Lecture 12

presentation

DAD – Distributed & Cloud App Development

Cristian Toma

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Agenda for Lecture 12

Distributed Systems and Distributed Applications Development | Cloud

Distributed Systems

It’s not just about the programming, but providing smart solutions

DAD Sections & References

1.3 Distributed Systems

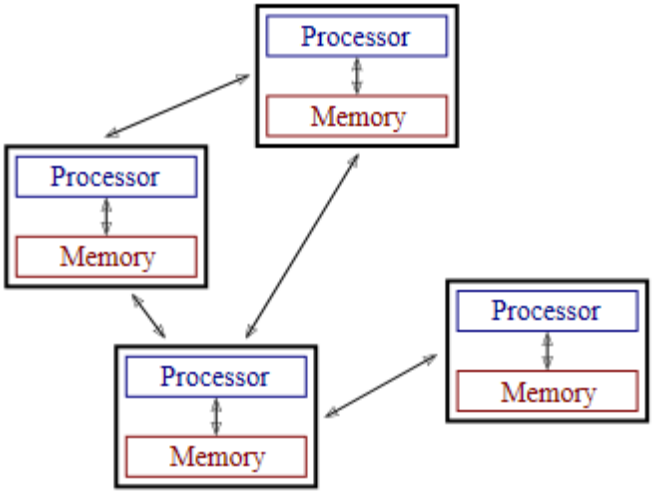
What is a Distributed System?

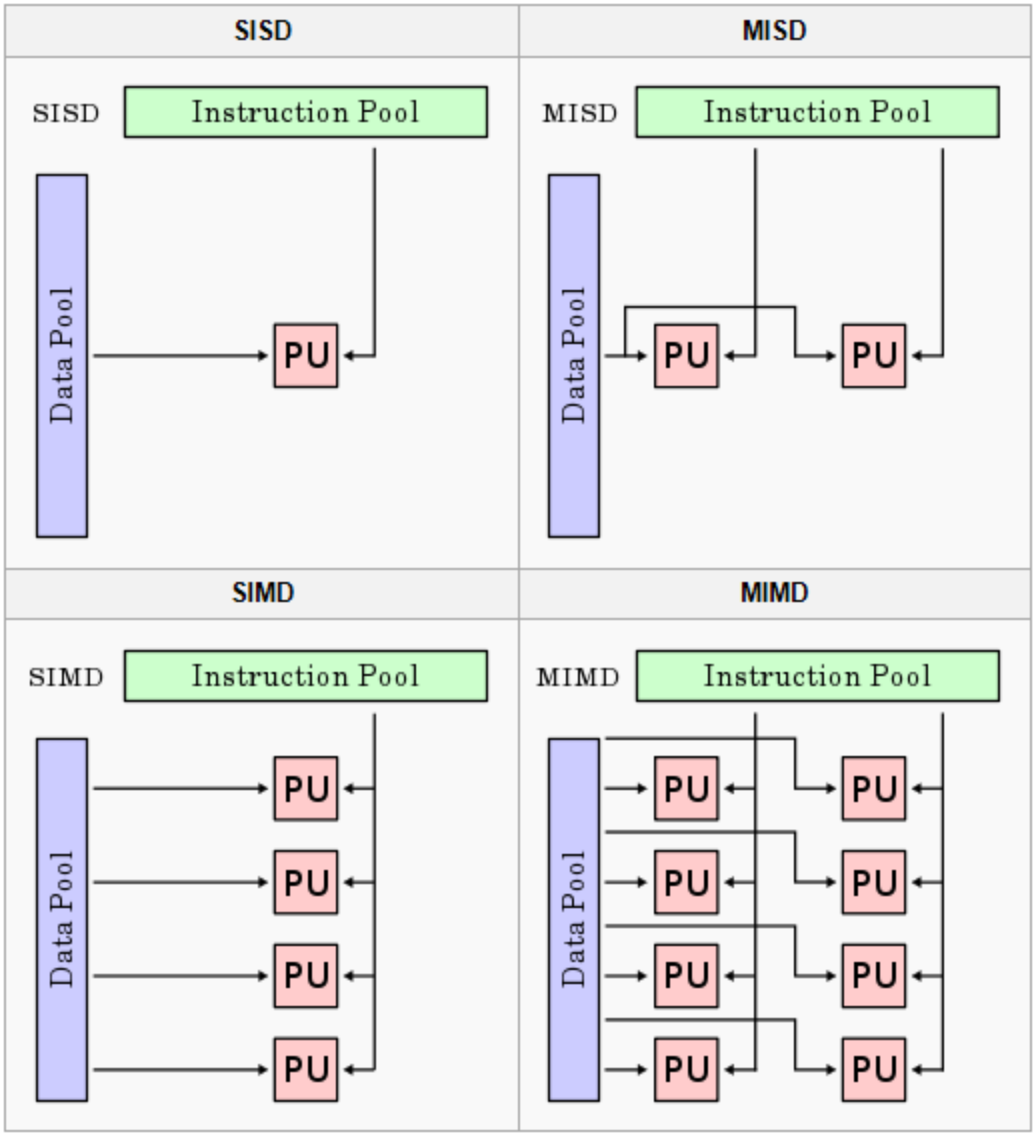
A distributed system is one in which components located into computers connected in network, communicate and coordinate their actions only by passing messages (sometimes, in addition, by migrating processes), in order to resolve a set of problems.

A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. This software enables computers to coordinate their activities and to share the re- sources of the system hardware, software, and data.

Flynn Taxonomy
Parallel vs. Distributed Systems

Parallel vs. Distributed Computing / Algorithms





Where is the picture for:
Distributed System and ***Parallel System***?

http://en.wikipedia.org/wiki/Distributed_computing

http://en.wikipedia.org/wiki/Flynn's_taxonomy

1.3 Distributed Systems

How to characterize a distributed system?

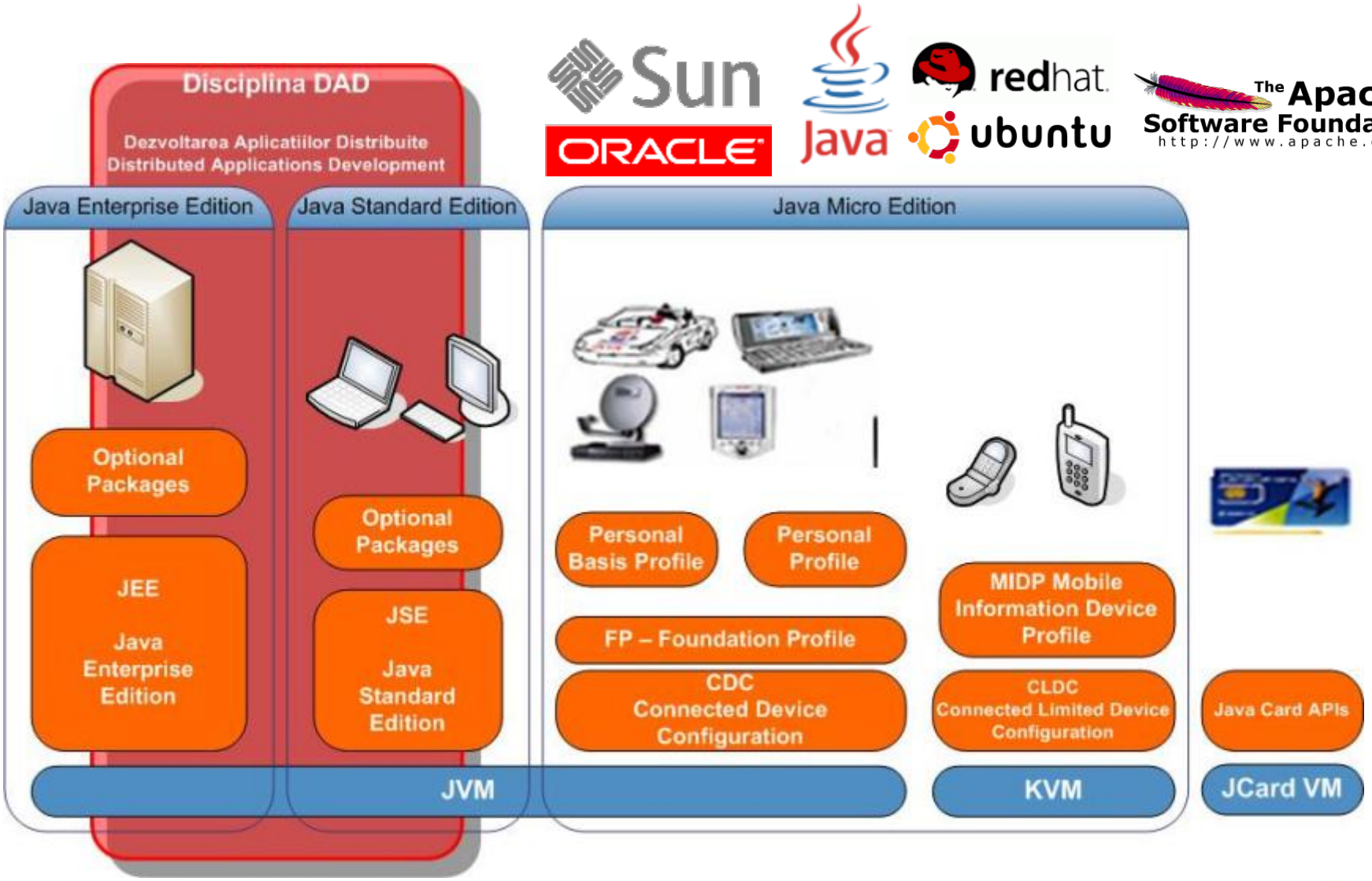
- concurrency of components
- lack of global clock
- independent failures of components

Leslie Lamport :-)

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done!

Prime motivation = to share the resources

Java – SE & EE 70% + C/C++ 5 % + Node.js 5 % + OS Admin 5 % + Distributed Frameworks 15 %



1.3 Distributed Systems

What are the challenges?

- heterogeneity of their components
- openness
- security
- scalability – the ability to work well when the load or the number of users increases
- failure handling
- concurrency of components
- transparency
- providing quality of service

1.3 Distributed Systems

What are the resources?

- Hardware
  - Not every single resource is for sharing
- Data
  - Databases
  - Proprietary software
  - Software production
  - Collaboration

Sharing Resources

• Different resources are handled in different ways, there are however some generic requirements:

– Namespace for identification
– Name translation to network address
– Synchronization of multiple access

1.3 Distributed Systems

Concept of Distributed Architecture

A distributed system can be demonstrated by the client-server architecture, which forms the base for multi-tier architectures; alternatives are the broker architecture such as CORBA, and the Service-Oriented Architecture (SOA). In this architecture, information processing is not confined to a single machine rather it is distributed over several independent computers.

There are several technology frameworks to support distributed architectures, including Java EE / Java-Kotlin Reactive Micro-services, .NET, CORBA, Scala Akka, Globus Toolkit Grid services, etc..

Middleware is an infrastructure that appropriately supports the development and execution of distributed applications.

1.3 Distributed Systems

Basis of Distributed Architecture

The basis of a distributed architecture is its transparency, reliability, and availability.

The following table lists the different forms of transparency in a distributed system

Transparency	Description
Access	Hides the way in which resources are accessed and the differences in data platform.
Location	Hides where resources are located.
Technology	Hides different technologies such as programming language and OS from user.
Migration / Relocation	Hide resources that may be moved to another location which are in use.
Replication	Hide resources that may be copied at several location.
Concurrency	Hide resources that may be shared with other users.
Failure	Hides failure and recovery of resources from user.
Persistence	Hides whether a resource (software) is in memory or disk.

1.3 Distributed Systems

Distributed Systems Advantages

It has following advantages:

- Resource sharing − Sharing of hardware and software resources.
- Openness − Flexibility of using hardware and software of different vendors.
- Concurrency − Concurrent processing to enhance performance.
- Scalability − Increased throughput by adding new resources.
- Fault tolerance − The ability to continue in operation after a fault has occurred.

Distributed Systems Disadvantages

Its disadvantages are:

- Complexity − They are more complex than centralized systems.
- Security − More susceptible to external attack.
- Manageability − More effort required for system management.
- Unpredictability − Unpredictable responses depending on the system organization and network load.

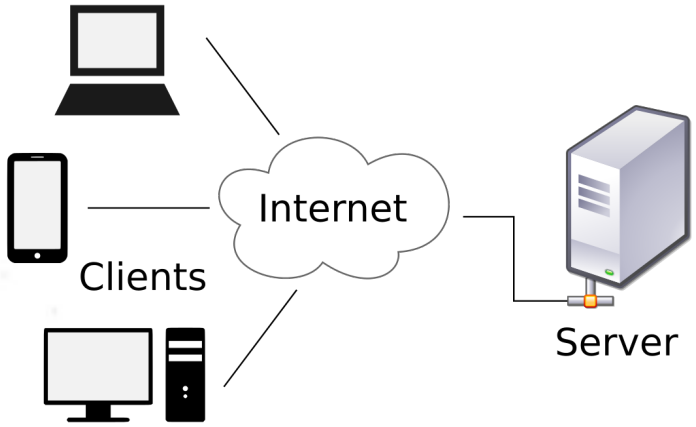
1.3 Distributed Systems

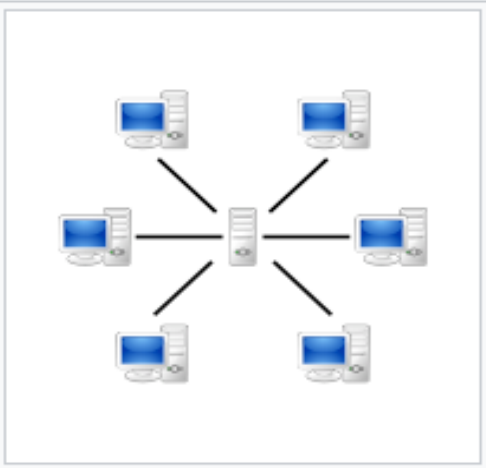
Client-Server Architecture

The client-server architecture is the most common distributed system architecture which decomposes the system into two major subsystems or logical processes −

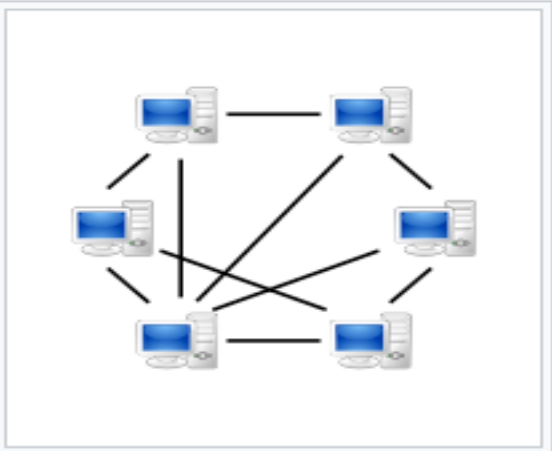
- Client − This is the first process that issues a request to the second process i.e. the server.
- Server − This is the second process that receives the request, carries it out, and sends a reply to the client.

In this architecture, the application is modelled as a set of services those are provided by servers and a set of clients that use these services. The servers need not to know about clients, but the clients must know the identity of servers.





