Distributed System

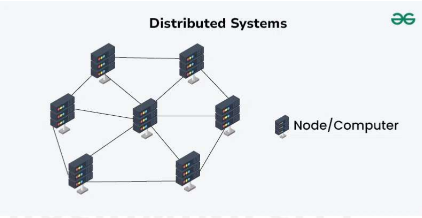
MSc. Huynh Nam

Introduction

- What is a distributed system
- Design goals
- Type of distributed systems

What is a distributed system

- A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.
  - 1st: A collection of computing elements each being able to behave independently of each other
    - **Node**: hardware device or a software process
  - 2nd: That users (people or applications) believe they are dealing with a single system
    - That one way or another the autonomous nodes need to collaborate
- Three characteristics:
  - Collection of autonomous computing elements
  - Single coherent system
  - Middleware and distributed systems



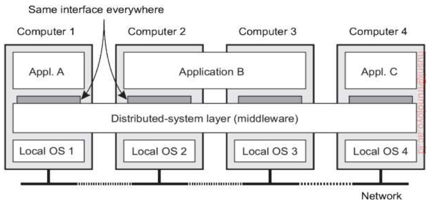
Collection of autonomous computing elements

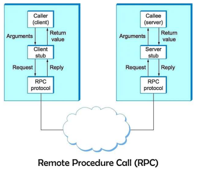
- Nodes can act independently from each other, ignore each other.
- Node reacts to incoming messages, which are then processed and, in turn leading to further communication through **message passing**.
- Common reference of time (**global clock**) leads to fundamental question regarding the *synchronization* and *coordination* with a distributed system.
- Managing group membership (admission control) is difficult: **open** and **close group** (all member is trusted)**->** need mechanism
  - 1st : authenticate a node -> issue: a scalability **bottleneck**.
  - 2nd : check if it is indeed communicating with another group member or not.
  - 3rd: facing trust issues in communicating with nonmember.
- Organized as an **overlay network**
  - Structured overlay: each node has a well-define set of neighbors with whom it can communicate.
  - Unstructured overlay: each node has a number of references to randomly selected other nodes.

Single coherent system (single-system view, single-system image)

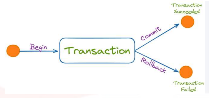
- In distributed system, end user would not be able to tell exactly on which computer a process is currently executing, where data us stored, replicating data …. So called **distribution transparency**.
- An important **trade-off**: That unexpected behavior in which, for example, some applications may continue to execute successfully which others come to a grinding halt -> difficult to hide.

Middleware and distributed systems





Remote Procedure Call (RPC)



Database
Application Services
Web Services
Networking
Security

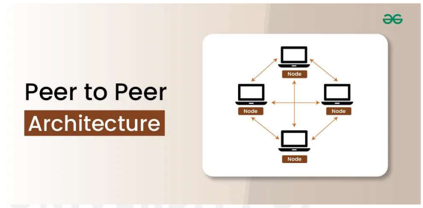
- A separated layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system: **resource management**
  - Facilities for inter-application communication
  - Security services
  - Accounting services
  - Masking of and recovery from failures
- Middleware services are offered in a networked environment (main difference with OS). Some typical middleware services
  - Communication: Remote Procedure Call (PRC)
  - Transaction
  - Service composition
  - Reliability

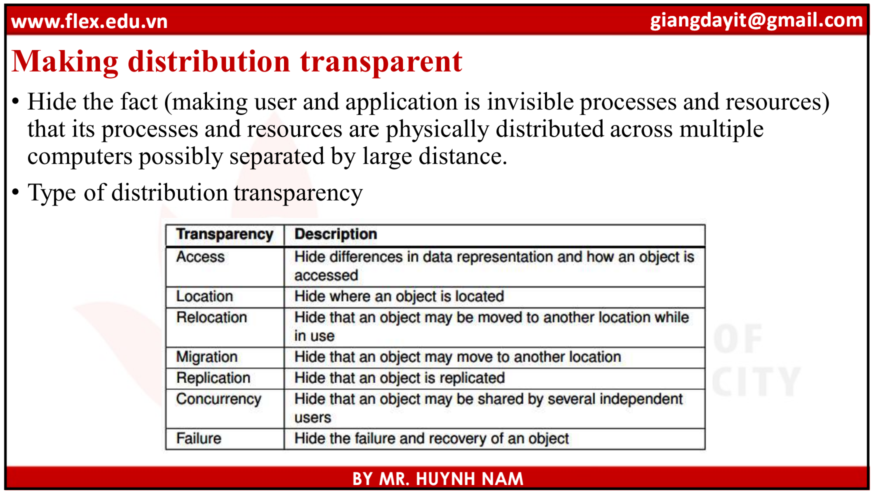
Design goals

- There are four important goals to make distributed system worth the effort
  - Supporting resource sharing
  - Making distribution transparent
  - Being open
  - Being scalable
    - Scaling techniques
- Pitfalls

Supporting resource sharing

- Making group of people (**groupware**) easy to access and share remote resources.
- Resources: virtually anything (peripherals, storage, data file, network …)
  - Peer-to-peer network: distribution of media files such as video, audio. The technology is used for distributing large amounts of data, as in case software updates, backup services, and data synchronization across multiple servers.

