SOFTWARE DESIGN & ARCHITECTURE (LECTURE-4)

USAMA MUSHARAF

LECTURER (Department of Computer Science)

FAST-NUCES PESHAWAR

CONTENTS

Intro to Software Architecture

- Conceptual Model of Architecture Representation
- Architectural Views
- Views and View Point
- 4+1 View Model
- Discussion on Uber Case Study (System Design)

CATEGORIES OF ARCHITECTURAL STYLES

- **Hierarchical Software Architecture**
  - Layered
- **Data Flow Software Architecture**
  - Pipe and Filter
  - Batch Sequential
- **Data Centered Software Architecture**
  - Black board
  - Shared Repository
- **Component-Based Software Architecture**
- **Distributed Software Architecture**
  - Client Server
  - Peer to Peer
  - REST
  - SOA
  - Microservices
  - Cloud Architecture
- **Event Based Software Architecture**

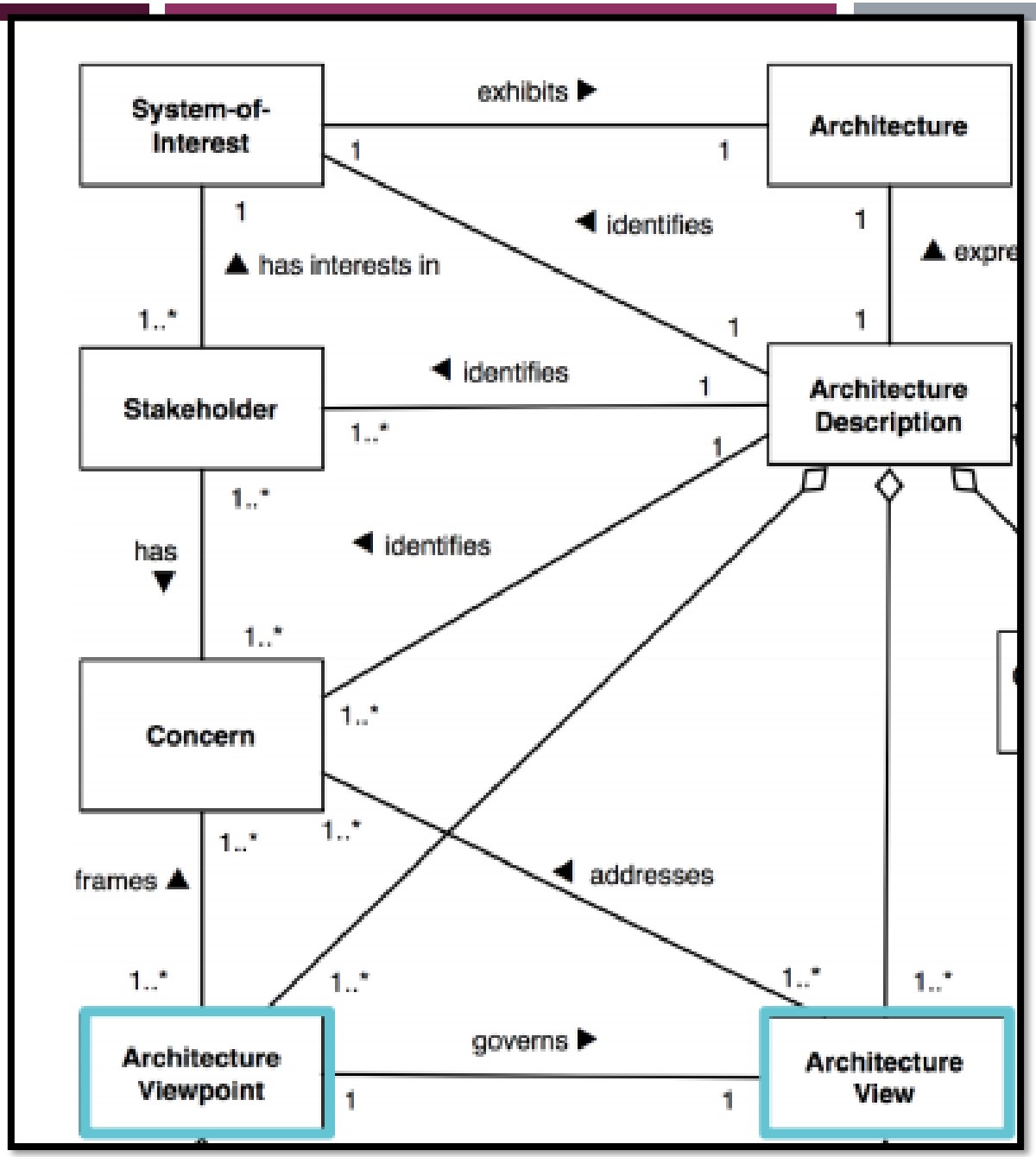
SOFTWARE ARCHITECTURE

Software Architecture

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.

Conceptual Model of an Architecture Description

ISO/IEC/IEEE 42010


System-of-Interest
exhibits
Architecture
1
1
1
identifies
1
has interests in
1
1..*
1
Stakeholder
identifies
1
Architecture Description
1..*
1
1..*
identifies
has
1..*
Concern
1..*
1..*
1..*
frames
addresses
1..*
1..*
1..*
1..*
Architecture Viewpoint
governs
Architecture View
1
1

INTRODUCTION TO VIEWS

Dictionary Meaning

Manner of looking at something

Why (multiple) view ?

For better understanding and managing.

Multi dimensional view must be taken for any complex entity because of its complex nature ,

It can’t be described in 1 dimensional view.

INTRODUCTION TO VIEWS

For example, In civil what are the views of a building…

- *Room layout*
- *3D view of building / room*
- *Electrical diagram*
- *Plumbing diagram*
- *Security alarm diagram*
- *AC duct diagram etc…etc…*

- Which of the above view is Architecture?
- **In Software, What are views ? ……..**

