverse is true?
20. Suppose you have a stateful service. What techniques could you use to scale it vertically or horizontally?

Communications