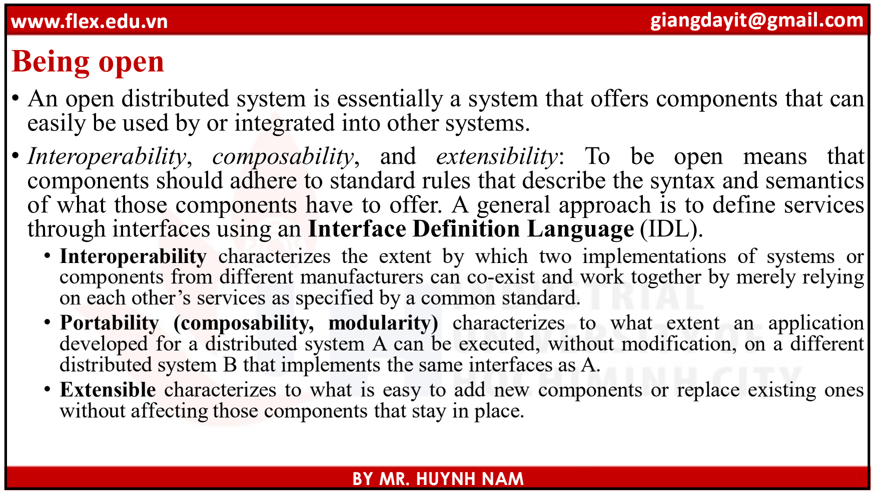




Making distribution transparent

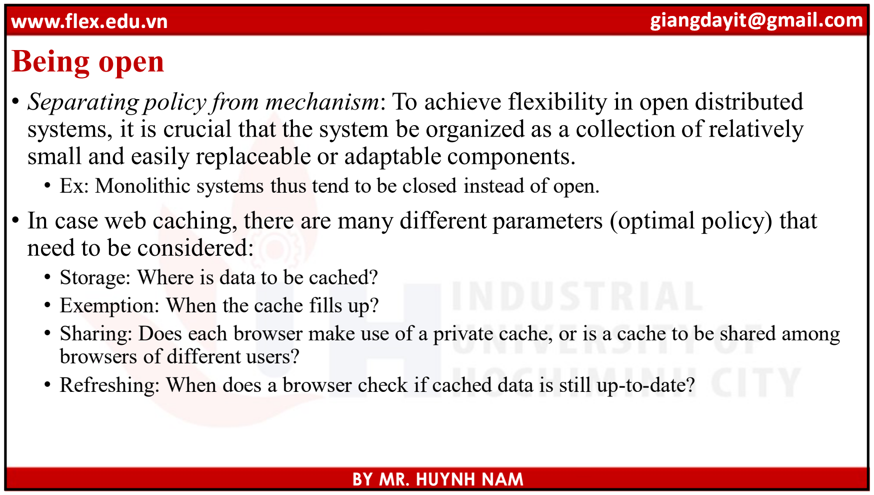
- Hide the fact (making user and application is invisible processes and resources) that its processes and resources are physically distributed across multiple computers possibly separated by large distance.
- Type of distribution transparency

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object



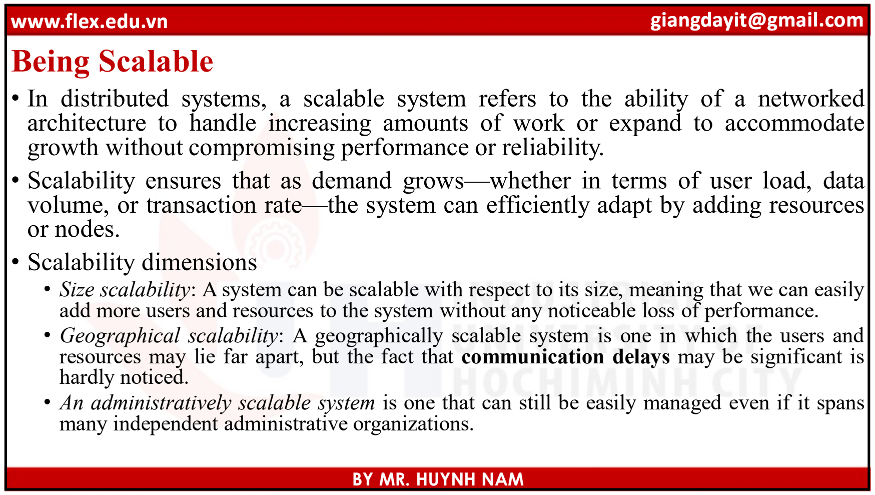
Being open

- An open distributed system is essentially a system that offers components that can easily be used by or integrated into other systems.
- *Interoperability*, *composability*, and *extensibility*: To be open means that components should adhere to standard rules that describe the syntax and semantics of what those components have to offer. A general approach is to define services through interfaces using an **Interface Definition Language** (IDL).
  - **Interoperability** characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard.
  - **Portability (composability, modularity)** characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A.
  - **Extensible** characterizes to what is easy to add new components or replace existing ones without affecting those components that stay in place.



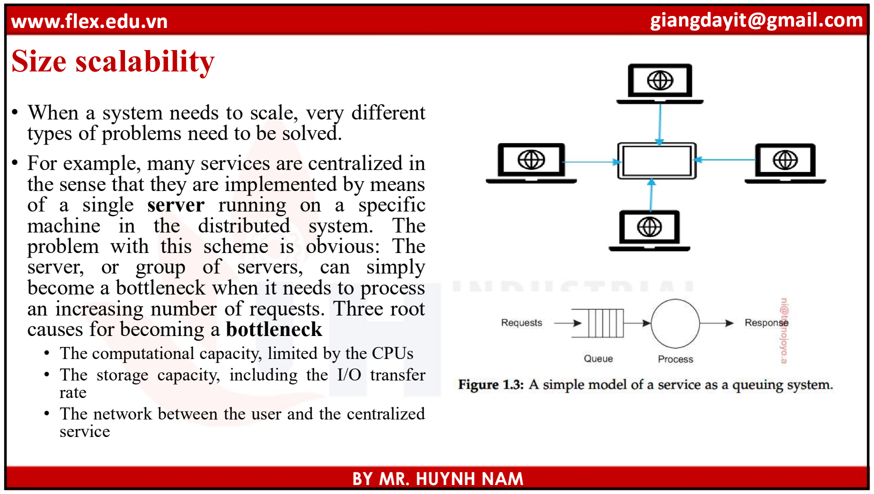
Being open

- *Separating policy from mechanism*: To achieve flexibility in open distributed systems, it is crucial that the system be organized as a collection of relatively small and easily replaceable or adaptable components.
  - Ex: Monolithic systems thus tend to be closed instead of open.
- In case web caching, there are many different parameters (optimal policy) that need to be considered:
  - Storage: Where is data to be cached?
  - Exemption: When the cache fills up?
  - Sharing: Does each browser make use of a private cache, or is a cache to be shared among browsers of different users?
  - Refreshing: When does a browser check if cached data is still up-to-date?