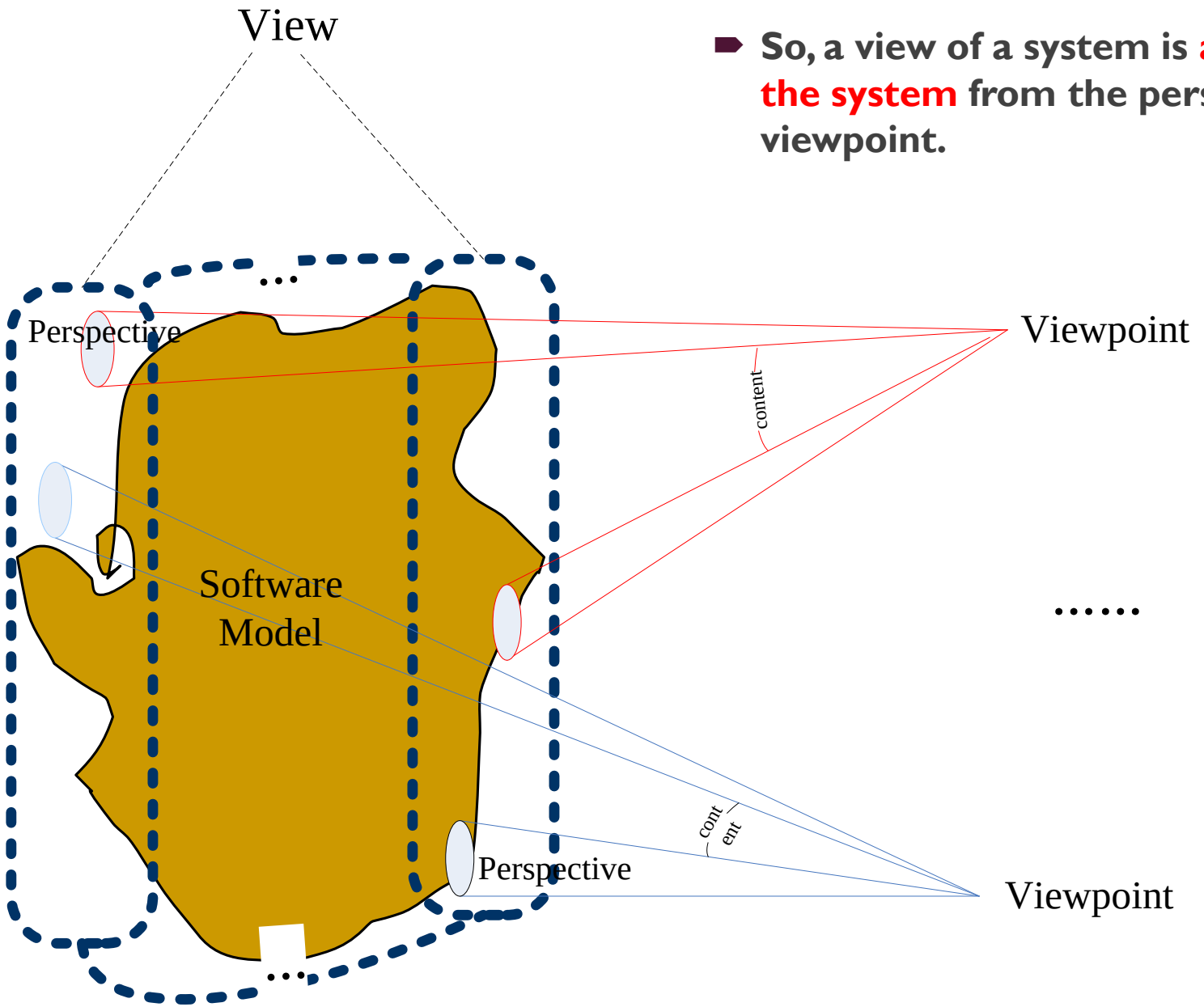
DEFINITION OF SW VIEW

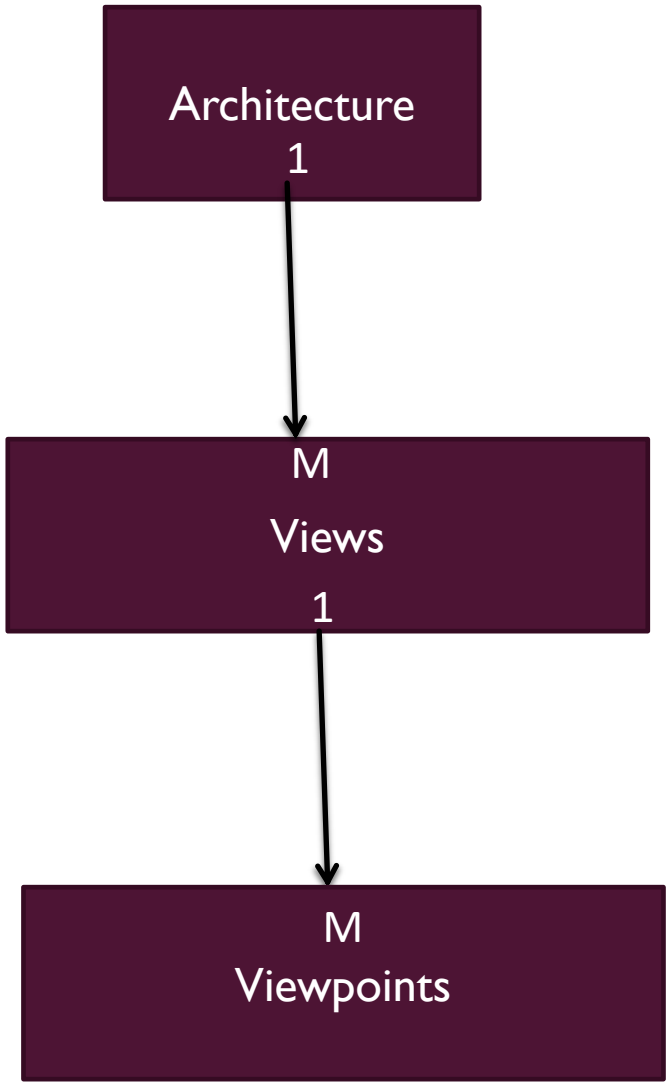
As per IEEE definition,

- Software architecture descriptions are commonly organized into views,
- Each view addresses a set of system concerns, following the conventions of its *viewpoint*.