A network based on the **client-server model**, where individual *clients* request services and resources from centralized servers



A **peer-to-peer (P2P) network** in which interconnected nodes ("peers") share resources amongst each other without the use of a centralized administrative system

1.3 Distributed Systems

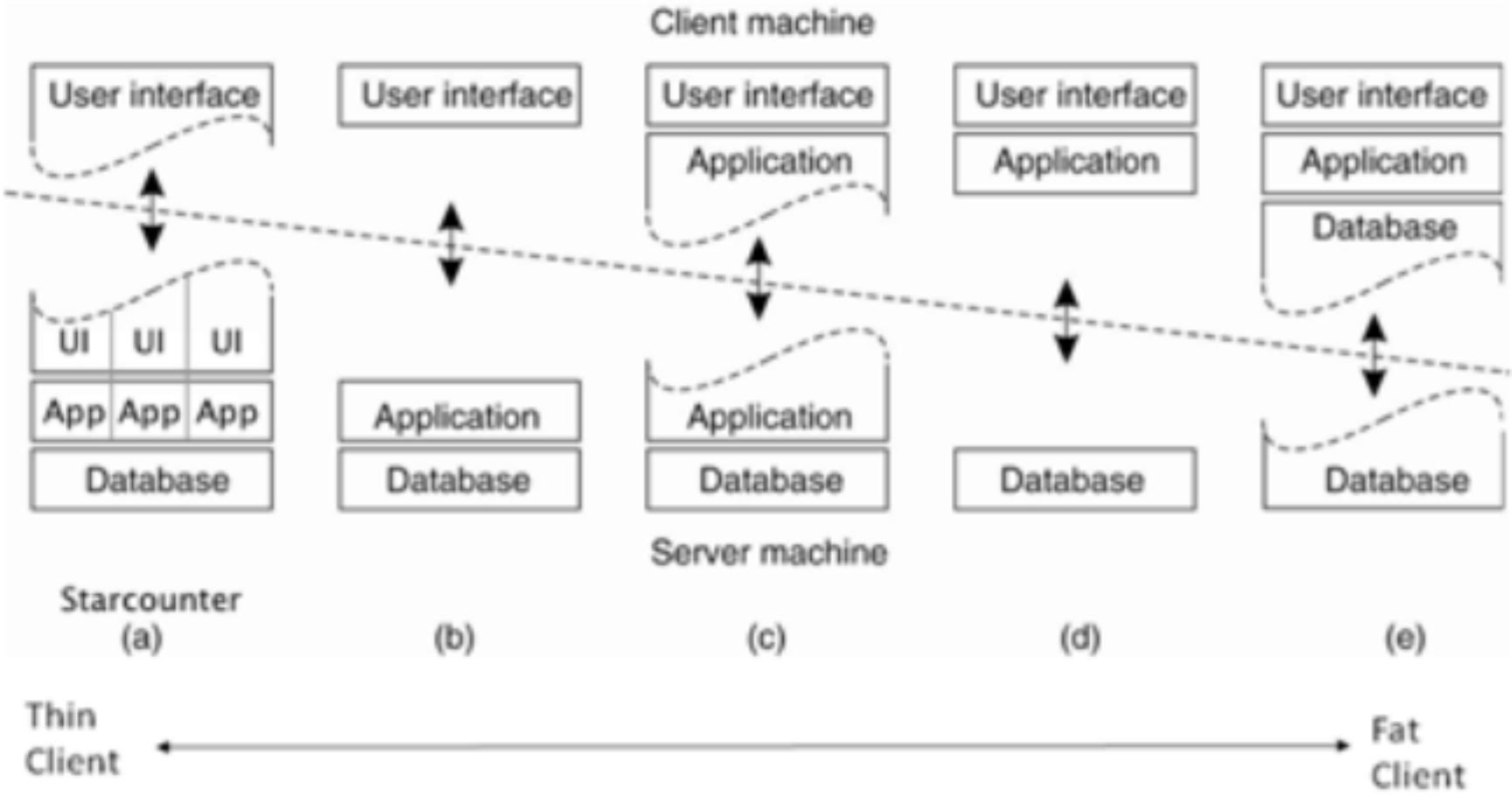
Thin-client model

In thin-client model, all the application processing and data management is carried by the server. The client is simply responsible for running the GUI software. It is used when legacy systems are migrated to client server architectures in which legacy system acts as a server in its own right with a graphical interface implemented on a client.
However, A major disadvantage is that it places a heavy processing load on both the server and the network.

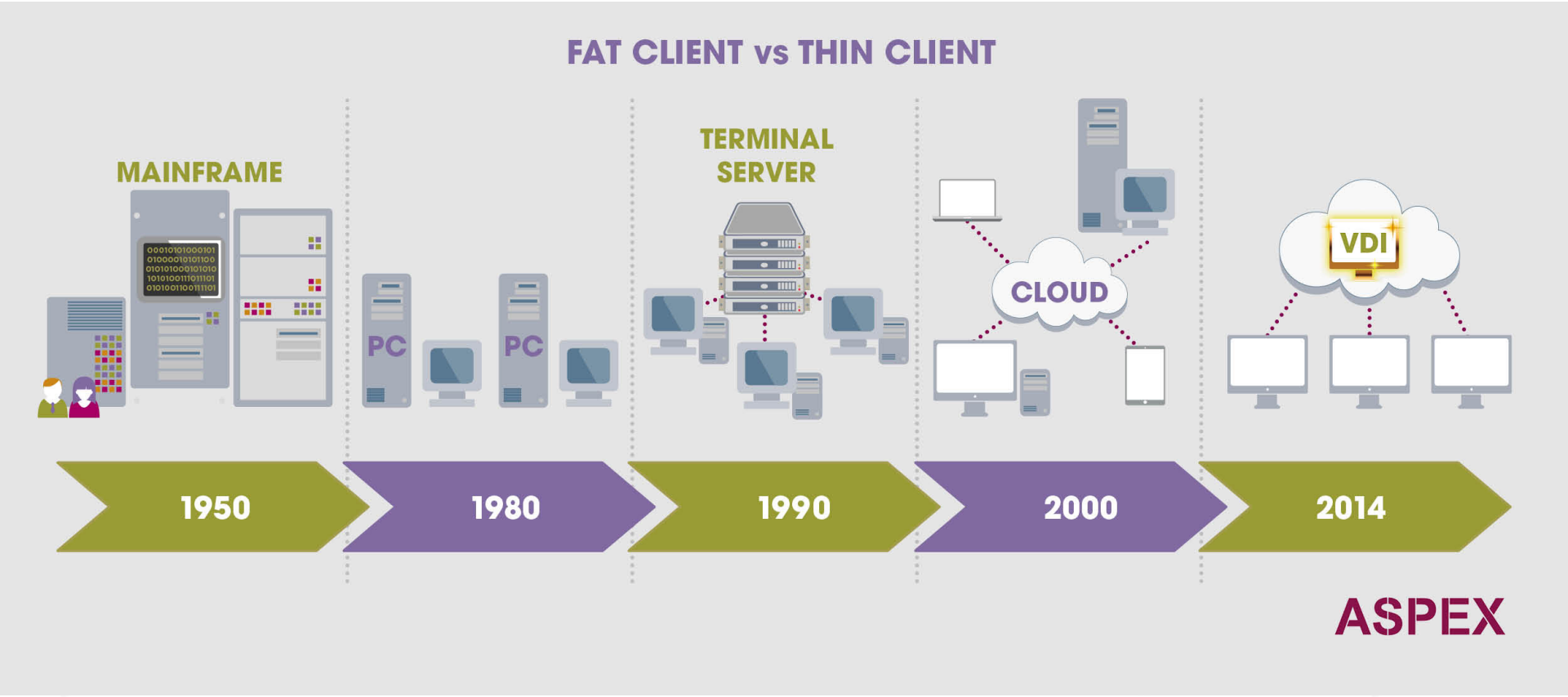
Thick/Fat-client model

In thick-client model, the server is in-charge of the data management. The software on the client implements the application logic and the interactions with the system user. It is most appropriate for new client-server systems where the capabilities of the client system are known in advance.
However, it is more complex than a thin client model especially for management, as all clients should have same copy/version of software application.

Thin vs. Thick/Fat Client



Thin vs. Thick/Fat Client



1.3 Distributed Systems

Thin/Thick Clients Advantages:

- Separation of responsibilities such as user interface presentation and business logic processing.
- Reusability of server components and potential for concurrency
- Simplifies the design and the development of distributed applications
- It makes it easy to migrate or integrate existing applications into a distributed environment.
- It also makes effective use of resources when a large number of clients are accessing a high-performance server.

Thin/Thick Clients Disadvantages:

- Lack of heterogeneous infrastructure to deal with the requirement changes.
- Security complications.
- Limited server availability and reliability.
- Limited testability and scalability.
- Fat clients with presentation and business logic together.

1.3 Distributed Systems

Multi-Tier Architecture (n-tier Architecture)

Multi-tier architecture is a client–server architecture in which the functions such as presentation, application processing, and data management are physically separated. By separating an application into tiers, developers obtain the option of changing or adding a specific layer, instead of reworking the entire application. It provides a model by which developers can create flexible and reusable applications.

The most general use of multi-tier architecture is the ***three-tier architecture***. A three-tier architecture is typically composed of a presentation tier, an application tier, and a data storage tier and may execute on a separate processor.

Presentation Tier

Presentation layer is the topmost level of the application by which users can access directly such as webpage or Operating System GUI (Graphical User interface). The primary function of this layer is to translate the tasks and results to something that user can understand. It communicates with other tiers so that it places the results to the browser/client tier and all other tiers in the network.

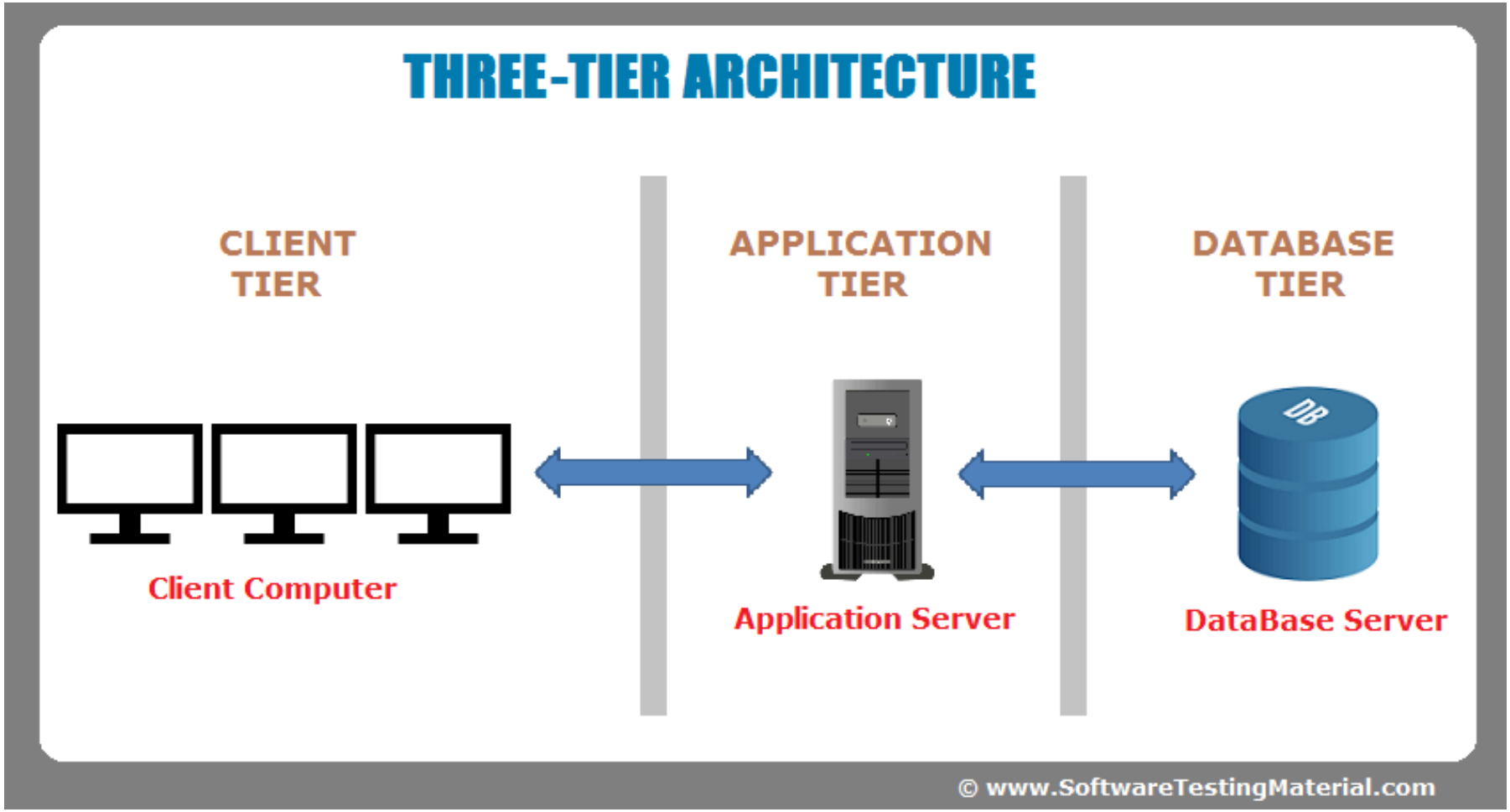
Application Tier (Business Logic, Logic Tier, or Middle Tier)

Application tier coordinates the application, processes the commands, makes logical decisions, evaluation, and performs calculations. It controls an application's functionality by performing detailed processing. It also moves and processes data between the two surrounding layers.

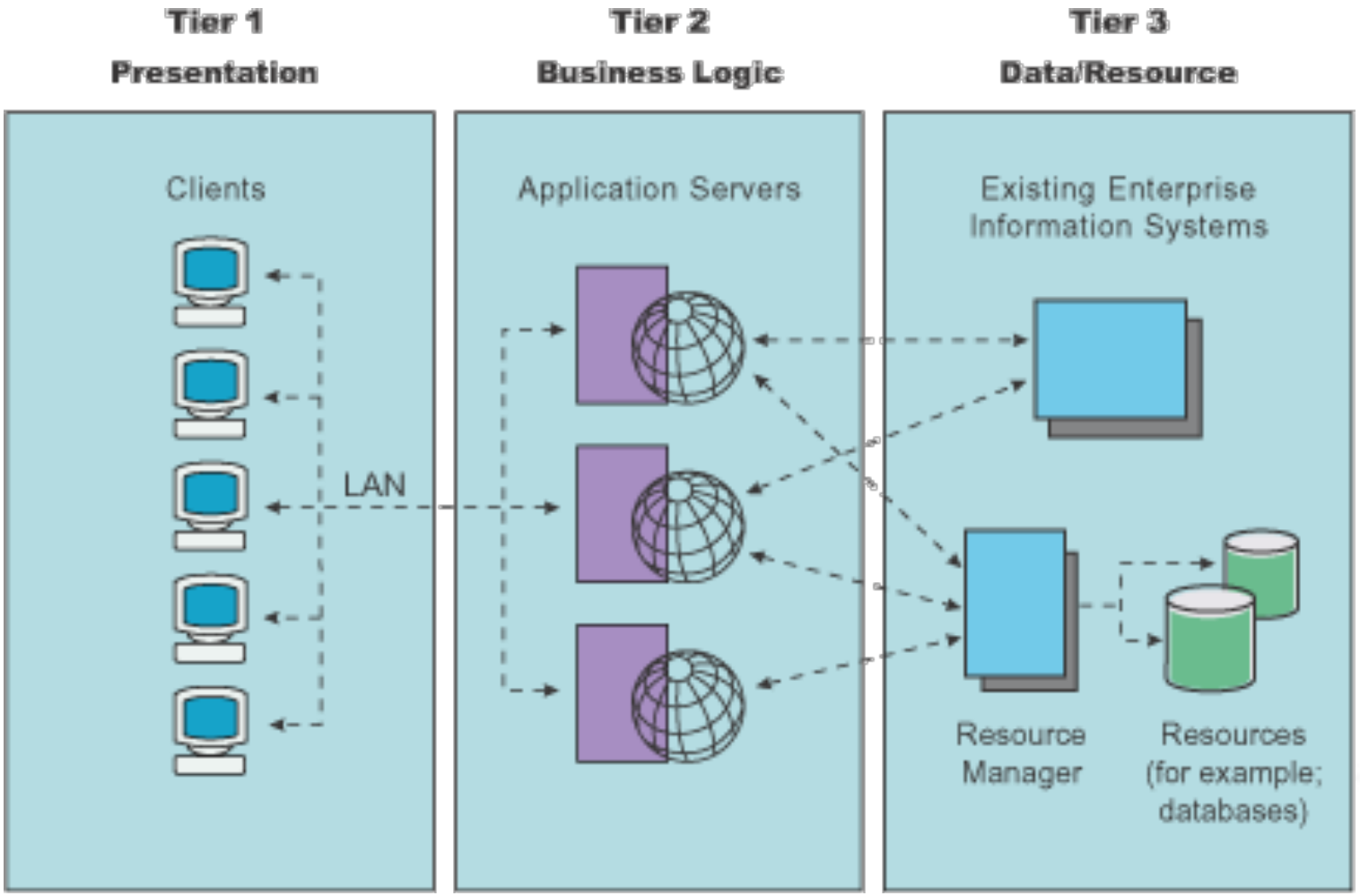
Data Tier

In this layer, information is stored and retrieved from the database or file system. The information is then passed back for processing and then back to the user. It includes the data persistence mechanisms (database servers, file shares, etc.) and provides API (Application Programming Interface) to the application tier which provides methods of managing the stored data.