Being Scalable

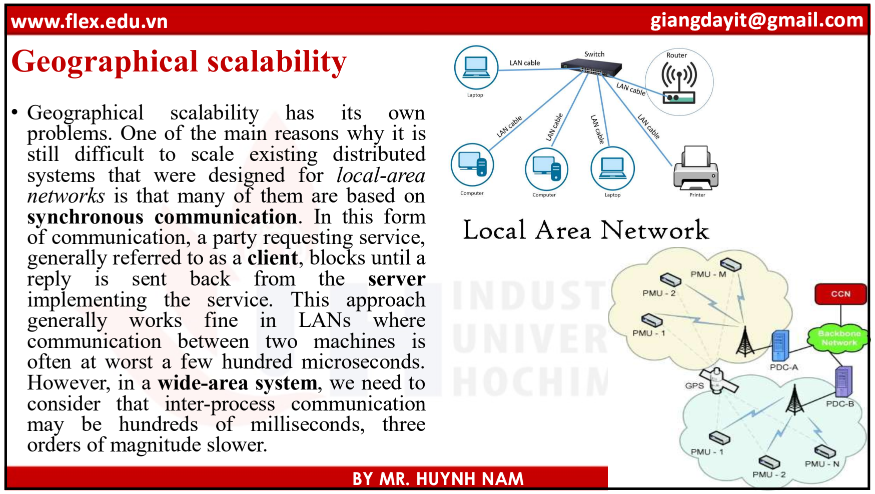
- In distributed systems, a scalable system refers to the ability of a networked architecture to handle increasing amounts of work or expand to accommodate growth without compromising performance or reliability.
- Scalability ensures that as demand grows—whether in terms of user load, data volume, or transaction rate—the system can efficiently adapt by adding resources or nodes.
- Scalability dimensions
  - *Size scalability*: A system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system without any noticeable loss of performance.
  - *Geographical scalability*: A geographically scalable system is one in which the users and resources may lie far apart, but the fact that **communication delays** may be significant is hardly noticed.
  - *An administratively scalable system* is one that can still be easily managed even if it spans many independent administrative organizations.



Size scalability

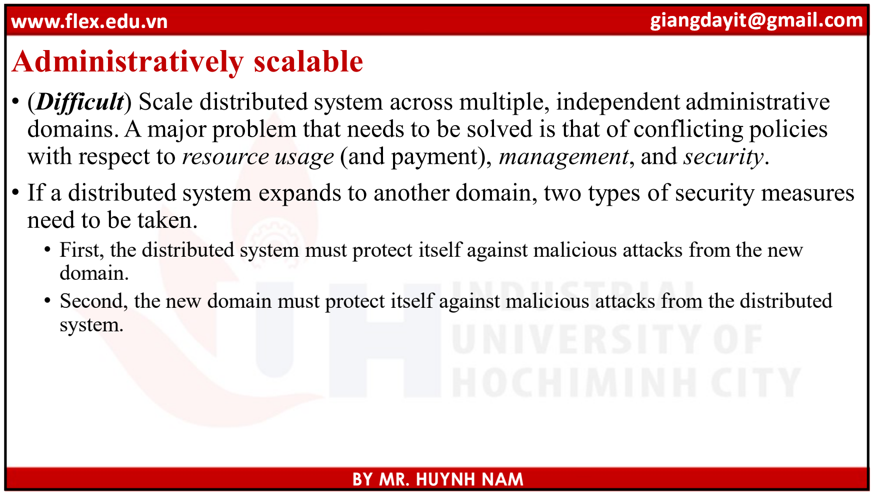
- When a system needs to scale, very different types of problems need to be solved.
- For example, many services are centralized in the sense that they are implemented by means of a single **server** running on a specific machine in the distributed system. The problem with this scheme is obvious: The server, or group of servers, can simply become a bottleneck when it needs to process an increasing number of requests. Three root causes for becoming a **bottleneck**
  - The computational capacity, limited by the CPUs
  - The storage capacity, including the I/O transfer rate
  - The network between the user and the centralized service

Figure 1.3: A simple model of a service as a queuing system.