  - Viewpoint - A position or direction from which something is observed or considered;
  - View – Details or full specification considered from that viewpoint

- So, a view of a system is **a representation of the system** from the perspective of a viewpoint.




Architecture
1
M
Views
1
M
Viewpoints

VIEW MODEL

Software designers can organize the description of their architecture decisions in different views.

4+1 VIEW MODEL

The 4+1 view is an architecture verification technique for studying and documenting software architecture design.

THE 4 +1 VIEW MODEL

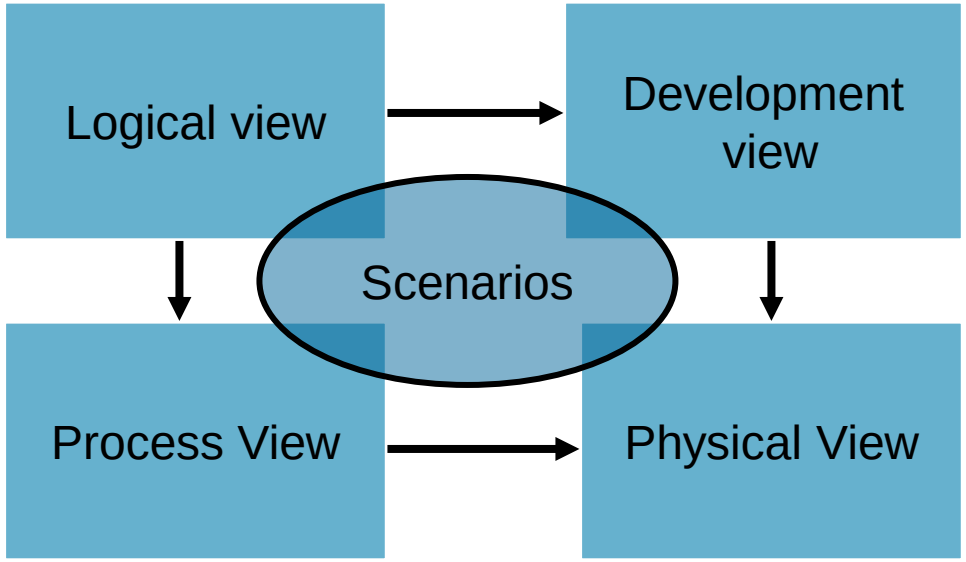
The 4+1 view model was originally introduced by Philippe Kruchten (Kruchten, 1995).

The model provides four essential views:

- the logical view,
- the process view,
- the physical view,
- the development view

- and fifth is the scenario view

4+1 VIEW MODEL OF ARCHITECTURE



THE 4+1 VIEW MODEL

- Multiple-view model that addresses different aspects and concerns of the system.

- Standardizes the software design documents and makes the design easy to understand by all stakeholders.