3-Tier Architecture

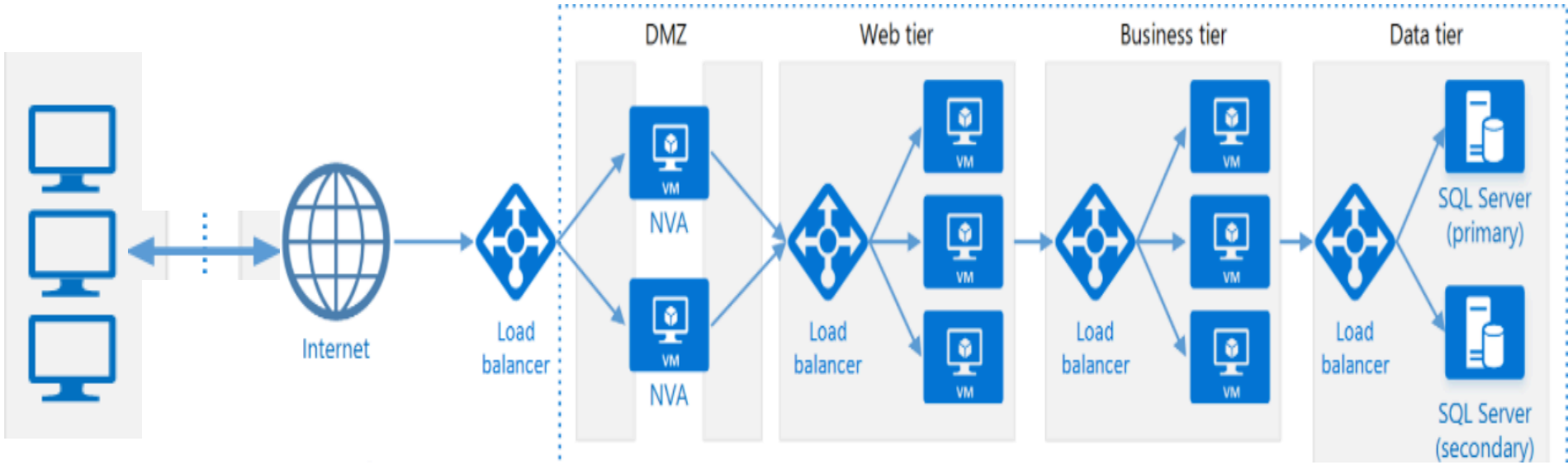


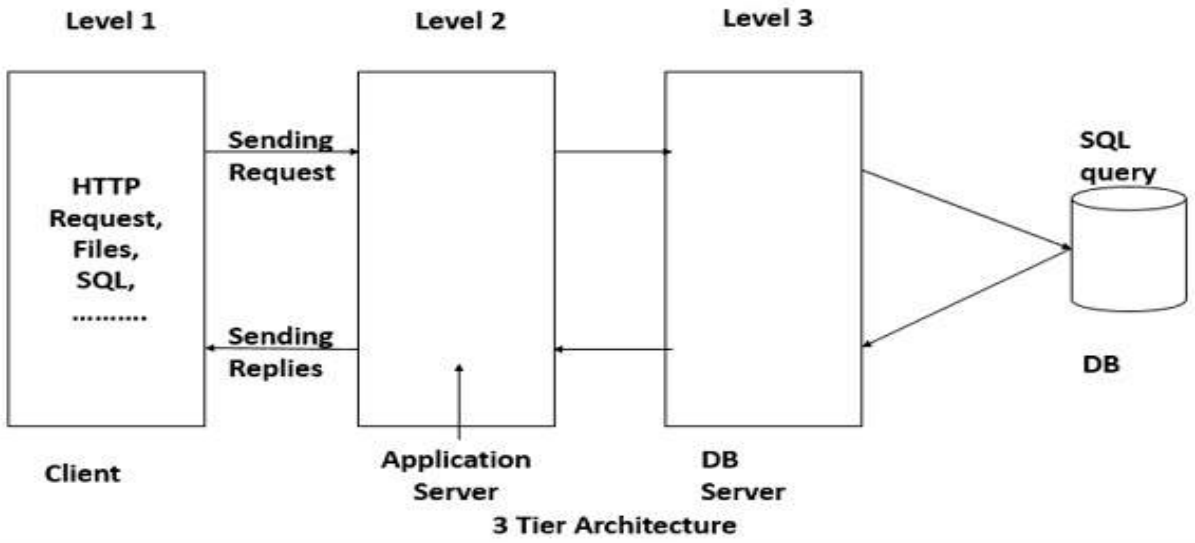
3-Tier Architecture



1.3 Distributed Systems

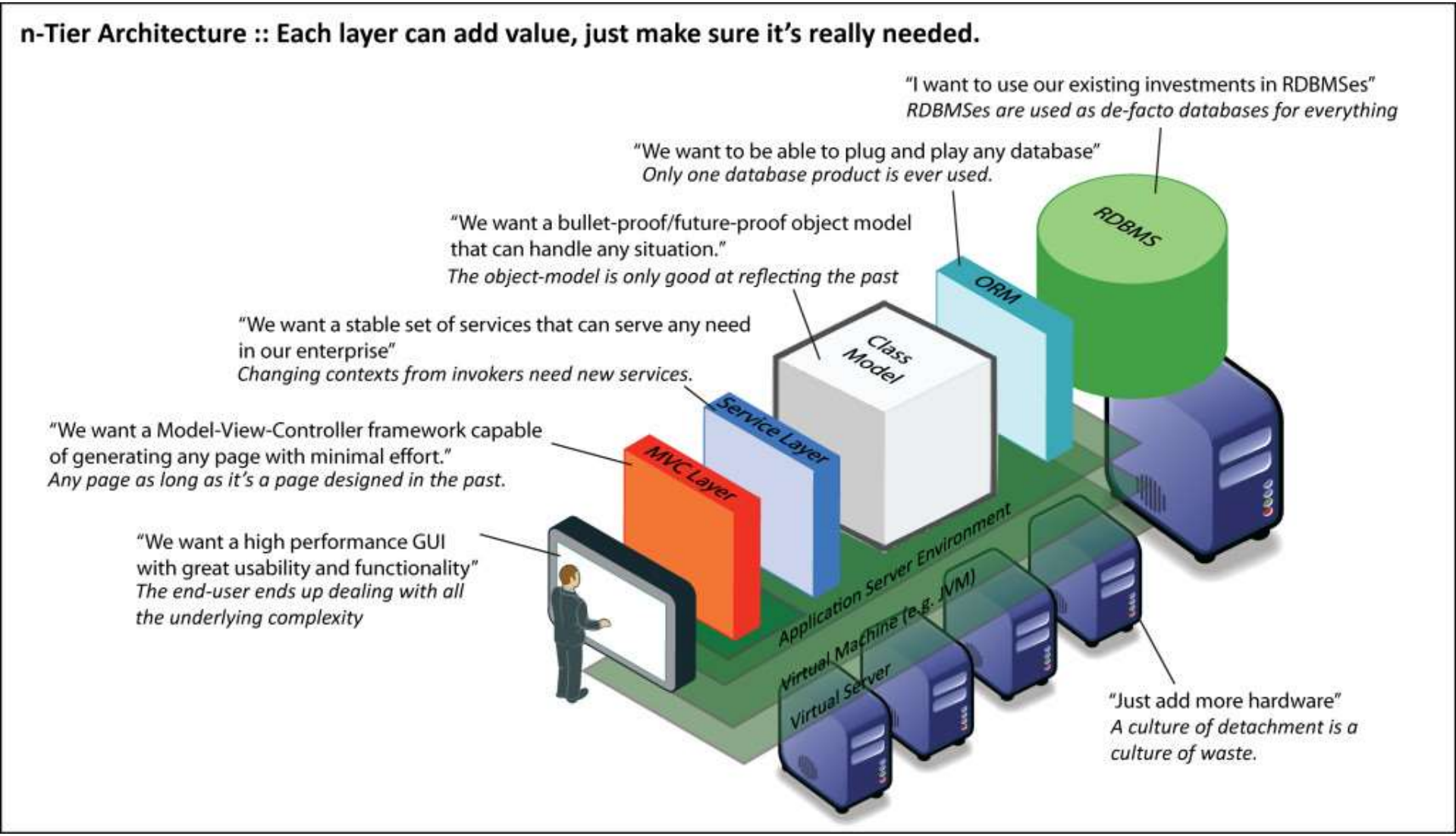
3-Tier/4-Tier Architecture





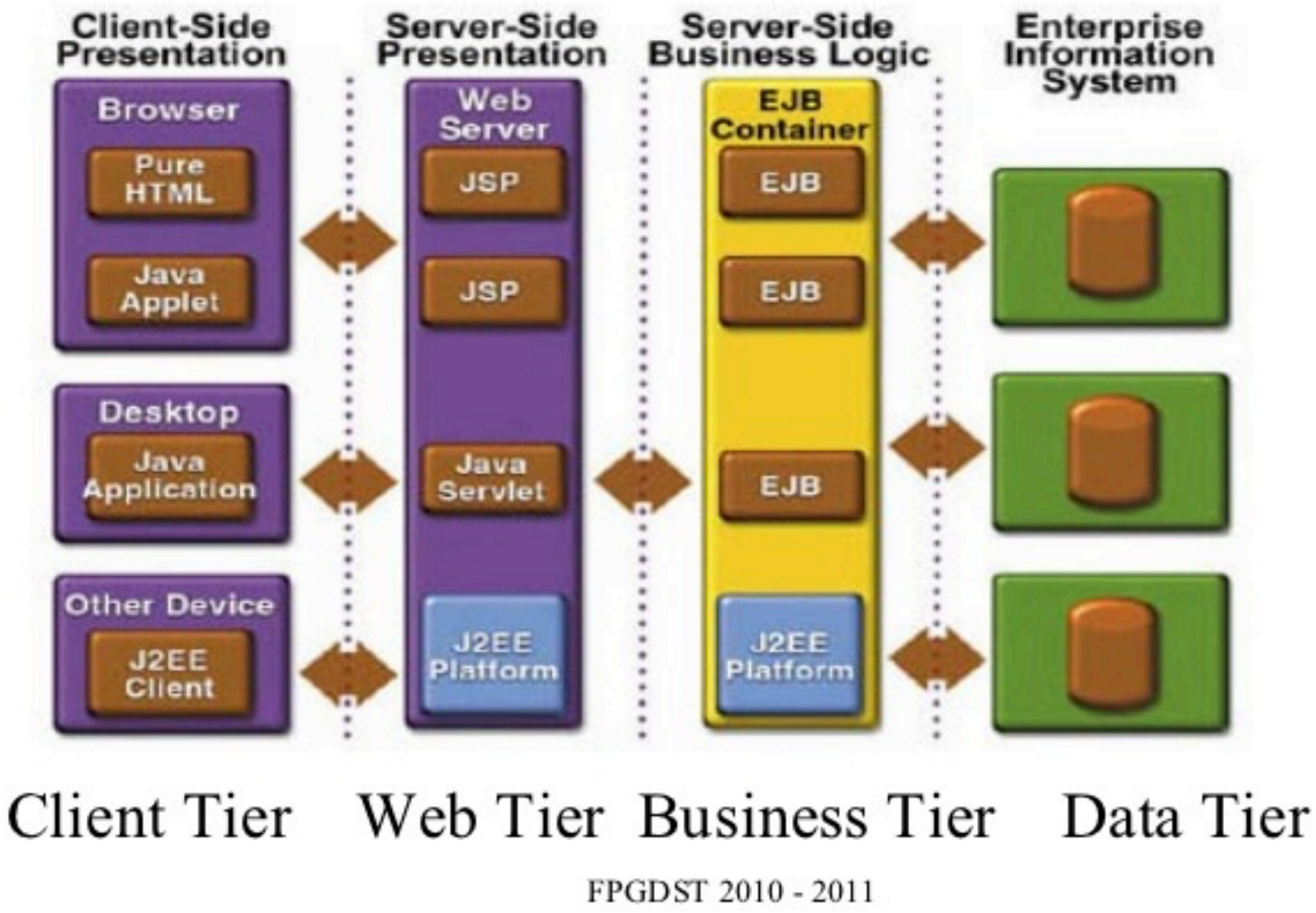
1.3 Distributed Systems

n-Tier Architecture



1.3 Distributed Systems

n-Tier Architecture



1.3 Distributed Systems

n-Tier Architecture

Advantages

- Better performance than a thin-client approach and is simpler to manage than a thick-client approach.
- Enhances the reusability and scalability − as demands increase, extra servers can be added.
- Provides multi-threading support and also reduces network traffic.
- Provides maintainability and flexibility.

Disadvantages

- Unsatisfactory Testability due to lack of testing tools.
- More critical server reliability and availability.

Broker Architectural Style

Broker Architectural Style is a middleware architecture used in distributed computing to coordinate and enable the communication between registered servers and clients. Here, object communication takes place through a middleware system called an object request broker (software bus).

However, client and the server do not interact with each other directly. Client and server have a direct connection to its proxy, which communicates with the mediator-broker. A server provides services by registering and publishing their interfaces with the broker and clients can request the services from the broker statically or dynamically by look-up.

CORBA (Common Object Request Broker Architecture) is a good implementation example of the broker architecture.

Broker Architectural Components

Components of Broker Architectural Style - The components of broker architectural style are discussed through following heads:

Broker

Broker is responsible for brokering the service requests, locating a proper server, transmitting requests, and sending responses back to clients. It also retains the servers' registration information including their functionality and services as well as location information.

Further, it provides APIs for clients to request, servers to respond, registering or unregistering server components, transferring messages, and locating servers.

Stub

Stubs are generated at the static compilation time and then deployed to the client side which is used as a proxy for the client. Client-side proxy acts as a mediator between the client and the broker and provides additional transparency between them and the client; a remote object appears like a local one.

The proxy hides the IPC (inter-process communication) at protocol level and performs marshaling of parameter values and un-marshaling of results from the server.

Skeleton

Skeleton is generated by the service interface compilation and then deployed to the server side, which is used as a proxy for the server. Server-side proxy encapsulates low-level system-specific networking functions and provides high-level APIs to mediate between the server and the broker.

Further, it receives the requests, unpacks the requests, unmarshals the method arguments, calls the suitable service, and also marshals the result before sending it back to the client.

Bridge

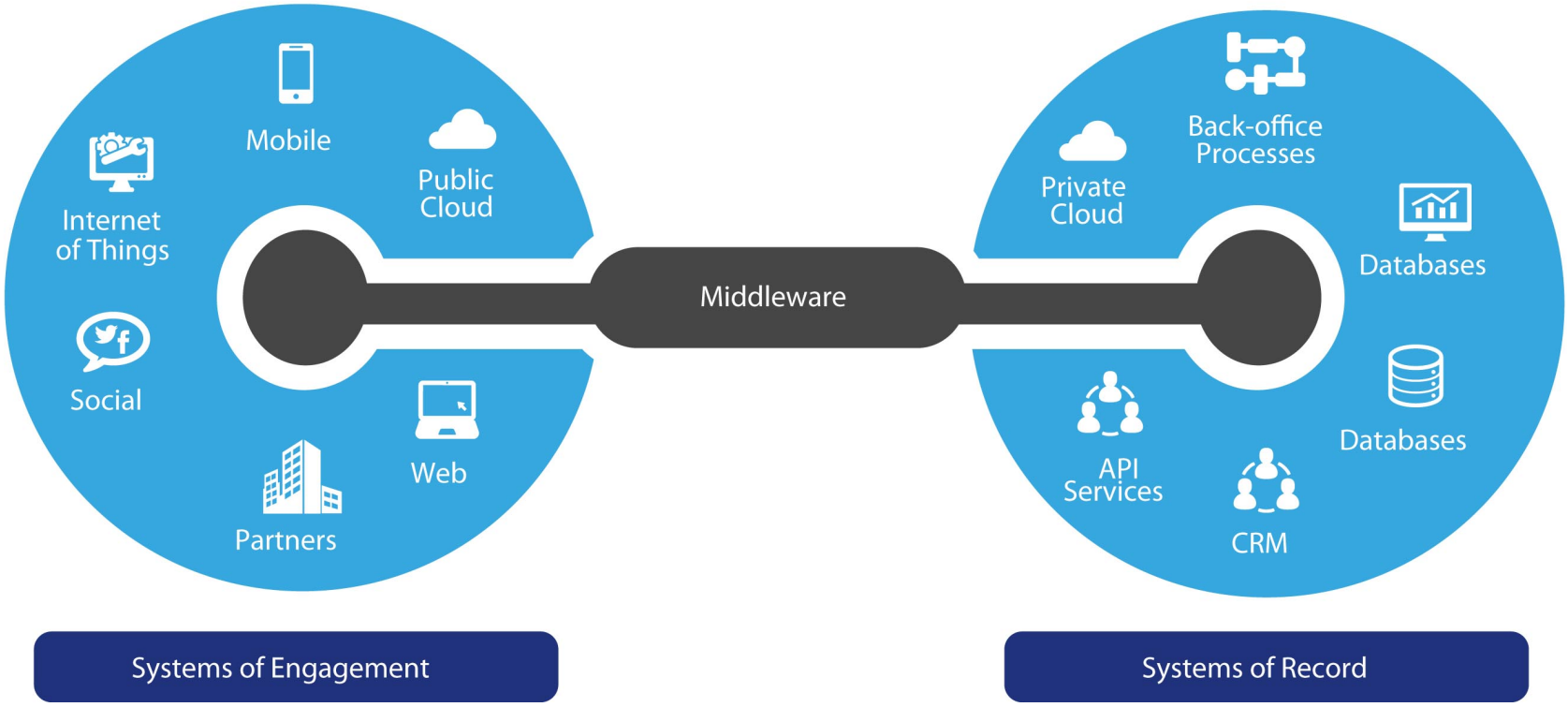
A bridge can connect two different networks based on different communication protocols. It mediates different brokers including DCOM, .NET remote, and Java CORBA brokers.