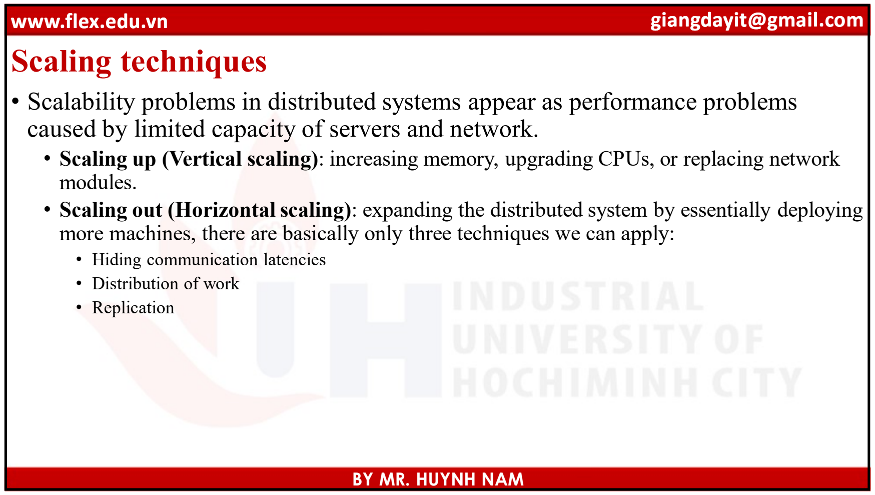
Geographical scalability

- Geographical scalability has its own problems. One of the main reasons why it is still difficult to scale existing distributed systems that were designed for *local-area networks* is that many of them are based on **synchronous communication**. In this form of communication, a party requesting service, generally referred to as a **client**, blocks until a reply is sent back from the **server** implementing the service. This approach generally works fine in LANs where communication between two machines is often at worst a few hundred microseconds. However, in a **wide-area system**, we need to consider that inter-process communication may be hundreds of milliseconds, three orders of magnitude slower.



Administratively scalable

- (***Difficult***) Scale distributed system across multiple, independent administrative domains. A major problem that needs to be solved is that of conflicting policies with respect to *resource usage* (and payment), *management*, and *security*.
- If a distributed system expands to another domain, two types of security measures need to be taken.
  - First, the distributed system must protect itself against malicious attacks from the new domain.
  - Second, the new domain must protect itself against malicious attacks from the distributed system.



Scaling techniques

- Scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network.
  - **Scaling up (Vertical scaling)**: increasing memory, upgrading CPUs, or replacing network modules.
  - **Scaling out (Horizontal scaling)**: expanding the distributed system by essentially deploying more machines, there are basically only three techniques we can apply:
    - Hiding communication latencies
    - Distribution of work
    - Replication

Hiding communication latencies

- Hiding communication latencies is applicable in the case of geographical scalability. The basic idea is simple: *try to avoid waiting for responses to remote-service requests as much as possible.*
- Essentially, this means constructing the requesting application in such a way that it uses only **asynchronous communication**. When a reply comes in, the application is interrupted, and a special handler is called to complete the previously issued request.
- Asynchronous communication can often be used in *batch-processing systems* and *parallel applications* in which independent tasks can be scheduled for execution while another task is waiting for communication to complete.
- Alternatively, a new *thread* of control can be started to perform the request. Although it blocks waiting for the reply, other threads in the process can continue.

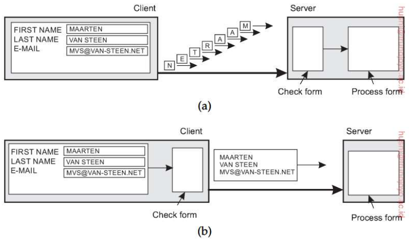


Figure 1.4: The difference between letting (a) a server or (b) a client check forms as they are being filled.

Partitioning and distribution of work

- Which involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system.

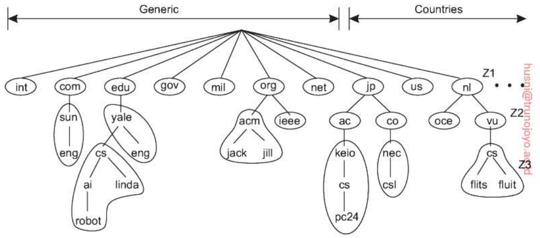
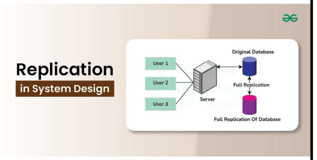
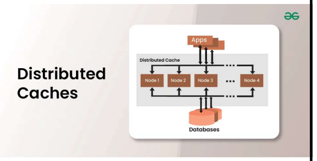


Figure 1.5: An example of dividing the (original) DNS name space into zones.

Replication

- Considering that scalability problems often appear in the form of performance degradation, it is generally a good idea to actually replicate components across a distributed system. Replication not only increases availability, but also helps to balance the load (**load balancer**) between components leading to better performance. Also, in geographically widely dispersed systems, having a copy nearby can hide much of the communication latency problems mentioned before.
- **Caching** is a special form of replication, although the distinction between the two is often hard to make or even artificial. As in the case of replication, caching results in making a copy of a resource, generally in the proximity of the client accessing that resource. However, in contrast to replication, caching is a decision made by the client of a resource and not by the owner of a resource.
- There is one serious drawback to caching and replication that may adversely affect scalability. Because we now have multiple copies of a resource, modifying one copy makes that copy different from the others. Consequently, caching and replication leads to **consistency** problems.
- Replication therefore often requires some **global synchronization mechanism**. Unfortunately, such mechanisms are extremely hard or even impossible to implement in a scalable way, if alone because network latencies have a natural lower bound.





Pitfalls

- Developing a distributed system is a formidable task.
- Distributed systems differ from traditional software because components are dispersed across a network.
- False assumptions when developing a distributed application
  - The network is reliable
  - The network is secure
  - The network is homogeneous
  - The topology does not change
  - Latency is zero
  - Bandwidth is infinite
  - Transport cost is zero
  - There is one administrator
- Issues when developing a distributed application: **reliability**, **security**, **heterogeneity**, and **topology** of the network; **latency** and **bandwidth**; transport **costs**; and finally **administrative** domains.