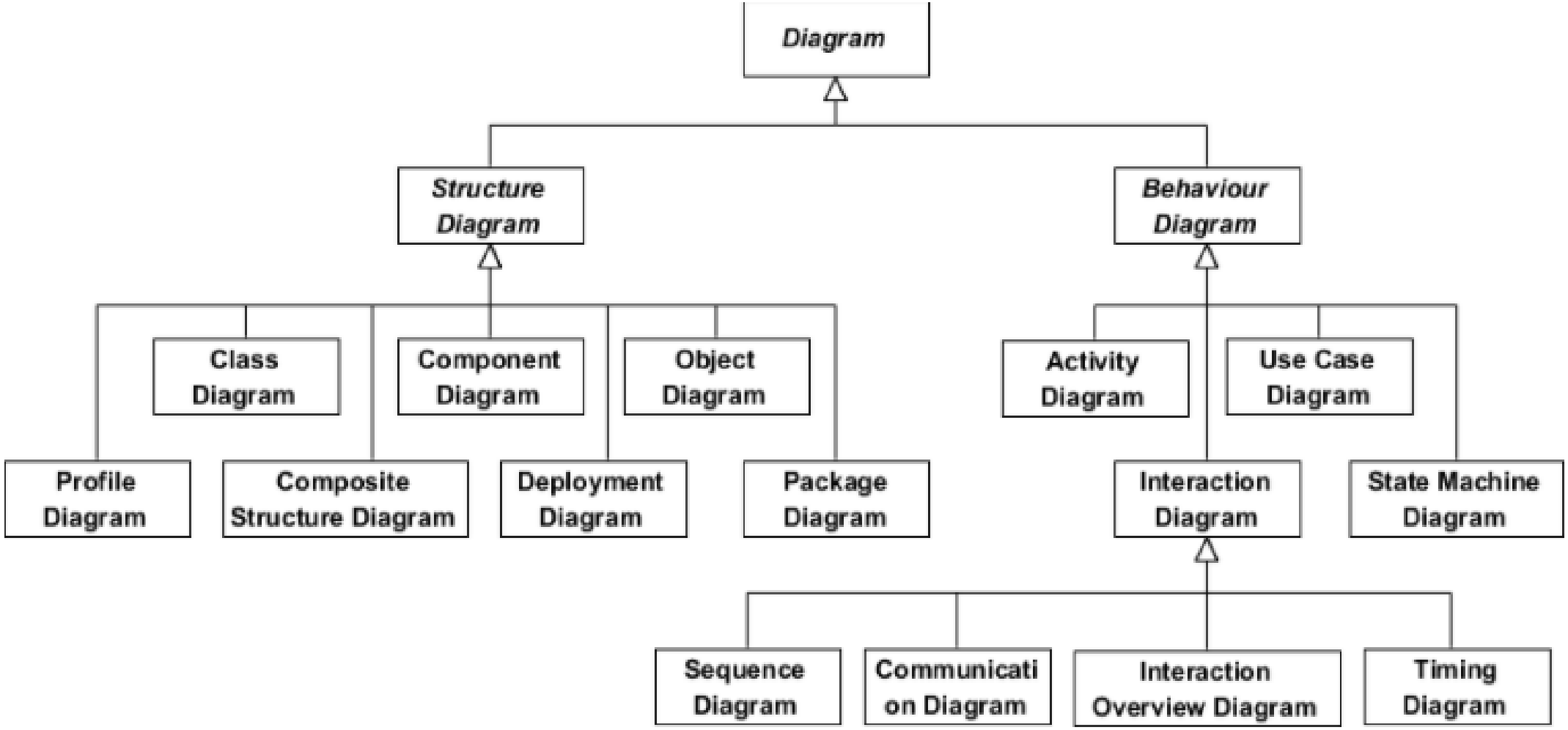


Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communicati on Diagram
Interaction Overview Diagram
Timing Diagram

THE SCENARIO VIEW- USE CASE VIEW

- The scenario view describes the functionality of the system, i.e., how the user employs the system and how the system provides services to the users.

- It helps designers to discover architecture elements during the design process and to validate the architecture design afterward.

USE CASE

- They use case view illustrates the functionality of the system.
- Using use case we can capture the goals of the user or what the user expects from the system.
- In UML, Use Cases can be created through use case diagrams or use case descriptions
- Use cases can be created by analysts' architects or even by the users.

THE LOGICAL OR CONCEPTUAL VIEW

- The logical view is based on application domain entities necessary to implement the functional requirements.

- The logical view specifies system decomposition into conceptual entities (such as objects) and connections between them (such as associations).

LOGICAL VIEW

- The logical view shows the parts that make up the system and how they interact with each other.
- It represents the abstractions that are used in the problem domain
  - These abstractions are classes and objects
- Different UML diagrams show the logical way such as class diagram state diagram sequence, diagram communication diagram and object diagram.

Logical view

- class diagram
- state diagram
- Sequence diagram
- communication diagram
- object diagram

THE DEVELOPMENT OR MODULE VIEW

- The development view derives from the logical view and describes the static organization of the system modules.

- UML diagrams such as package diagrams and component diagrams are often used to support this view.

DEVELOPMENT VIEW

- The development view describes the modules are the components of the system.
- This might include packages or libraries.
- It gives a high-level view of the architecture of the system and helps in managing the layers of the system.
- UML provides two diagrams for development view.
  - component Diagram
  - package Diagrams

Development View

- Component Diagram
- Packages Diagram

THE PROCESS VIEW

- The process view focuses on the dynamic aspects of the system, i.e., its execution time behavior.
- This view maps functions, activities, and interactions onto runtime implementation.