Bridges are optional component, which hides the implementation details when two brokers interoperate and take requests and parameters in one format and translate them to another format.

1.3 Distributed Systems

Middleware API

Middleware is computer software that provides services to software applications beyond those available from the operating system. It can be described as "software glue".



1.3 Distributed Systems

Middleware

The term is most commonly used for software that enables communication and management of data in distributed applications. An IETF workshop in 2000 defined middleware as "those services found above the transport (i.e. over TCP/IP) layer set of services but below the application environment" (i.e. below application-level APIs).[3] In this more specific sense *middleware* can be described as the dash ("-") in *client-server*, or the *-to-* in *peer-to-peer*.[*citation needed*] Middleware includes web servers, application servers, content management systems, and similar tools that support application development and delivery.

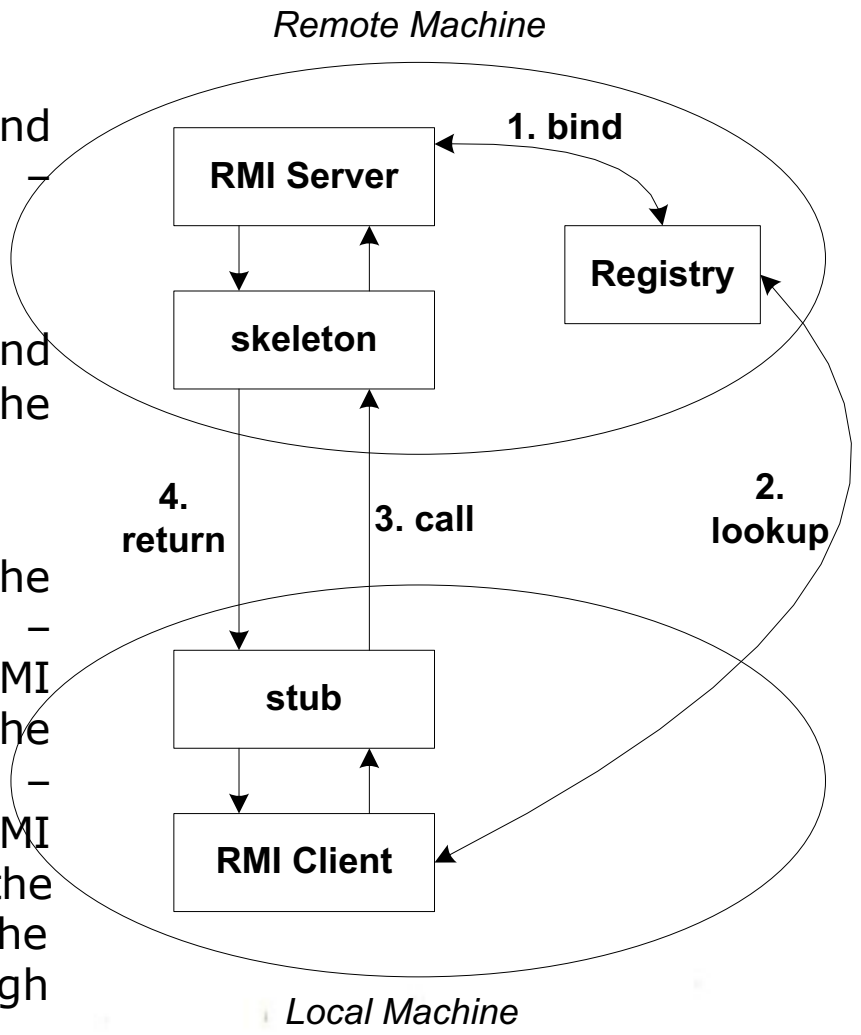
ObjectWeb defines middleware as: "The software layer that lies between the operating system and applications on each side of a distributed computing system in a network."[4]

Services that can be regarded as middleware include: enterprise application integration, data integration,
Remote Procedure Call (RPC)
Message Oriented Middleware (MOM),
Object Request Brokers (ORBs),
Service Oriented Architecture (SOA),
and the Enterprise Service Bus (ESB).

Database access services are often characterized as middleware. Some of them are language specific implementations and support heterogeneous features and other related communication features. Examples of database-oriented middleware include ODBC, JDBC and transaction processing monitors.

RPC/RMI Architecture

- RMI Server must register its name and address in the RMI Registry program – **bind**
- RMI Client is looking for the address and the name of the RMI server object in the RMI Registry program – **lookup**
- RMI Stub serializes and transmits the parameters of the call in sequence – ***"marshalling"*** to the RMI Skeleton. RMI Skeleton de-serializes and extracts the parameters from the received call – ***"unmarshalling".*** In the end, the RMI Skeleton calls the method inside the server object and send back the response to the RMI Stub through "marshalling" the return's parameter.

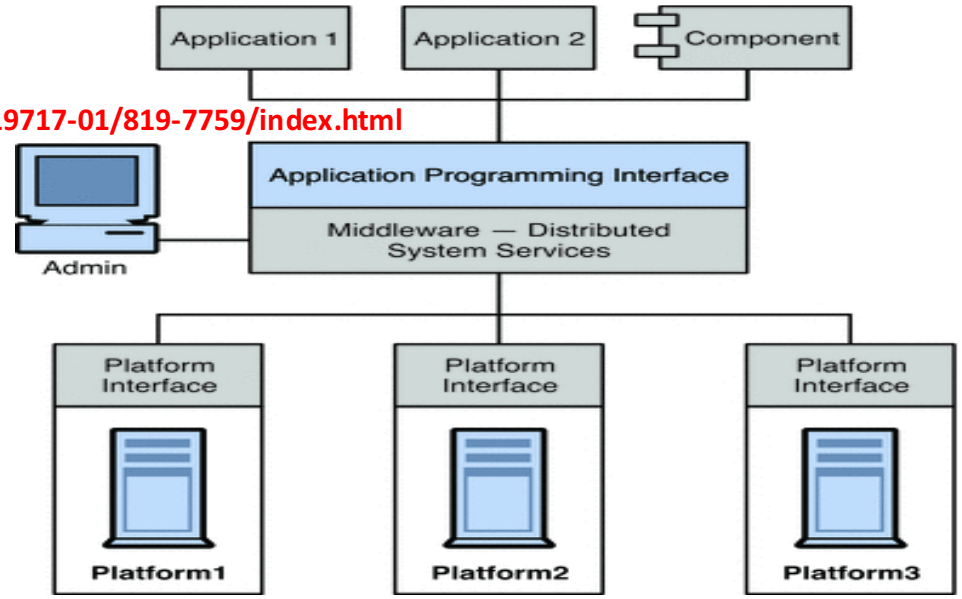


Middleware Technologies:

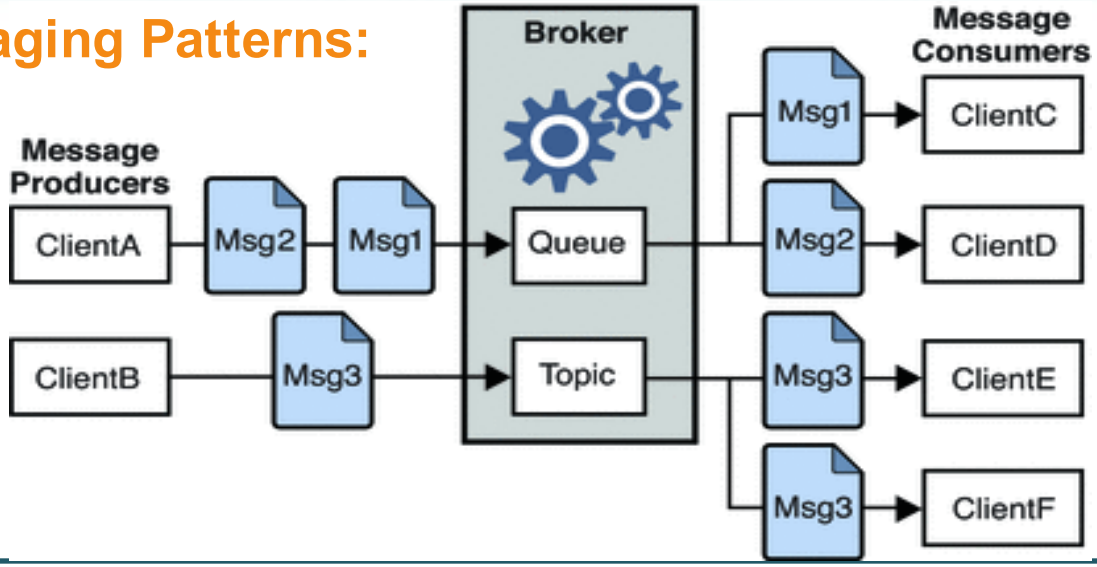
Middleware can be grouped into the following categories:

- ***Remote Procedure Call or RPC***-based middleware, which allows procedures in one application to call procedures in remote applications as if they were local calls. The middleware implements a linking mechanism that locates remote procedures and makes these transparently available to a caller. Traditionally, this type of middleware handled procedure-based programs; it now also includes object-based components.
- ***Message Oriented Middleware or MOM***-based middleware, which allows distributed applications to communicate and exchange data by sending and receiving messages asynchronously.
- ***Object Request Broker or ORB***-based middleware, which enables an application's objects to be distributed and shared across heterogeneous networks.

http://docs.oracle.com/cd/E19717-01/819-7759/index.html



JMS Messaging Patterns:



Clients A and B are message producers, sending messages to clients C, D, and E by way of two different kinds of destinations.

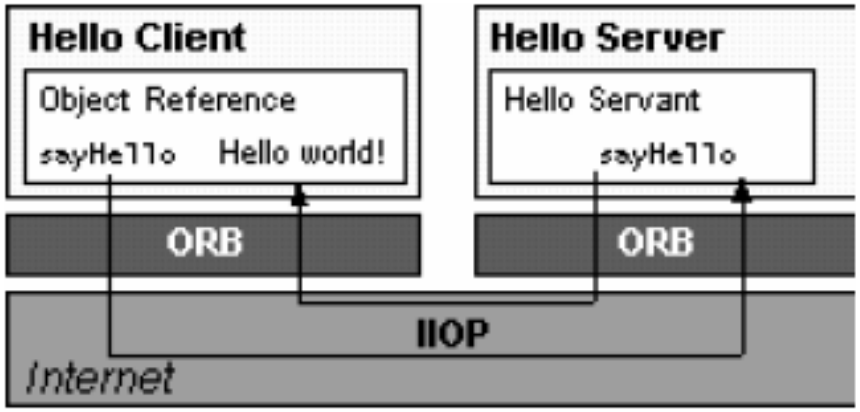
- Messaging between clients A, C, and D illustrates the point-to-point pattern. Using this pattern, a client sends a message to a queue destination from which only one receiver may get it. No other receiver accessing that destination can get that message.
- Messaging between clients B, E, and F illustrates the publish/subscribe pattern. Using this broadcast pattern, a client sends a message to a topic destination from which any number of consuming subscribers can retrieve it. Each subscriber gets its own copy of the message.

Message consumers in either domain can choose to get messages synchronously or asynchronously. Synchronous consumers make an explicit call to retrieve a message; asynchronous consumers specify a callback method that is invoked to pass a pending message. Consumers can also filter out messages by specifying selection criteria for incoming messages.

http://docs.oracle.com/cd/E19717-01/819-7759/index.html

1.3 Distributed Systems: CORBA

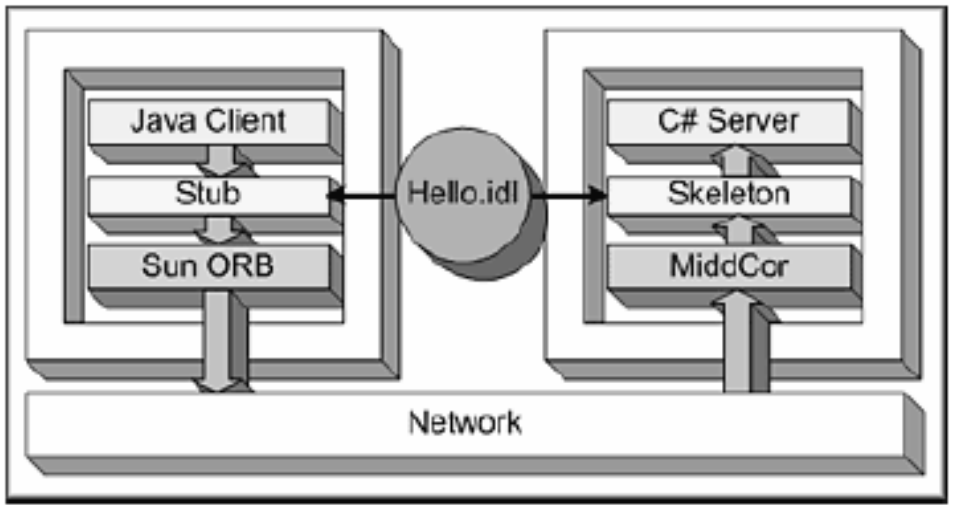
CORBA Architecture



From: http://java.sun.com/docs/books/tutorial/idl/hello/index.html

Hello.idl

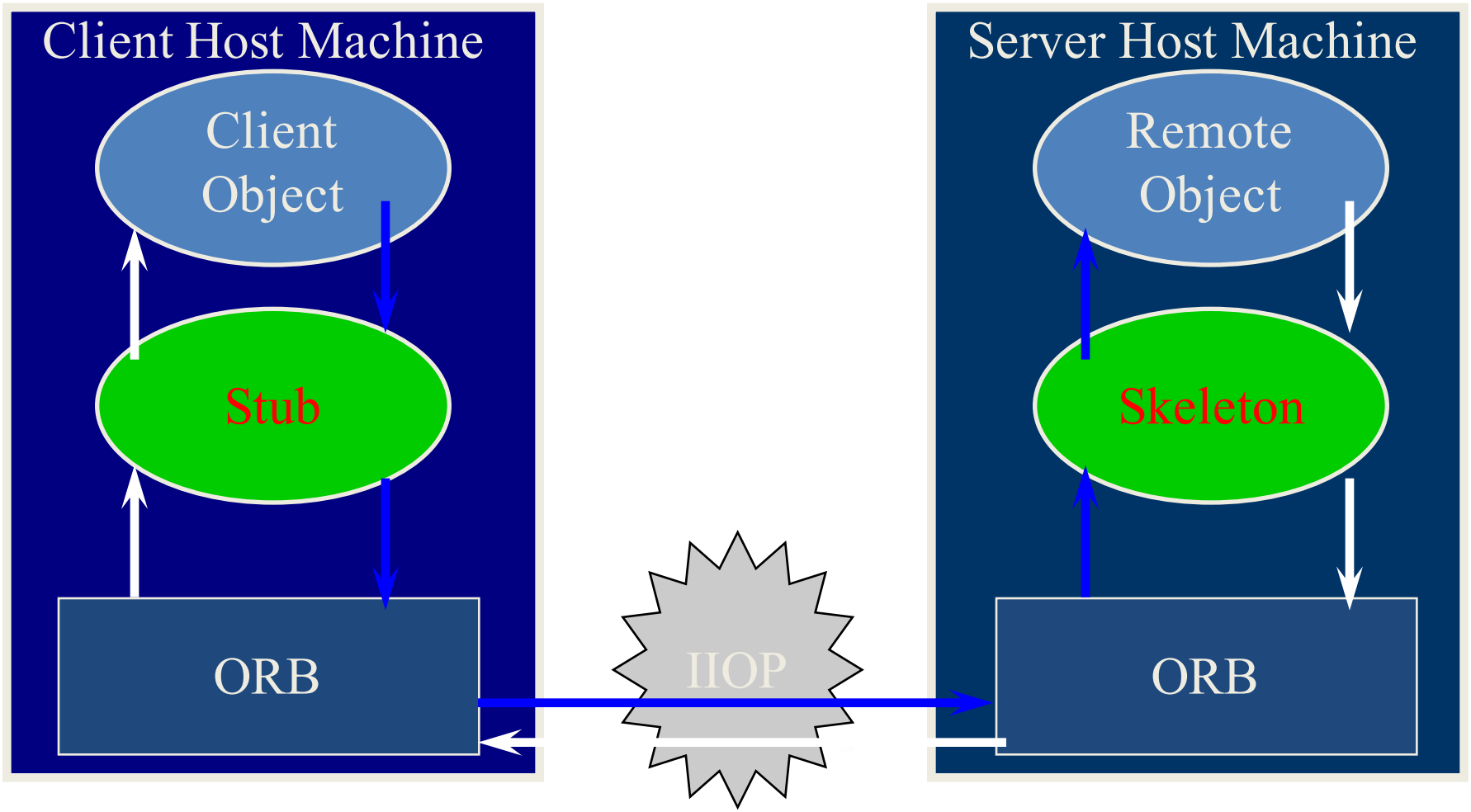
```
interface Hello {
    string sayHello();
};
```