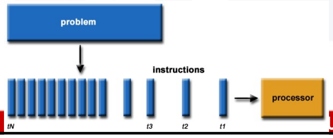
Type of distributed systems

- High performance distributed computing (HPC)
  - Parallel processing
- Distributed information systems
- Pervasive systems

HPC - Parallel Computing

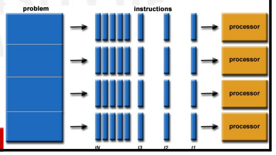
Serial Computing

- Traditionally, software has been written for serial computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time



Parallel Computing

- The simultaneous use of multiple compute resources to solve a computational problem:
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different processors
  - An overall control/coordination mechanism is employed

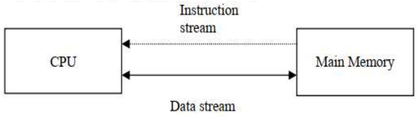


Characterize problem in parallel processing

- The **computational problem** should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The **compute resources** are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network

Parallel computer definition

- The term ‘stream’ refers to a sequence or flow of either instructions or data operated on by the computer.
- In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called instruction stream.
- Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called data stream.
- Instruction and data stream: it can be said that the sequence of instructions executed by CPU forms the Instruction streams and sequence of data (operands) required for execution of instructions form the Data streams.



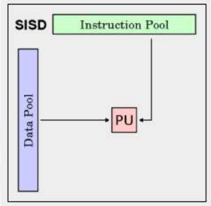
Flynn’s Classification

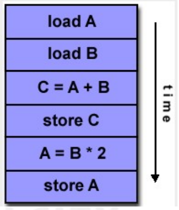
- There are different ways to classify parallel computers.
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn’s classification is based on multiplicity of instruction streams and data streams observed by the CPU during program execution. Each of these dimensions can have only one of two possible states: Single or Multiple.

SISD Single Instruction stream Single Data stream	**SIMD** Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	**MIMD** Multiple Instruction stream Multiple Data stream

Single Instruction, Single Data (SISD)

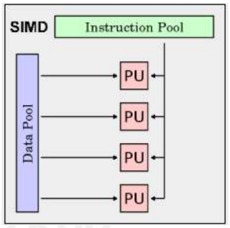
- Single Instruction: Only one instruction stream is being acted on by the CPU during any one clock cycle
- Single Data: Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest type of computer
- Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.

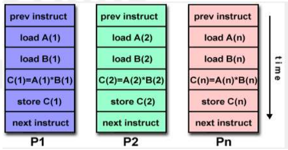



load A
load B
C = A + B
store C
A = B * 2
store A

Single Instruction, Multiple Data (SIMD)

- Single Instruction: All processing units execute the same instruction at any given clock cycle
- Multiple Data: Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines

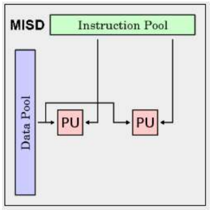


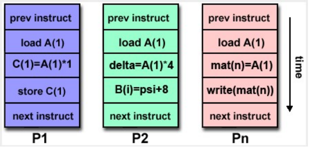


P1	P2	Pn
prev instruct	prev instruct	prev instruct
load A(1)	load A(2)	load A(n)
load B(1)	load B(2)	load B(n)
C(1)=A(1)*B(1)	C(2)=A(2)*B(2)	C(n)=A(n)*B(n)
store C(1)	store C(2)	store C(n)
next instruct	next instruct	next instruct

Multiple Instruction, Single Data (MISD)

- Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.
- Single Data: A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.
- Some conceivable uses might be:
  - Multiple frequency filters operating on a single signal stream
  - Multiple cryptography algorithms attempting to crack a single coded message.

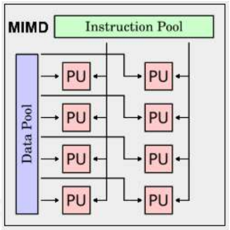


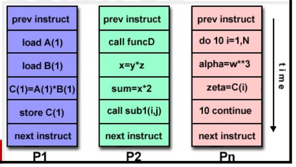


P1	P2	Pn
prev instruct	prev instruct	prev instruct
load A(1)	load A(1)	load A(1)
C(1)=A(1)*1	delta=A(1)*4	mat(n)=A(1)
store C(1)	B(i)=psi+8	write(mat(n))
next instruct	next instruct	next instruct

Multiple Instruction, Multiple Data (MIMD)

- Multiple Instruction: Every processor may be executing a different instruction stream
- Multiple Data: Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components



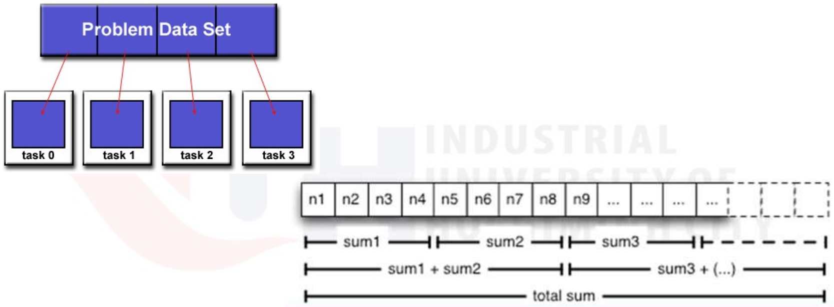


Parallelizing problem

- Key idea is Partitioning
- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.
- There are two basic ways to partition computational work among parallel tasks: domain decomposition and functional decomposition.
- Typically, combining these two types of problem decomposition is common and natural.

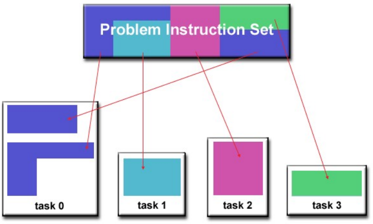
Domain Decomposition

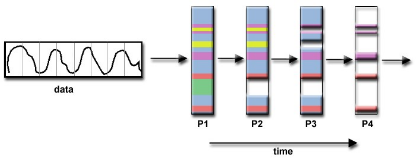
- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.