PROCESS VIEW

- Then we have the process view
- Through this view, we can describe the processes of the system and how they communicate with each other using process
- Using process view, we can find out what needs to happen to the system
- So using process view we can understand the overall functioning of the system
- Activity diagram in UML represents the process view

Process view

- Activity Diagram

THE PHYSICAL VIEW

- The physical view describes installation, configuration, and deployment of the software application.
- It concerns itself with how to deliver the deploy-able system.
- The physical view shows the mapping of software onto hardware.

PHYSICAL VIEW

- The physical view is the view that models the execution environment of the system
- Using this view, we can model the software entities onto the hardware that will host and run the entities
- The physical view in UML is represented through deployment diagrams

Physical view

- Deployment Diagrams

DATA CENTERED / SHARED DATA SOFTWARE ARCHITECTURE

SHARED DATA SOFTWARE ARCHITECTURE

- Data-centered software architecture is characterized by a centralized data store that is shared by all surrounding software components.

- The software system is decomposed into two major partitions: data store and independent software component or agents.

SHARED DATA SOFTWARE ARCHITECTURE

- In pure data-centered software architecture, the software components don't communicate with each other directly; instead, all the communication is conducted via the data store.

- The shared data module provides all mechanisms for software components to access it, such as insertion, deletion, update, and retrieval.

SHARED DATA:

- Blackboard style
- Repository style

REPOSITORY ARCHITECTURE

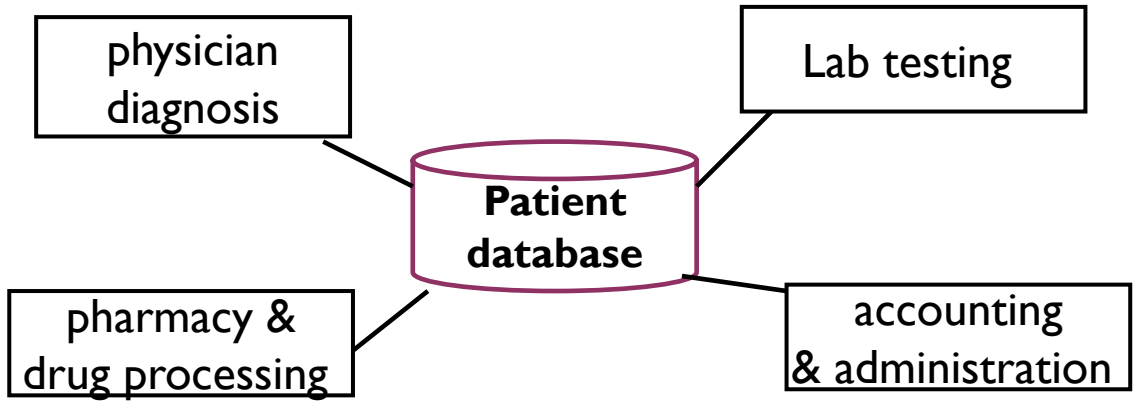
REPOSITORY ARCHITECTURE

- The repository architecture style is a data-centered architecture that supports user interaction for data processing.

- The software component agents of the data store control the computation and flow of logic of the system.

REPOSITORY ARCHITECTURE

- All data in a system is managed in a central repository that is accessible to all system components.
- Components do not interact directly, only through the repository.



REPOSITORY ARCHITECTURE

- Organizing tools around a repository is an efficient way to share large amounts of data.
  - There is no need to transmit data explicitly from one component to another.
- Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.
  - In practice, it may be difficult to distribute the repository over a number of machines.