From: http://www.molecular.com/news/recent_articles/051904_javaworld.aspx

1.3 Distributed Systems: CORBA

CORBA Architecture

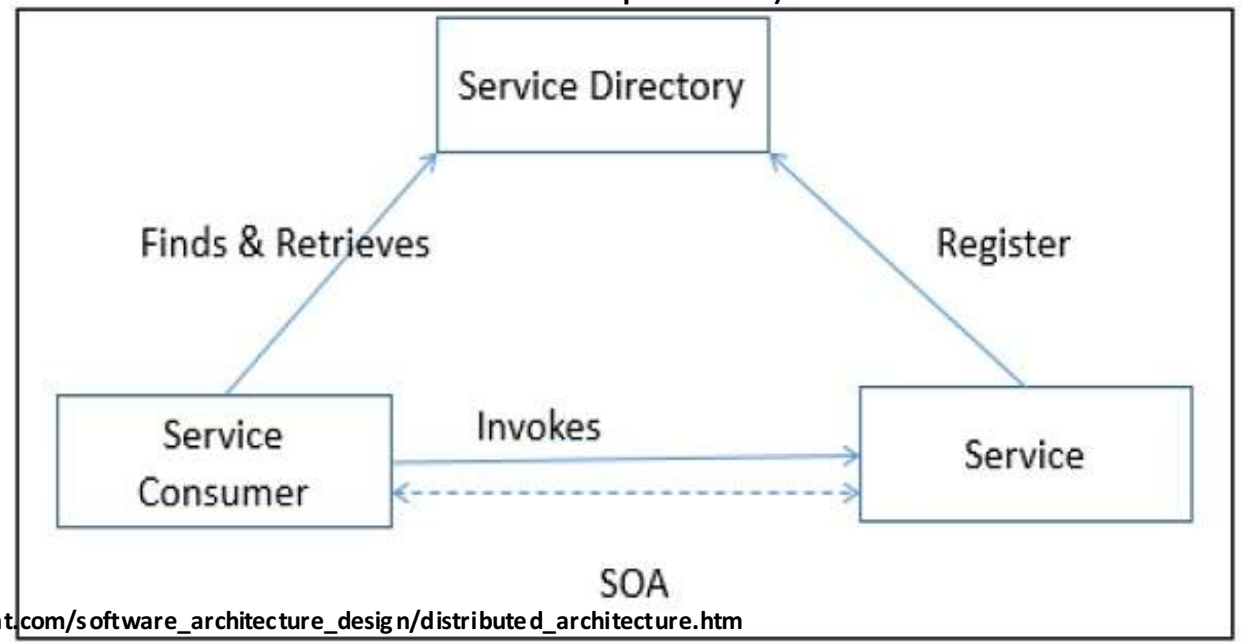


1.3 Distributed Systems: SOA

Service Oriented Architecture – SOA

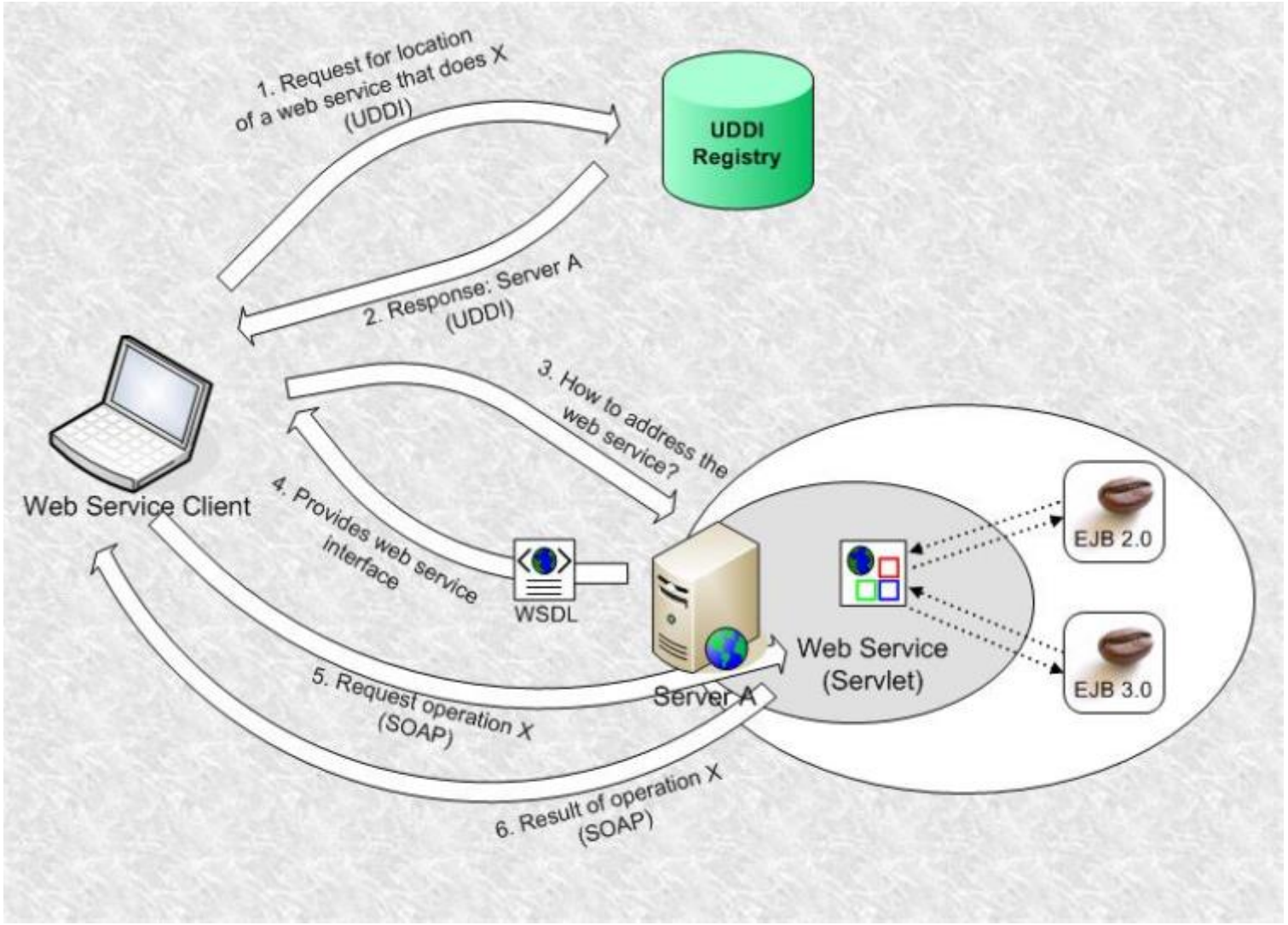
A service is a component of business functionality that is well-defined, self-contained, independent, published, and available to be used via a standard programming interface. The connections between services are conducted by common and universal message-oriented protocols such as the SOAP Web service protocol, which can deliver requests and responses between services loosely.

Service-oriented architecture is a client/server design which support business-driven IT approach in which an application consists of software services and software service consumers (also known as clients or service requesters).



1.3 Distributed Systems: SOA

Service Oriented Architecture – SOA



Service Oriented Architecture – SOA

Features of SOA

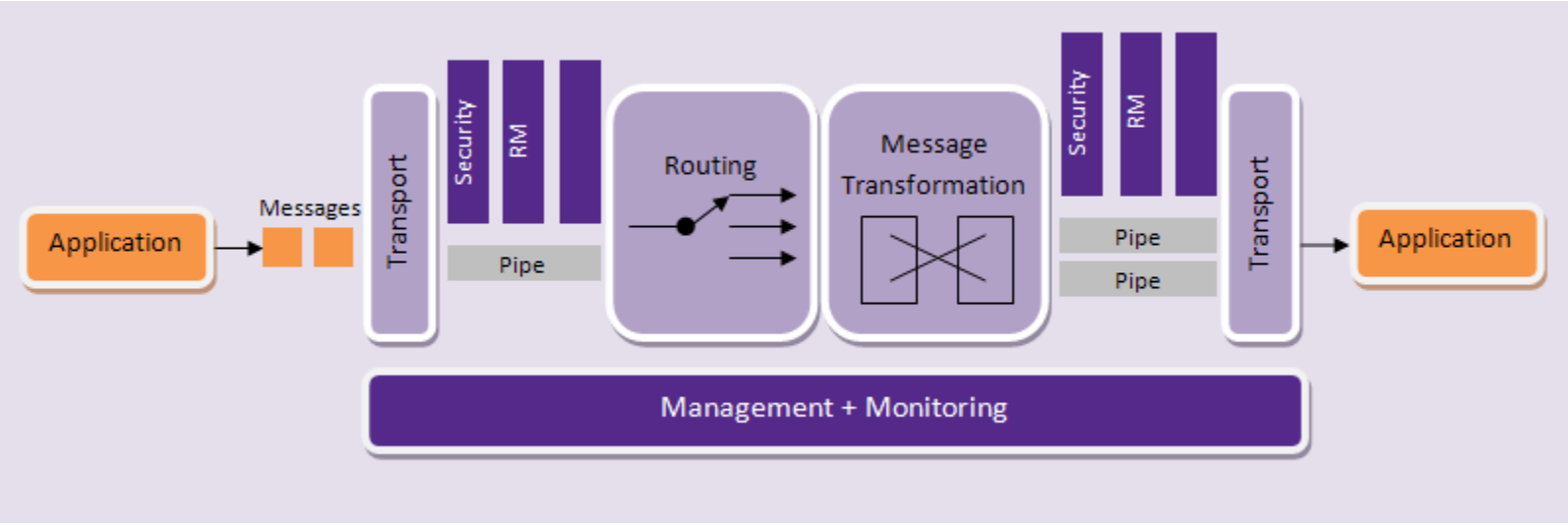
A service-oriented architecture provides the following features −

- Distributed Deployment − Expose enterprise data and business logic as loosely, coupled, discoverable, structured, standard-based, coarse-grained, stateless units of functionality called services.
- Composability − Assemble new processes from existing services that are exposed at a desired granularity through well defined, published, and standard complaint interfaces.
- Interoperability − Share capabilities and reuse shared services across a network irrespective of underlying protocols or implementation technology.
- Reusability − Choose a service provider and access to existing resources exposed as services.

1.3 Distributed Systems: ESB

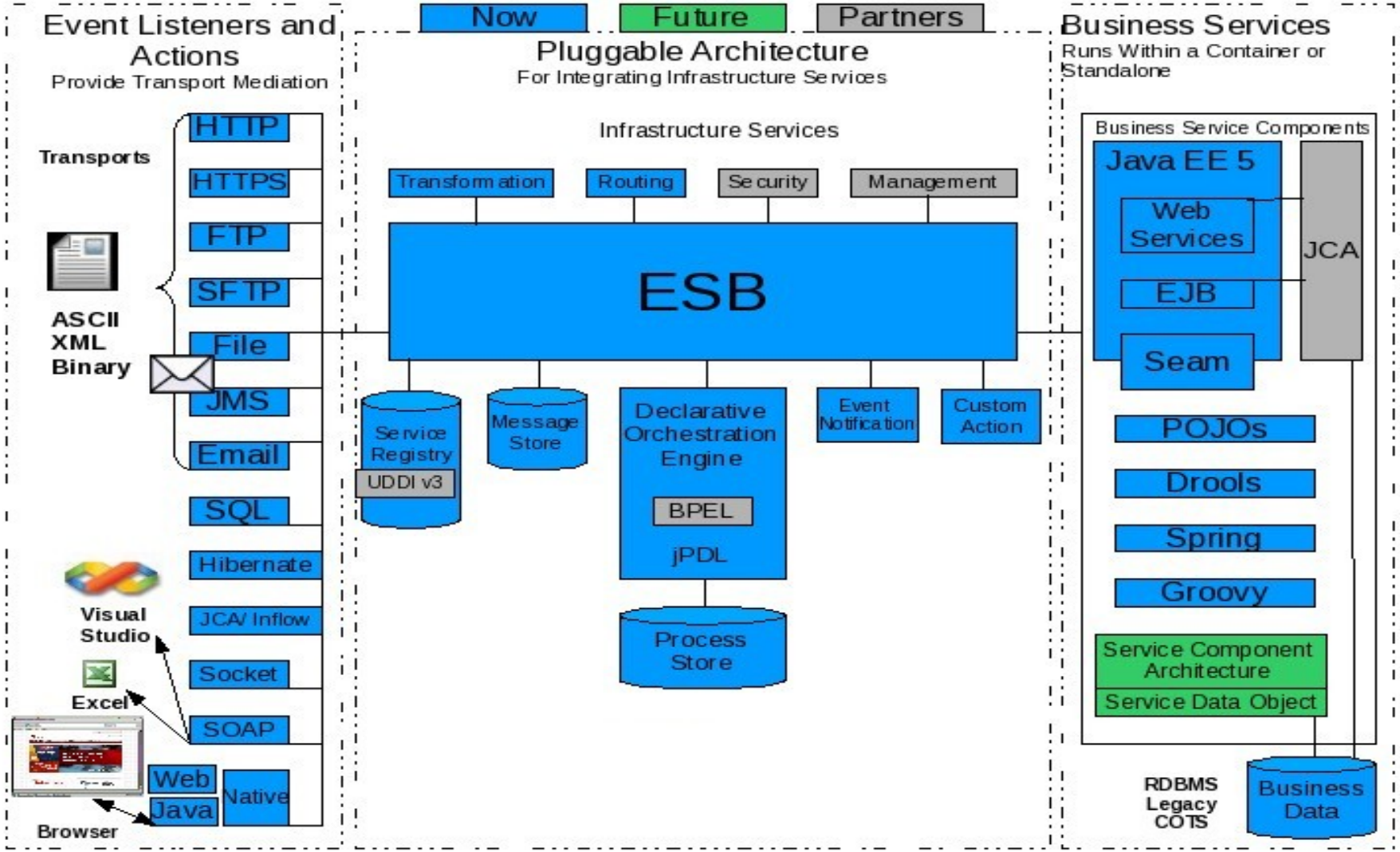
Enterprise Service Bus Architecture – ESB

An **enterprise service bus** (**ESB**) implements a communication system between mutually interacting software applications in a service-oriented architecture (SOA). As it implements a distributed computing architecture, it implements a special variant of the more general client-server model, wherein, in general, any application using ESB can behave as server or client in turns. ESB promotes agility and flexibility with regard to high-level protocol communication between applications. The primary goal of the high-level protocol communication is enterprise application integration (EAI) of heterogeneous and complex service or application landscapes (a view from the network level).