Functional Decomposition

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.





Limitation of computational parallel

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized.
- Where N = number of processors and S = the fraction of code is serial.

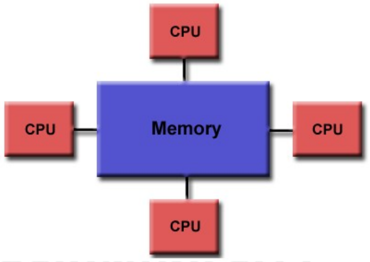
```
                         1
    speedup   =    ------------
                    P   +   S
                   ---
                    N
```

Parallel Computer Memory Architecture

- Shared Memory: UMA, NUMA
- Distributed Memory
- Hybrid Distributed-Shared Memory

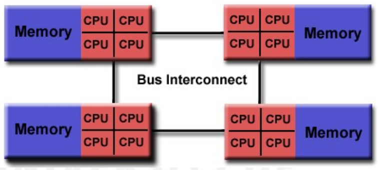
Uniform Memory Access (UMA)

- Most represented today by Symmetric Multiprocessor (SMP) machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.



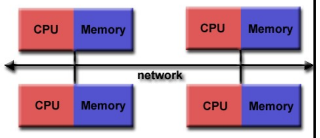
Non-Uniform Memory Access (NUMA)

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA



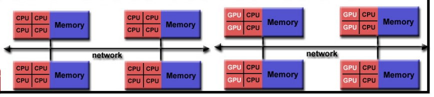
Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.



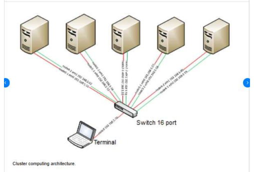
Hybrid Distributed-Shared Memory

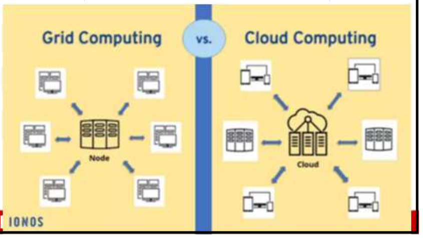
- General Characteristics
  - The largest and fastest computers in the world today employ both shared and distributed memory architectures.
  - The shared memory component can be a shared memory machine and/or graphics processing units (GPU).
  - The distributed memory component is the networking of multiple shared memory/GPU machines, which know only about their own memory - not the memory on another machine. Therefore, network communications are required to move data from one machine to another.
  - Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages
  - Whatever is common to both shared and distributed memory architectures.
  - Increased scalability is an important advantage
  - Increased programmer complexity is an important disadvantage



High performance distributed computing (HPC)

- An important class of distributed systems is the one used for high-performance computing tasks. Roughly speaking, one can make a distinction between two subgroups.
  - In **cluster computing** the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high-speed local-area network. In addition, each node runs the same operating system.
  - The situation becomes very different in the case of **grid computing**. This subgroup consists of distributed systems that are often constructed as a federation of computer systems, where each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology.
    - From the perspective of grid computing, a next logical step is to simply outsource the entire infrastructure that is needed for compute-intensive applications. In essence, this is what **cloud computing** is all about: providing the facilities to dynamically construct an infrastructure and compose what is needed from available services. Unlike grid computing, which is strongly associated with high-performance computing, cloud computing is much more than just providing lots of resources.





High performance distributed computing (HPC)

- High-performance computing started with the introduction of multiprocessor machines. In this case, multiple CPUs are organized in such a way that they all have access to the same physical memory.
- In contrast, in a multicomputer system several computers are connected through a network and there is no sharing of main memory.
- The shared-memory model proved to be highly convenient for improving the performance of programs and it was relatively easy to program.

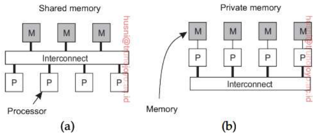


Figure 1.6: A comparison between (a) multiprocessor and (b) multicomputer architectures.

Cluster computer

- Cluster computing systems became popular when the price/performance ratio of personal computers and workstations improved.
- Cluster computing is used for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines.

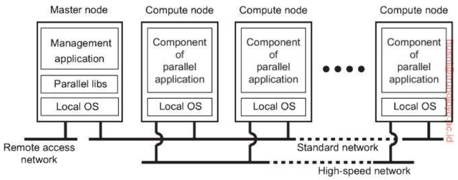


Figure 1.7: An example of a cluster computing system.

Grid computing

- A characteristic feature of traditional cluster computing is its homogeneity. In most cases, the computers in a cluster are largely the same, have the same operating system, and are all connected through the same network. However, as we just discussed, there has been a trend towards more hybrid architectures in which nodes are specifically configured for certain tasks. This diversity is even more prevalent in grid computing systems: no assumptions are made concerning similarity of hardware, operating systems, networks, administrative domains, security policies, etc.
- A key issue in a grid-computing system is that resources from different organizations are brought together to allow the collaboration of a group of people from different institutions, indeed forming a federation of systems. Such a collaboration is realized in the form of a virtual organization. The processes belonging to the same virtual organization have access rights to the resources that are provided to that organization. Typically, resources consist of compute servers (including supercomputers, possibly implemented as cluster computers), storage facilities, and databases. In addition, special networked devices such as telescopes, sensors, etc., can be provided as well.

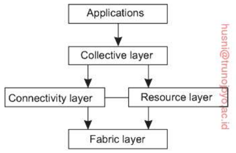


Figure 1.8: A layered architecture for grid computing systems.