EXAMPLE: INTEGRATED DEVELOPMENT ENVIRONMENT

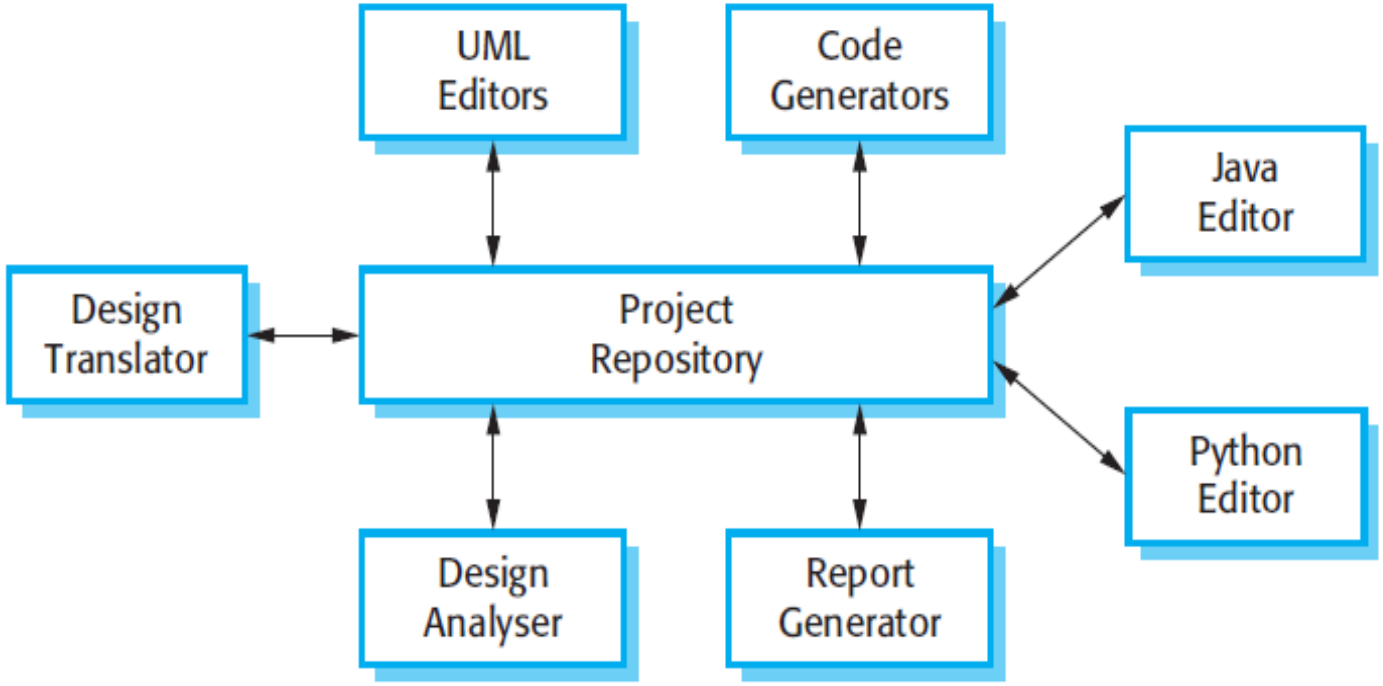


Figure 6.9 A repository architecture for an IDE

APPLICABLE DOMAINS OF REPOSITORY ARCHITECTURE:

- Suitable for large, complex information systems where many software component clients need to access them in different ways
- Requires data transactions to drive the control flow of computation

ADVANTAGES

- Components can be independent—they do not need to know of the existence of other components.
- Changes made by one component can be propagated to all components.
  - All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.

DISADVANTAGES

- The repository is a single point of failure so problems in the repository affect the whole system.
- May be inefficiencies in organizing all communication through the repository.
- Distributing the repository across several computers may be difficult.

BLACKBOARD ARCHITECTURE

BLACKBOARD ARCHITECTURE

- The blackboard architecture was developed for speech recognition applications in the 1970s.

- Other applications for this architecture are image pattern recognition and weather broadcast systems.

BLACKBOARD ARCHITECTURE

- The word blackboard comes from classroom teaching and learning.
- Teachers and students can share data in solving classroom problems via a blackboard.
- Students and teachers play the role of agents to contribute to the problem solving.
- They can all work in parallel, and independently, trying to find the best solution.

BLACKBOARD ARCHITECTURE

The entire system is decomposed into two major partitions.