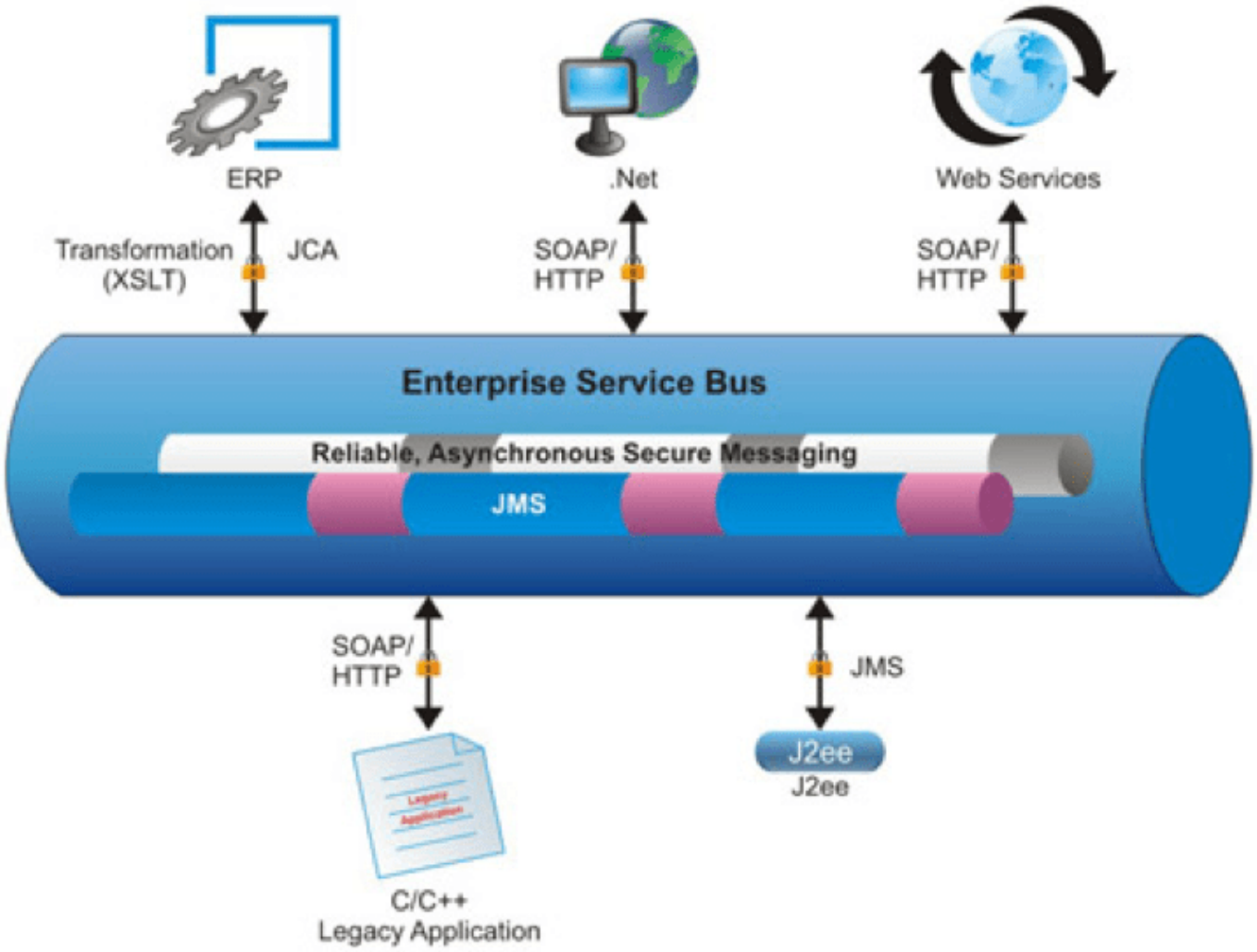
1.3 Distributed Systems: ESB

Enterprise Service Bus Architecture – ESB



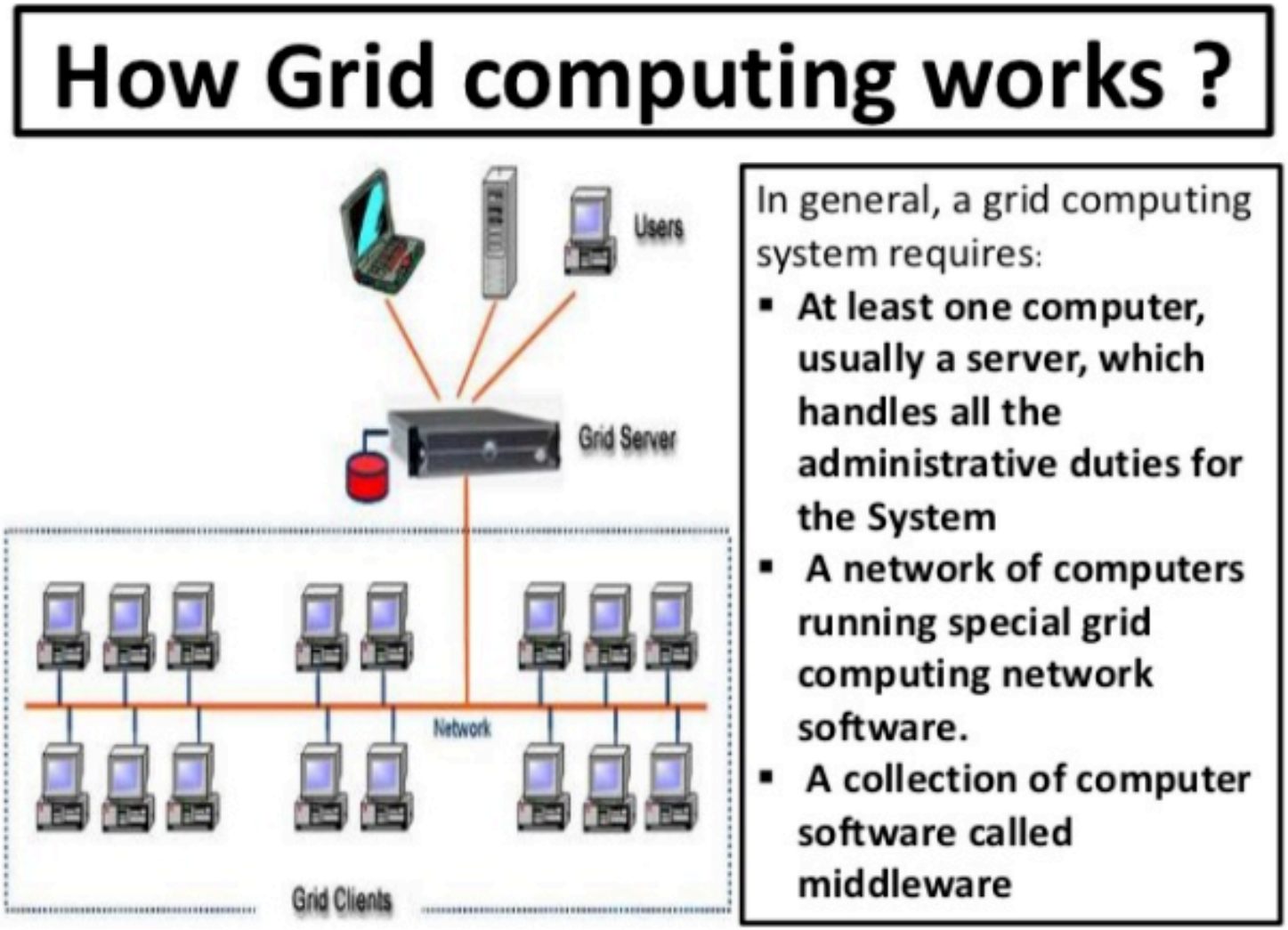
1.3 Distributed Systems: ESB

Enterprise Service Bus Architecture – ESB



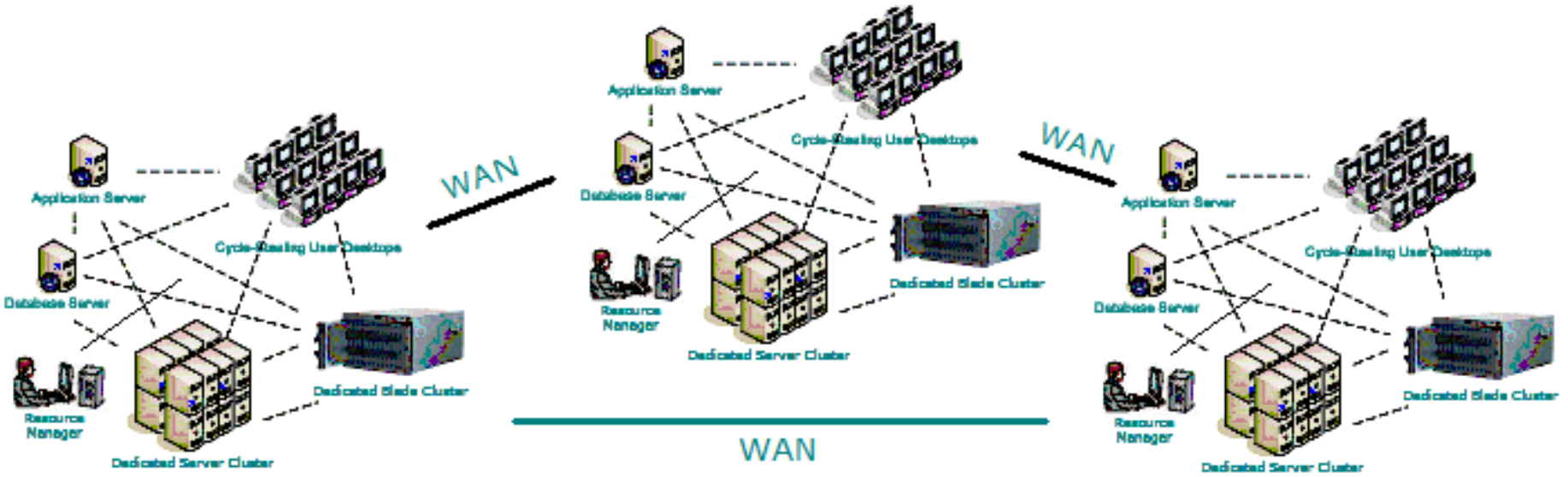
1.3 Distributed Systems: GRID

Grid Computing



1.3 Distributed Systems: GRID

Grid Computing



1.4 Distributed Systems Challenges

1. Heterogeneity – variety and difference in:
- networks
- computer hardware
- OS
- programming languages
- implementations by different developers

2. OPENNESS

- computer system - can the system be extended and re-implemented in various ways?
- distributed system - can new resource-sharing services be added and made available for use by variety of client programs?

3. Security
- Confidentiality
- Integrity
- Availability

Main goal: High availability

4. Scalability
–the ability of the system system to work well when the system load or the number of users increases

Challenges with building scalable distributed systems:
- Controlling the cost of physical resources
- Controlling the performance loss
- Preventing software resources running out (like 32-bit internet addresses, which are being replaced by 128 bits)
- Avoiding performance bottlenecks

5. Failure handling
Techniques for dealing with failures
- Detecting failures
- Masking failures

- messages can be retransmitted
- disks can be replicated in a synchronous action
- Tolerating failures
- Recovery from failures
- Redundancy – redundant components

- at least two different routes
- database can be replicated in several servers

1.4 Distributed Systems Challenges

6. Concurrency – Several clients trying to access shared resource at the same time.

Any object with shared resources in a DS must be responsible that it operates correctly in a concurrent environment.

7. Transparency

Transparency – concealment from the user and the applications programmer of the separation of components in a Distributed System for the system to be perceived as a whole rather than a collection of independent components:

- Access transparency – access to local and remote re- sources identical
- Location transparency – resources accessed without knowing their physical or network location
- Concurrency transparency – concurrent operation of processes using shared resources without interference be- tween them
- Replication transparency – multiple instances seem like one
- Failure transparency – fault concealment
- Mobility transparency – movement of resources/clients within a system without affecting the operation of users or programs

8. Quality of service

Main nonfunctional properties of systems that affect *Quality of Service (QoS)*:

- reliability
- security
- performance

*** Distributed Systems**

- ***Data Migration***
- ***Calculus / Business Logic Migration***
- Process Migration !?

Distributed Systems Challenges

In Distributed Systems & Cloud the core things are about:

- ***Data Migration***
- ***Calculus / Business Logic Migration***
- Process Migration !?

Cloud, Containers & Micro-services are parts of any Distributed Computing Systems

Section Conclusion

Fact: **DAD needs Java and ECMAScript/JS/node.js**

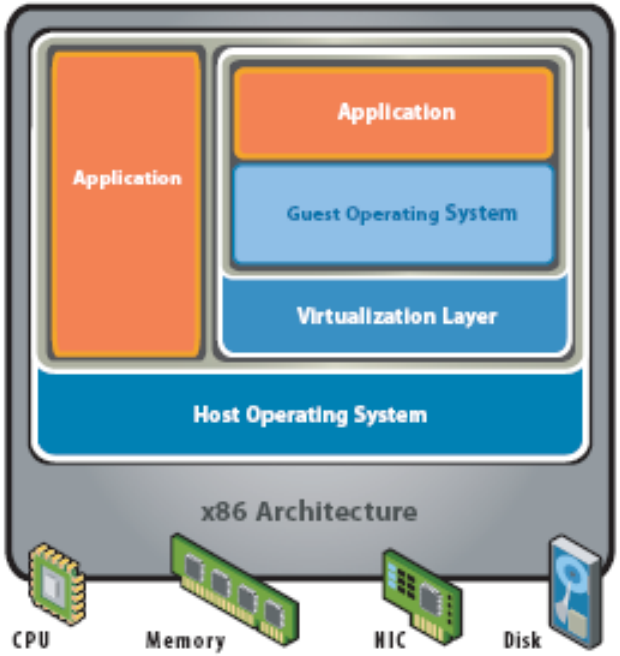
In few **samples** it is simple to remember: Types of the distributed architectural patterns of the distributed systems.

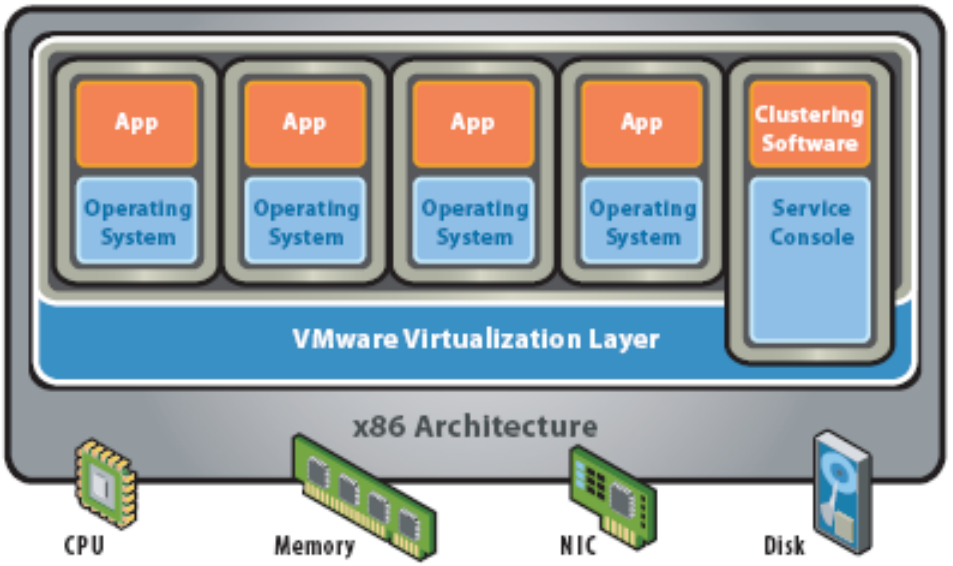
Computing Clouds: IaaS, “Native Clouds”: PaaS, CaaS, FaaS | (SaaS), Dev-Ops & INFRA: Jenkins - Bamboo

Computing Cloud, Dev-Ops & INFRA

Distributed Systems: Cloud Architecture

Cloud Concepts – Intro – Virtualization Overview





Hosted Architecture

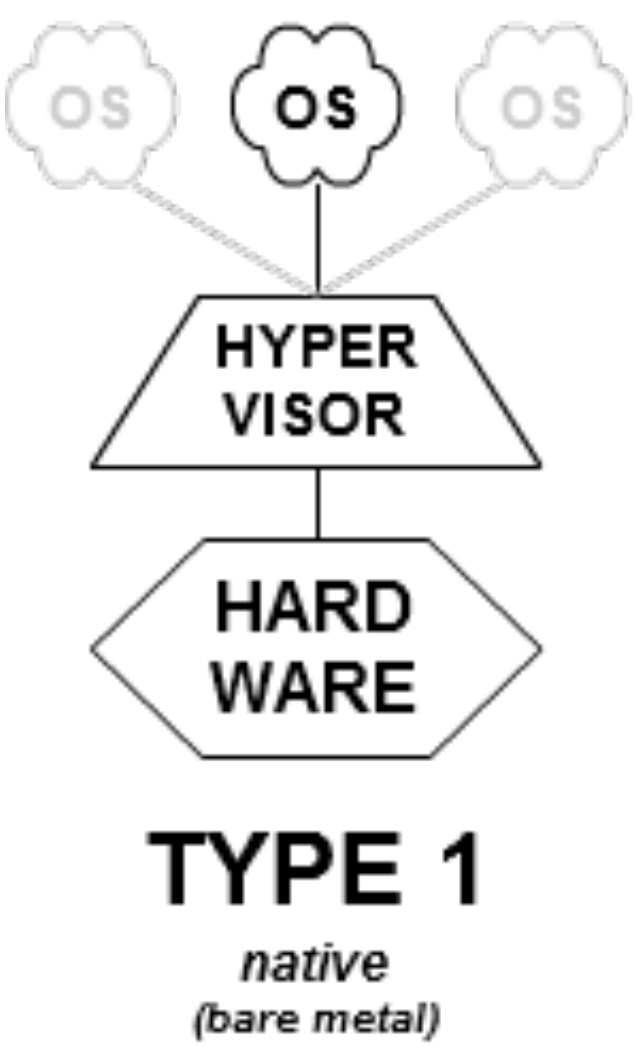
- Installs and runs as an application
- Relies on host OS for device support and physical resource management