Cloud computing

- How to organize computational grids that were easily accessible, organizations in charge of running data centers were facing the problem of opening up their resources to customers. Eventually, this lead to the concept of utility computing by which a customer could upload tasks to a data center and be charged on a per-resource basis. Utility computing formed the basis for what is now called cloud computing.
- Cloud computing is characterized by an easily usable and accessible pool of virtualized resources. Which and how resources are used can be configured dynamically, providing the basis for scalability: if more work needs to be done, a customer can simply acquire more resources. The link to utility computing is formed by the fact that cloud computing is generally based on a pay-per-use model in which guarantees are offered by means of customized **service level agreements (SLAs)**

Cloud computing

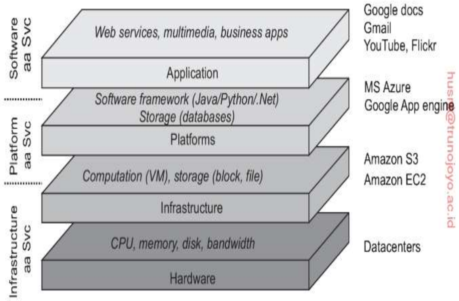
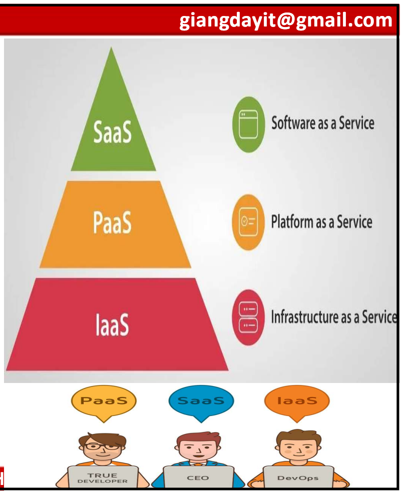


Figure 1.9: The organization of clouds (adapted from Zhang et al. [2010]).

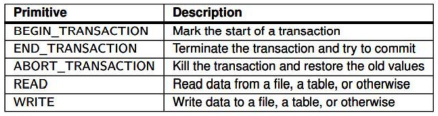


Distributed information systems

- Another important class of distributed systems is found in organizations that were confronted with a wealth of networked applications, but for which interoperability turned out to be a painful experience.
- In many cases, a networked application simply consists of a server running that application (often including a database) and making it available to remote programs, called **clients**. Such clients send a request to the server for executing a specific operation, after which a response is sent back. Integration at the lowest level allows clients to wrap a number of requests, possibly for different servers, into a single larger request and have it executed as a **distributed transaction**. The key idea is that all, or none of the requests are executed.
- As applications became more sophisticated and were gradually separated into independent components (notably distinguishing database components from processing components), it became clear that integration should also take place by letting applications communicate directly with each other. This has now lead to a huge industry that concentrates on **Enterprise Application Integration (EAI).**

Distributed transaction processing

- In practice, operations on a database are carried out in the form of transactions. Programming using transactions requires special primitives that must either be supplied by the underlying distributed system or by the language runtime system.



Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Figure 1.10: Example primitives for transactions.

- Ordinary statements, procedure calls, and so on, are also allowed inside a transaction. In particular, remote procedure calls (RPCs), that is, procedure calls to remote servers, are often also encapsulated in a transaction, leading to what is known as a transactional RPC.

ACID

- This all-or-nothing property of transactions is one of the four characteristic properties that transactions have. More specifically, transactions adhere to the so-called ACID properties:
  - **Atomic**: To the outside world, the transaction happens indivisibly
  - **Consistent**: The transaction does not violate system invariants
  - **Isolated**: Concurrent transactions do not interfere with each other
  - **Durable**: Once a transaction commits, the changes are permanent
- In distributed systems, transactions are often constructed as a number of sub-transactions, jointly forming a nested transaction

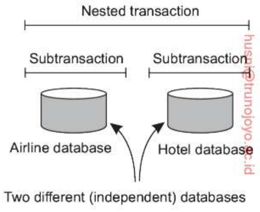


Figure 1.11: A nested transaction.

Transaction processing monitor

- In the early days of enterprise middleware systems, the component that handled distributed (or nested) transactions formed the core for integrating applications at the server or database level.
- This component was called a **transaction processing monitor** or **TP monitor** for short. Its main task was to allow an application to access multiple server/databases by offering it a transactional programming model.
- Essentially, the TP monitor coordinated the commitment of sub-transactions following a standard protocol known as **distributed commit**.

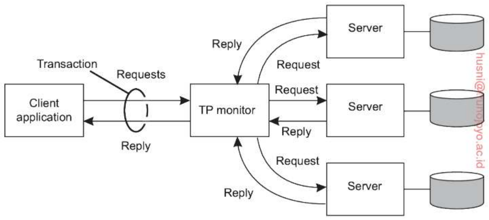


Figure 1.12: The role of a TP monitor in distributed systems.

Enterprise application integration

- Application components should be able to communicate directly with each other and not merely by means of the request/reply behavior that was supported by transaction processing systems.
- This need for inter-application communication led to many different communication models, The main idea was that existing applications could directly exchange information.

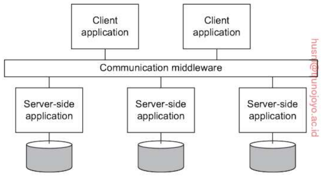


Figure 1.13: Middleware as a communication facilitator in enterprise application integration.

Enterprise application integration

- Several types of communication middleware exist. With **remote procedure calls (RPC)**, an application component can effectively send a request to another application component by doing a local procedure call, which results in the request being packaged as a message and sent to the callee. Likewise, the result will be sent back and returned to the application as the result of the procedure call.
- As the popularity of object technology increased, techniques were developed to allow calls to remote objects, leading to what is known as **remote method invocations (RMI)**. An RMI is essentially the same as an RPC, except that it operates on objects instead of functions. RPC and RMI have the disadvantage that the caller and callee both need to be up and running at the time of communication. In addition, they need to know exactly how to refer to each other. This tight coupling is often experienced as a serious drawback and has led to what is known as **message-oriented middleware**, or simply **MOM**.
- In this case, applications send messages to logical contact points, often described by means of a subject. Likewise, applications can indicate their interest for a specific type of message, after which the communication middleware will take care that those messages are delivered to those applications. These so-called **publish/subscribe** systems form an important and expanding class of distributed systems.

Pervasive systems

- The introduction of mobile and embedded computing devices, leading to what are generally referred to as **pervasive systems**.
- A pervasive system is often equipped with many **sensors** that pick up various aspects of a user's behavior.
- Many devices in pervasive systems are characterized by being small, battery-powered, mobile, and having only a wireless connection, although not all these characteristics apply to all devices. As calling **Internet of thing (IoT)**.
- Three different types of pervasive systems, although there is considerable overlap between the three types: **ubiquitous computing** systems, **mobile** systems, and **sensor** networks.

Ubiquitous computing systems

- In a ubiquitous computing system, we go one step further: the system is pervasive and continuously present. The latter means that a user will be continuously interacting with the system, often not even being aware that interaction is taking place.
  - (**Distribution**) Devices are networked, distributed, and accessible in a transparent manner
  - (**Interaction**) Interaction between users and devices is highly unobtrusive
  - (**Context awareness**) The system is aware of a user's context can optimize interaction
  - (**Autonomy**) Devices operate autonomously without human intervention, and are thus highly self-managed
    - *Address allocation*: For networked devices to communicate, they need an IP address. Addresses can be allocated automatically using protocols like the Dynamic Host Configuration Protocol (DHCP)
    - *Adding devices*: It should be easy to add devices to an existing system. A step towards automatic configuration is realized by the Universal Plug and Play Protocol (UPnP). Using UPnP, devices can discover each other and make sure that they can set up communication channels between them.
    - *Automatic updates*: Many devices in a ubiquitous computing system should be able to regularly check through the Internet if their software should be updated. If so, they can download new versions of their components and ideally continue where they left off.
  - (**Intelligence**) The system can handle a wide range of dynamic actions and interactions

Mobile computing systems

- Mobility often forms an important component of pervasive systems.
- There are several issues that set mobile computing aside to pervasive systems in general.
  - First, the devices that form part of a (distributed) mobile system may vary widely. Typically, mobile computing is now done with devices such as smartphones and tablet computers. However, completely different types of devices are now using the Internet Protocol (IP) to communicate, placing mobile computing in a different perspective. Such devices include remote controls, pagers, active badges, car equipment, various GPS-enabled devices, and so on. A characteristic feature of all these devices is that they use wireless communication. Mobile implies wireless so it seems (although there are exceptions to the rules).
  - Second, in mobile computing the location of a device is assumed to change over time. A changing location has its effects on many issues. Consequently, we may need to pay special attention to dynamically discovering services, but also letting services announce their presence. So needing to know the actual geographical coordinates of a device such as in tracking and tracing applications

Mobile computing systems

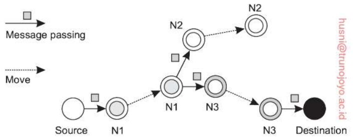


Figure 1.14: Passing messages in a (mobile) disruption-tolerant network.

- Changing locations also has a profound effect on communication. To illustrate, consider a (wireless) mobile ad hoc network, generally abbreviated as a MANET. Suppose that two devices in a MANET have discovered each other in the sense that they know each other's network address. How do we route messages between the two? Static routes are generally not sustainable as nodes along the routing path can easily move out of their neighbor's range, invalidating the path. For large MANETs, using a priori set-up paths is not a viable option. What we are dealing with here are so-called disruption-tolerant networks: networks in which connectivity between two nodes can simply not be guaranteed. Getting a message from one node to another may then be problematic, to say the least.

Sensor networks

- A sensor network generally consists of tens to hundreds or thousands of relatively small nodes, each equipped with one or more sensing devices. In addition, nodes can often act as actuators, a typical example being the automatic activation of sprinklers when an event has been detected.
- Many sensor networks use wireless communication, and the nodes are often battery powered. Their limited resources, restricted communication capabilities, and constrained power consumption demand that efficiency is high on the list of design criteria.
- To organize a sensor network as a distributed database, there are essentially two extremes, as shown in Figure 1.16. First, sensors do not cooperate but simply send their data to a centralized database located at the operator's site. The other extreme is to forward queries to relevant sensors and to let each compute an answer, requiring the operator to aggregate the responses.

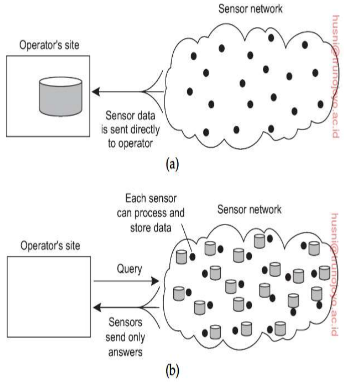


Figure 1.16: Organizing a sensor network database, while storing and processing data (a) only at the operator's site or (b) only at the sensors.

Summary

- Distributed systems consist of autonomous computers that work together to give the appearance of a single coherent system. This combination of independent, yet coherent collective behavior is achieved by collecting application independent protocols into what is known as middleware: a software layer logically placed between operating systems and distributed applications. Protocols include those for communication, transactions, service composition, and perhaps most important, reliability.
- Design goals for distributed systems include sharing resources and ensuring openness. In addition, designers aim at hiding many of the intricacies related to the distribution of processes, data, and control.

THANK YOU

Q & A