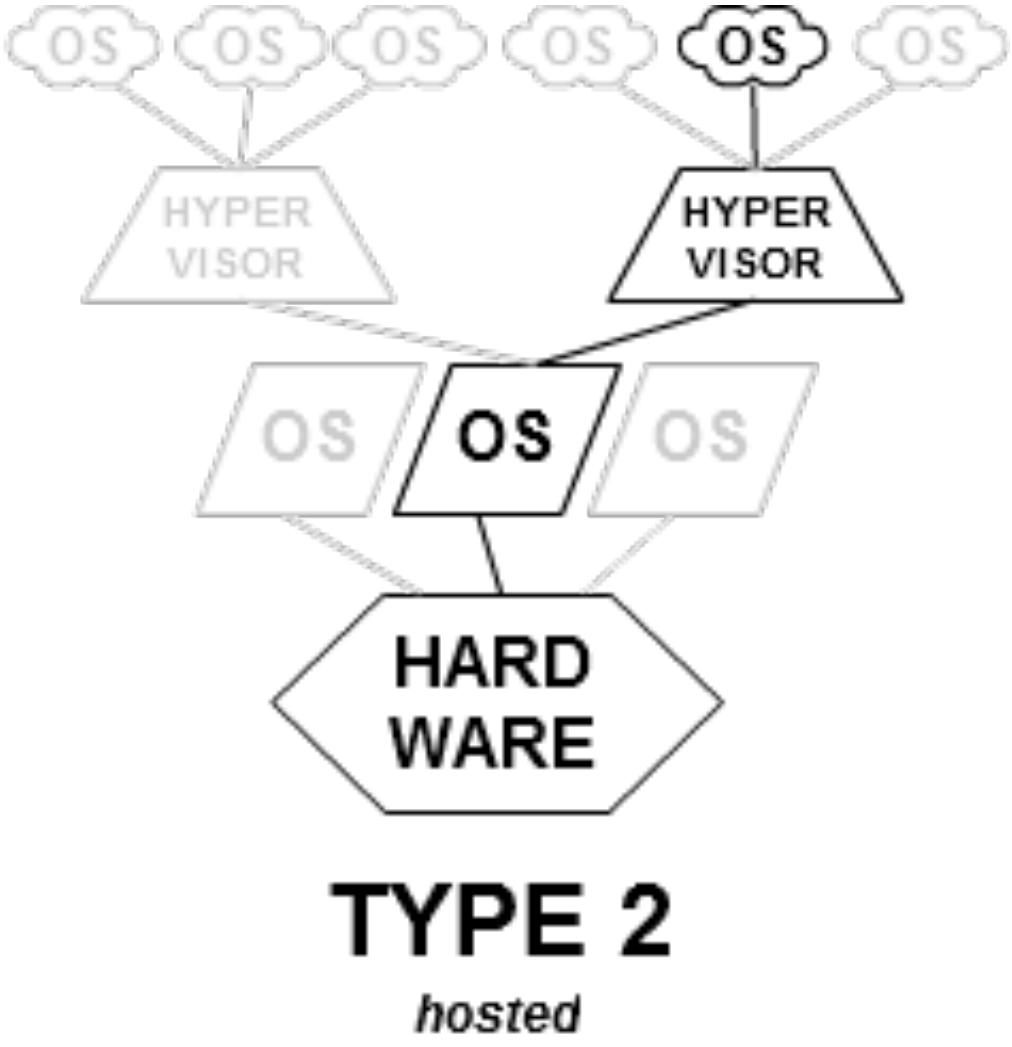
Bare-Metal (Hypervisor) Architecture

- Lean virtualization-centric kernel
- Service Console for agents and helper applications

Figure 2: Virtualization Architectures

Cloud Concepts – Intro – Virtualization Overview

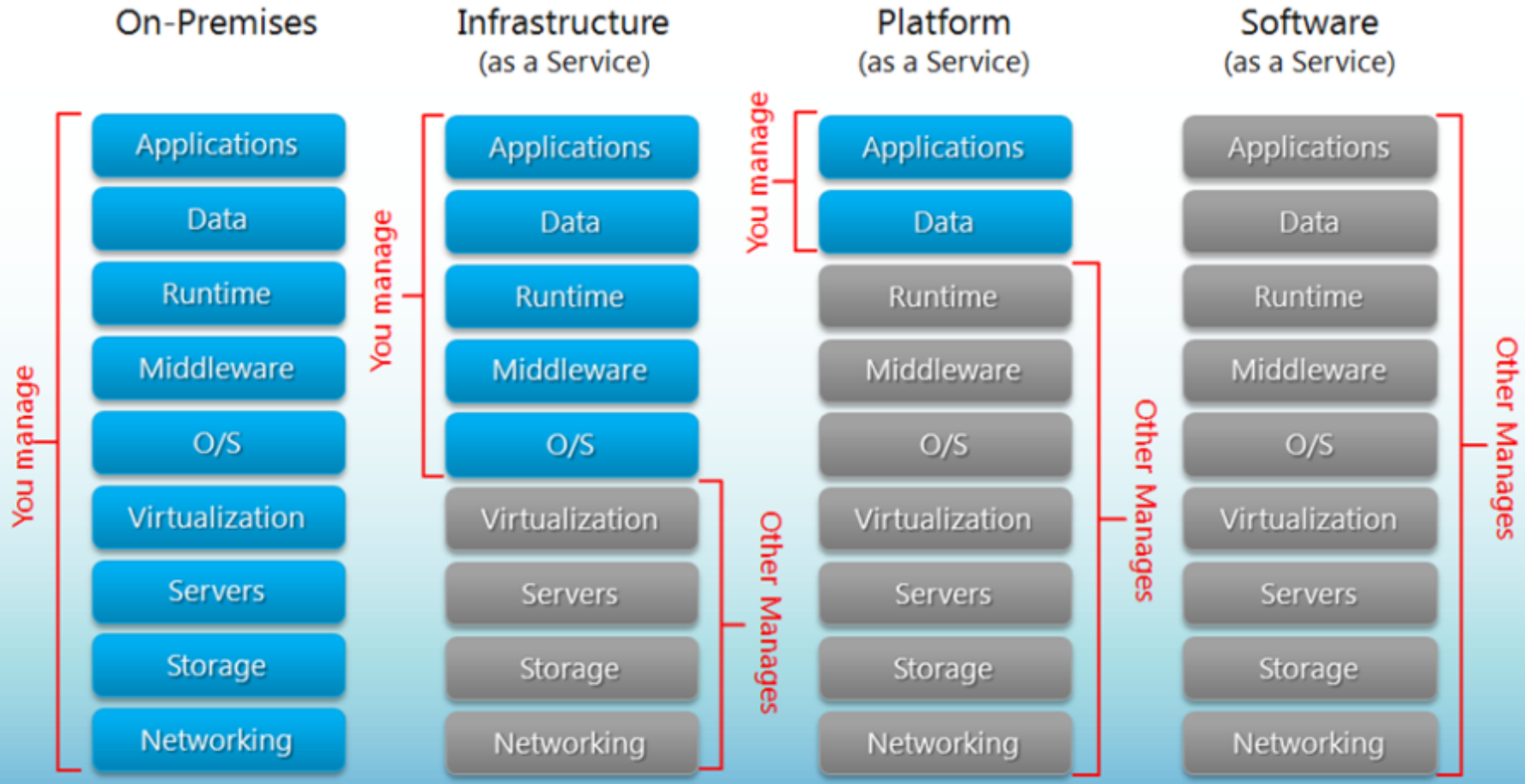




1.3 Distributed Systems: Cloud

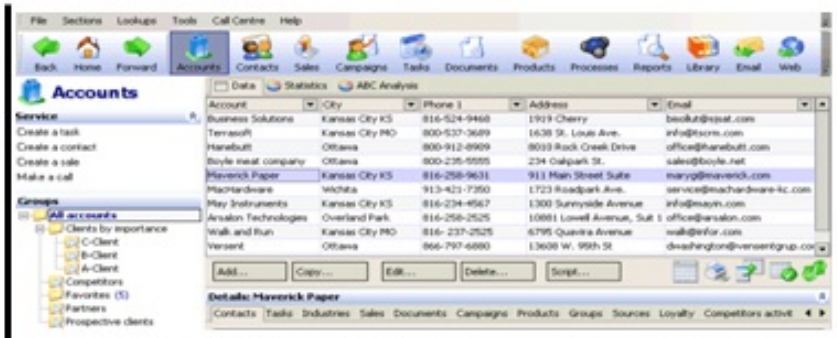
Separation of Responsibilities

Cloud Concepts – Intro – IaaS, PaaS, SaaS



Cloud Concepts – Intro – IaaS, PaaS, SaaS

SaaS



SalesForce.com, Google Apps

PaaS

Tomcat App Server

Java VM

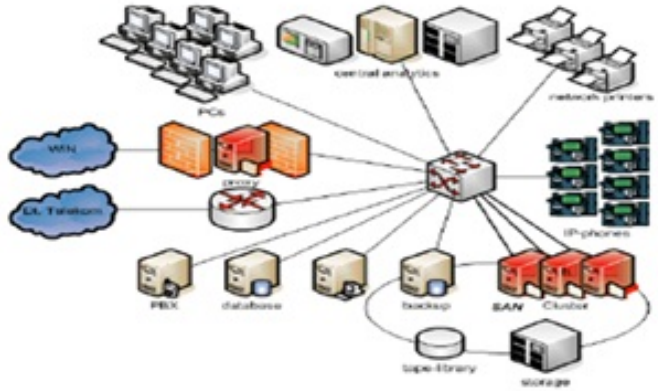
MySQL DB

Ubuntu Linux

Google App Engine for: Java, Ruby, Python & GO

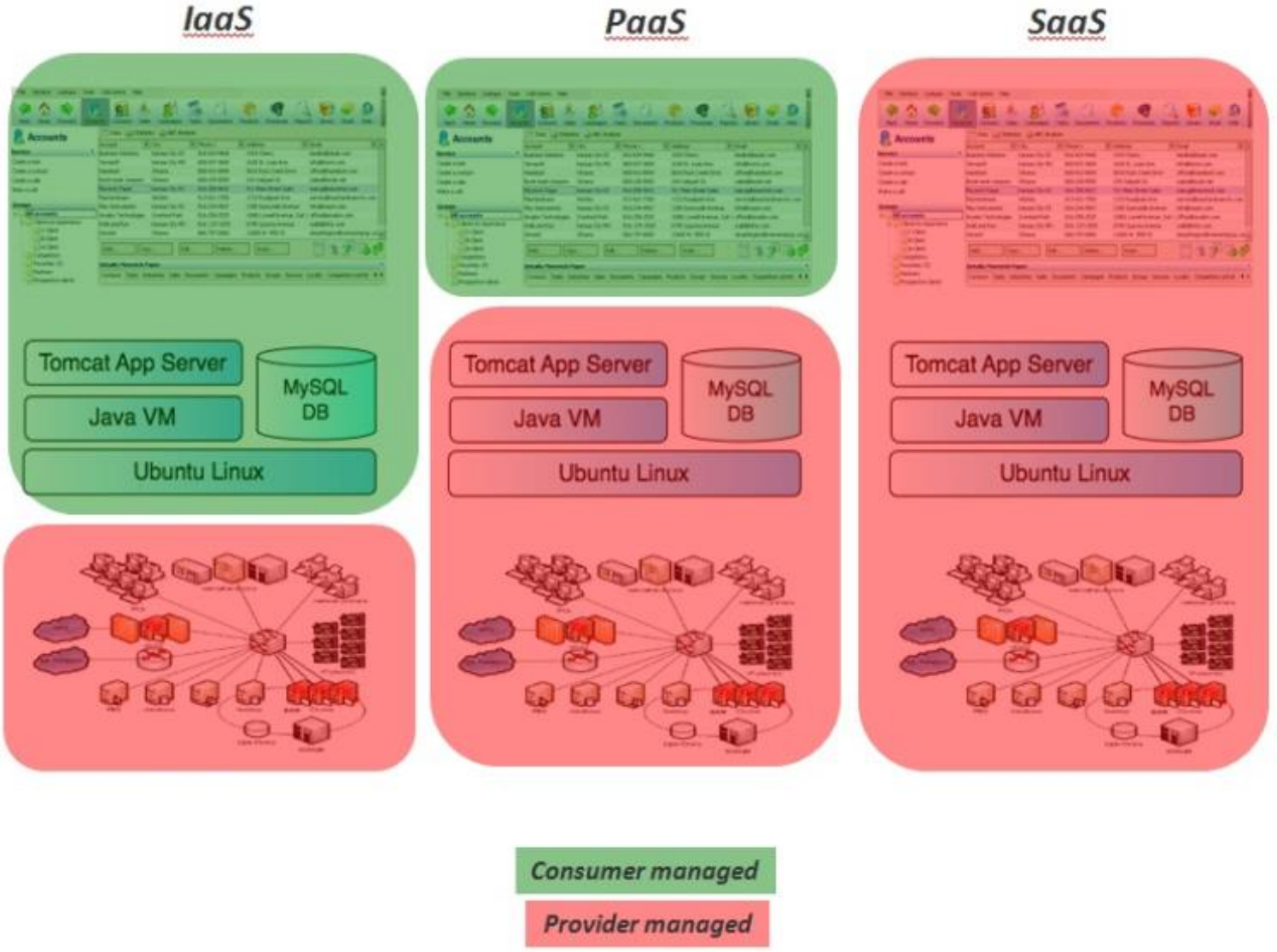
VMForce.com, MS Azure

IaaS

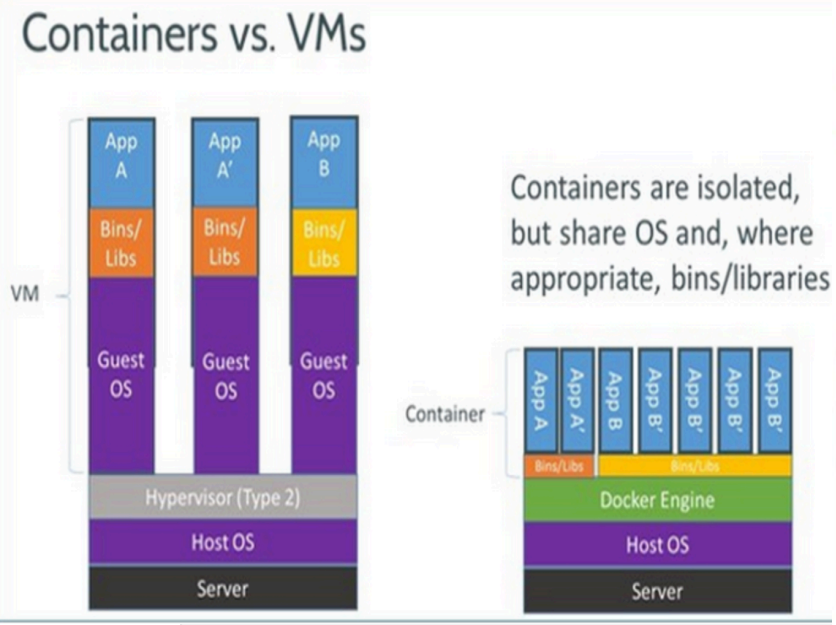


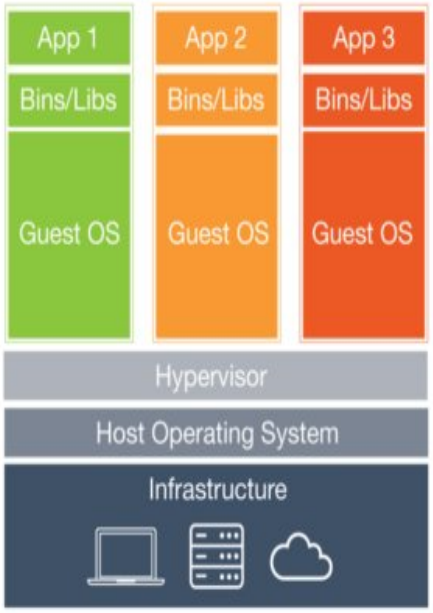
vCloud Express/Datacenter, Amazon EC2

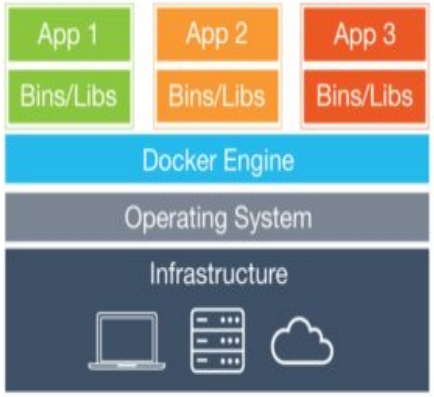
Cloud Concepts – Intro – IaaS, PaaS, SaaS

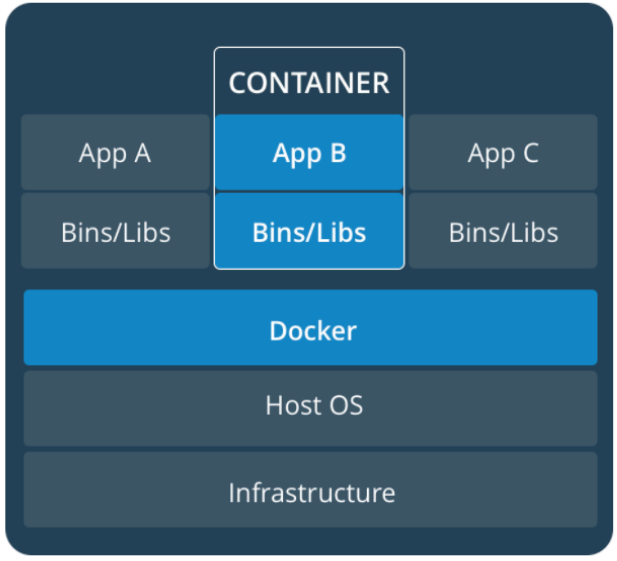


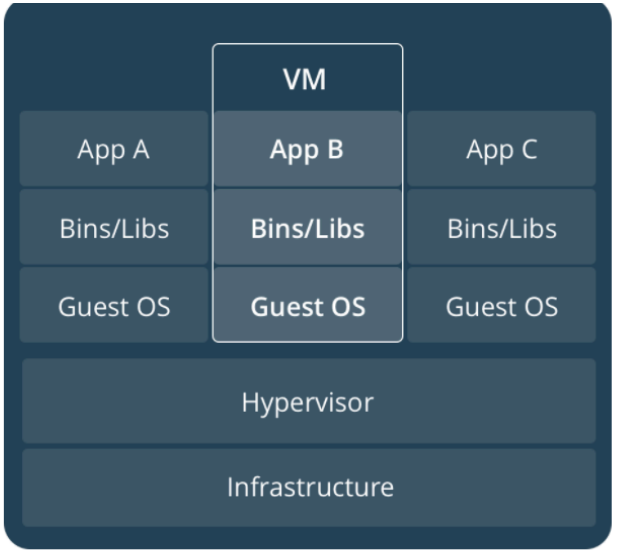
Deployment: Std. OS vs. VMs vs. Docker Containers



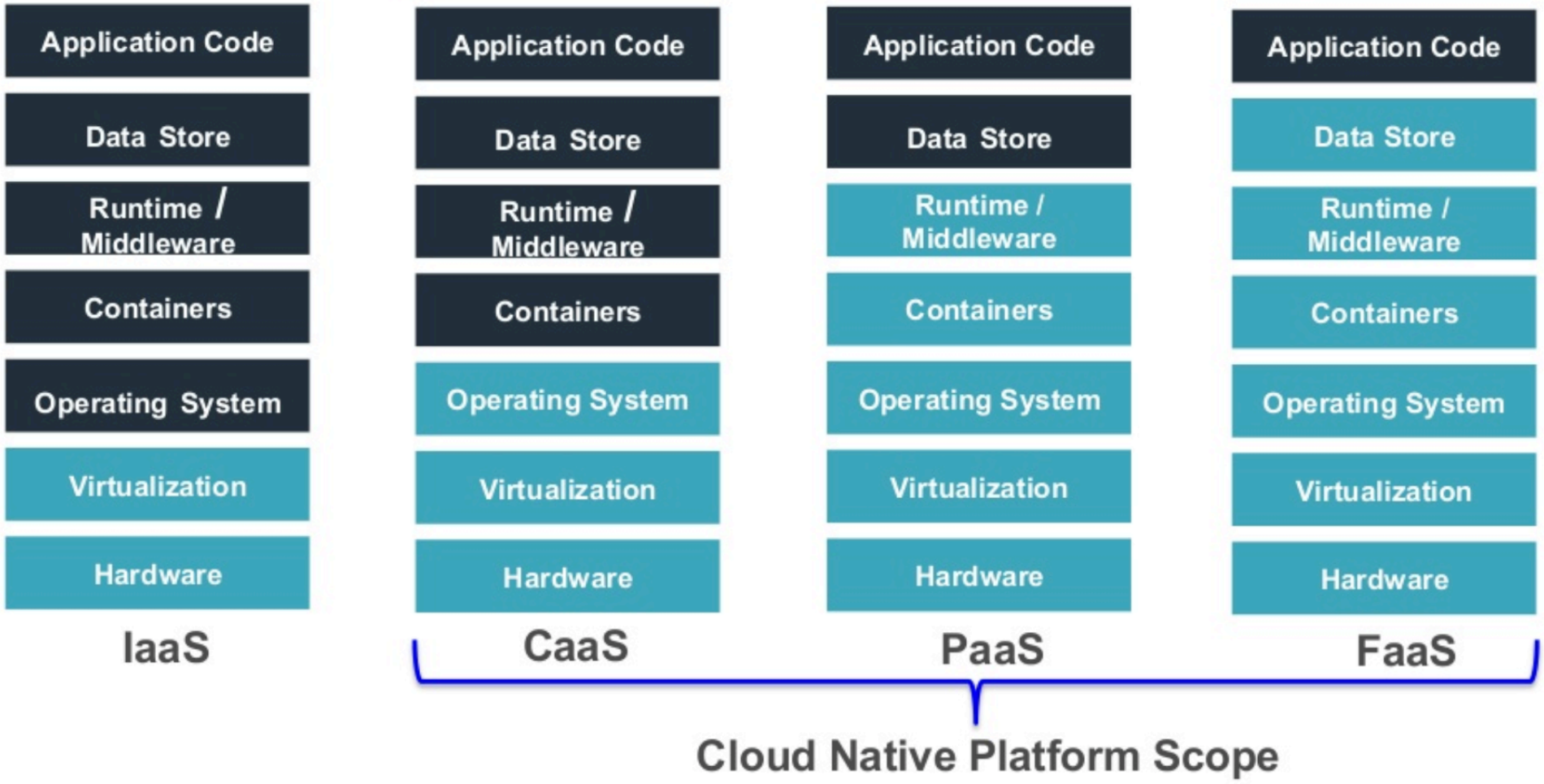






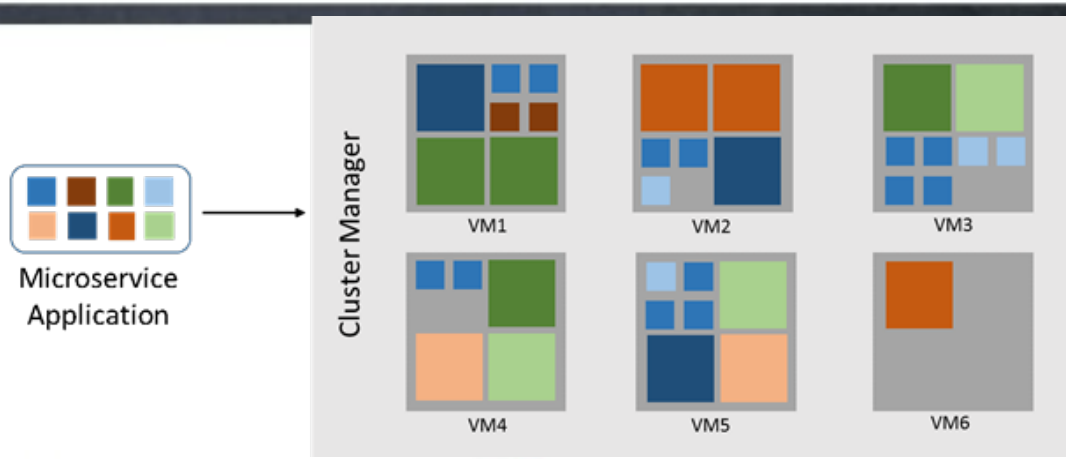


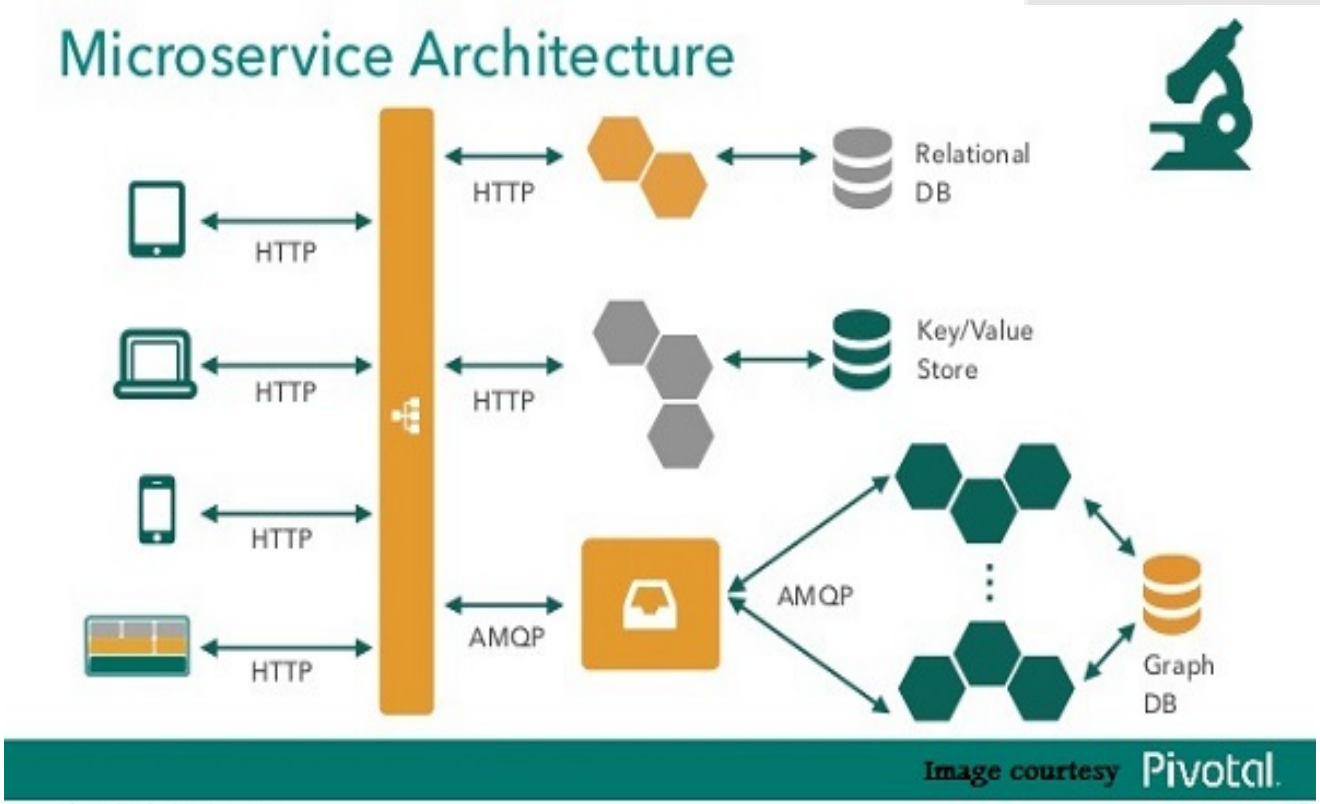
Cloud Concepts – Intro – IaaS, PaaS, CaaS, SaaS



Distributed Systems: Micro-services

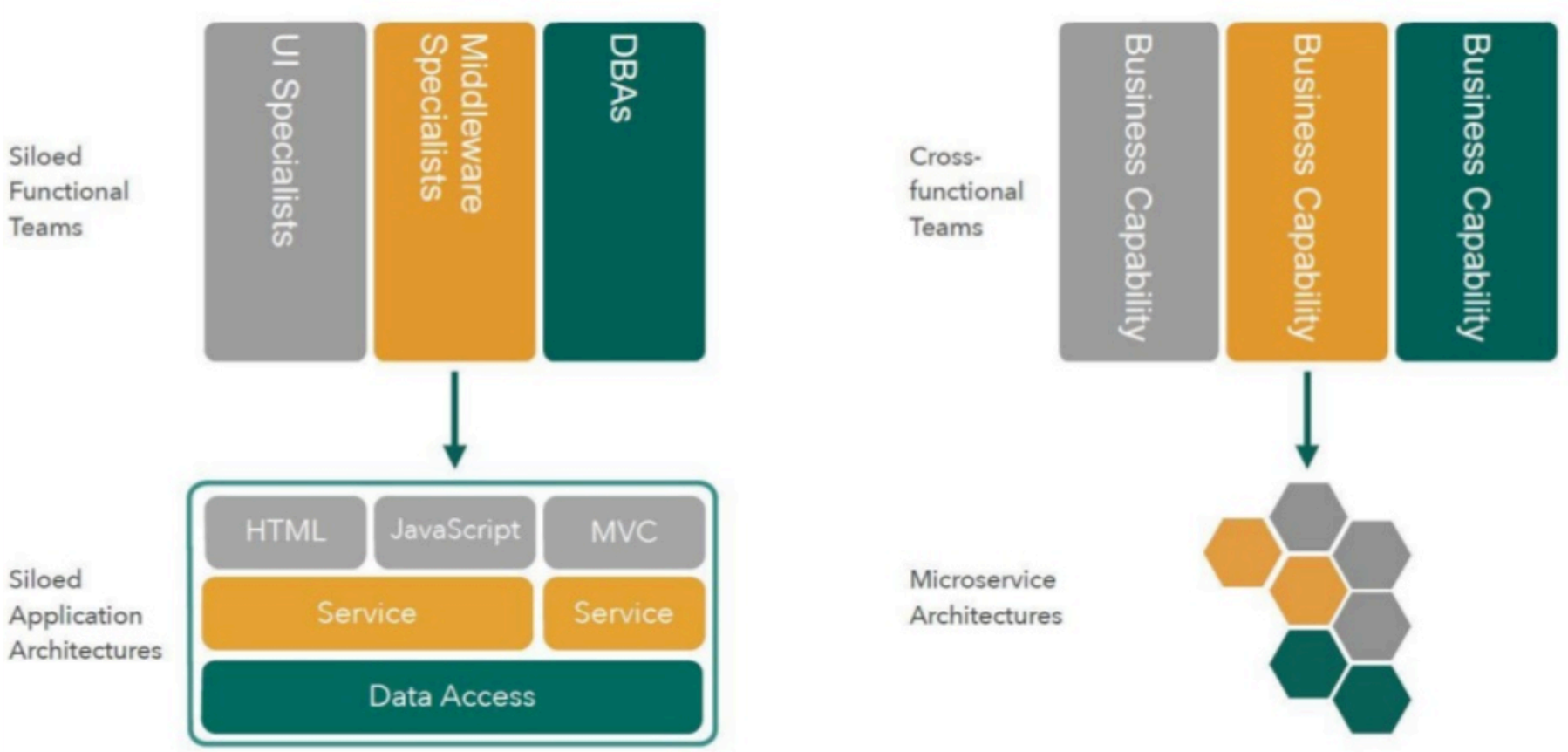
Micro-services Architecture





Distributed Systems

Micro-services Architecture



Section Conclusions

Computing Clouds

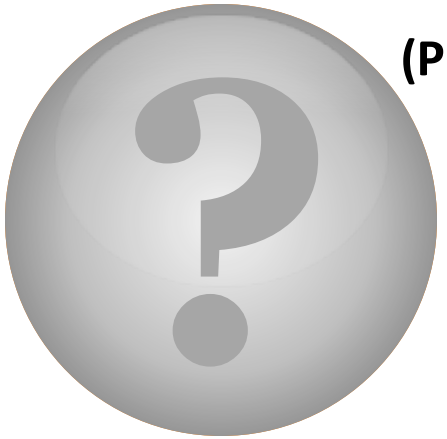
Computing Clouds
for easy sharing

Share knowledge, Empowering Minds

Communicate & Exchange Ideas

Some “myths”:

(Distributed Systems).Equals(Distributed Computing) == true?



(Parallel System).Equals(Parallel Computing) == true?

(Parallel System == Distributed System) != true?

(Sequential vs. Parallel vs. Concurrent vs. Distributed *Programming*) *? (Different) : (Same)*

Questions & Answers!

But wait…

There’s More!

```
if (HTC !=  HPC)
    HTC  (High Throughput Computing) >
    MTC (Many Task  Computing) >
    HPC (High Performance Computing);
```

… Will be continued! - In the next lectures …

Thanks!