- One partition, called the blackboard, is used to store data.
- while the other partition, called knowledge sources, stores domain specific knowledge.
- There also may be a third partition, called the controller, that is used to initiate the blackboard and knowledge sources and that takes a main role and overall supervision control.

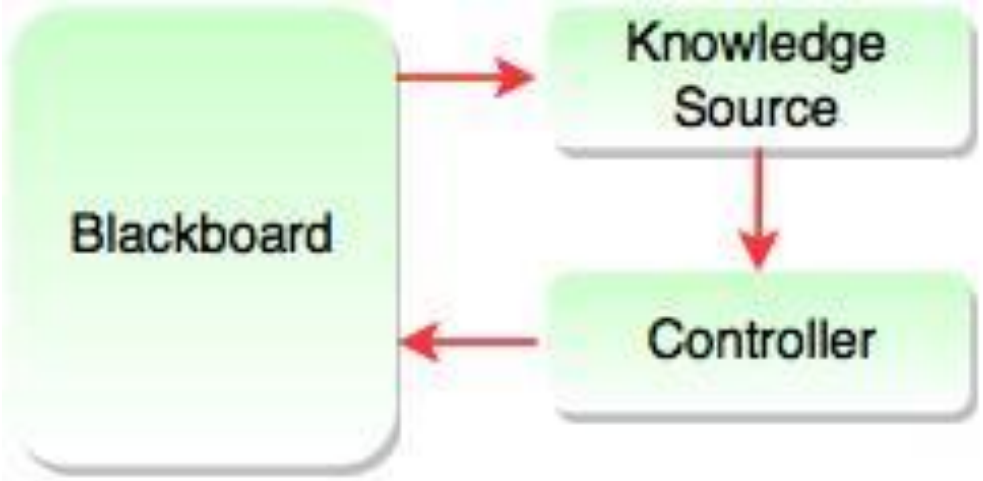
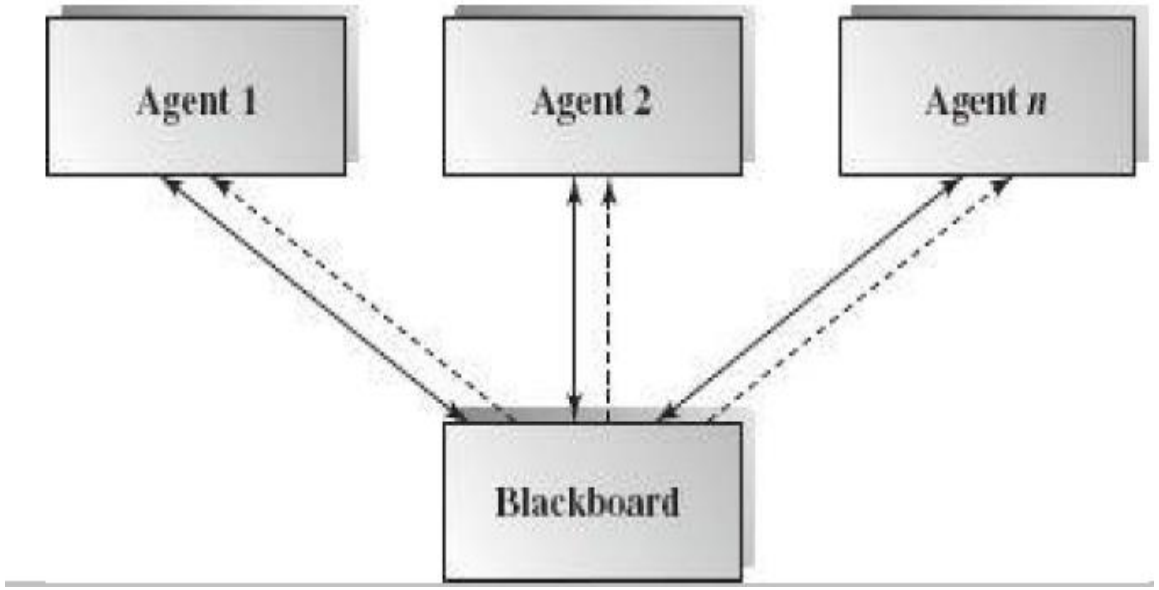


Fig. Blackboard Architectural Style

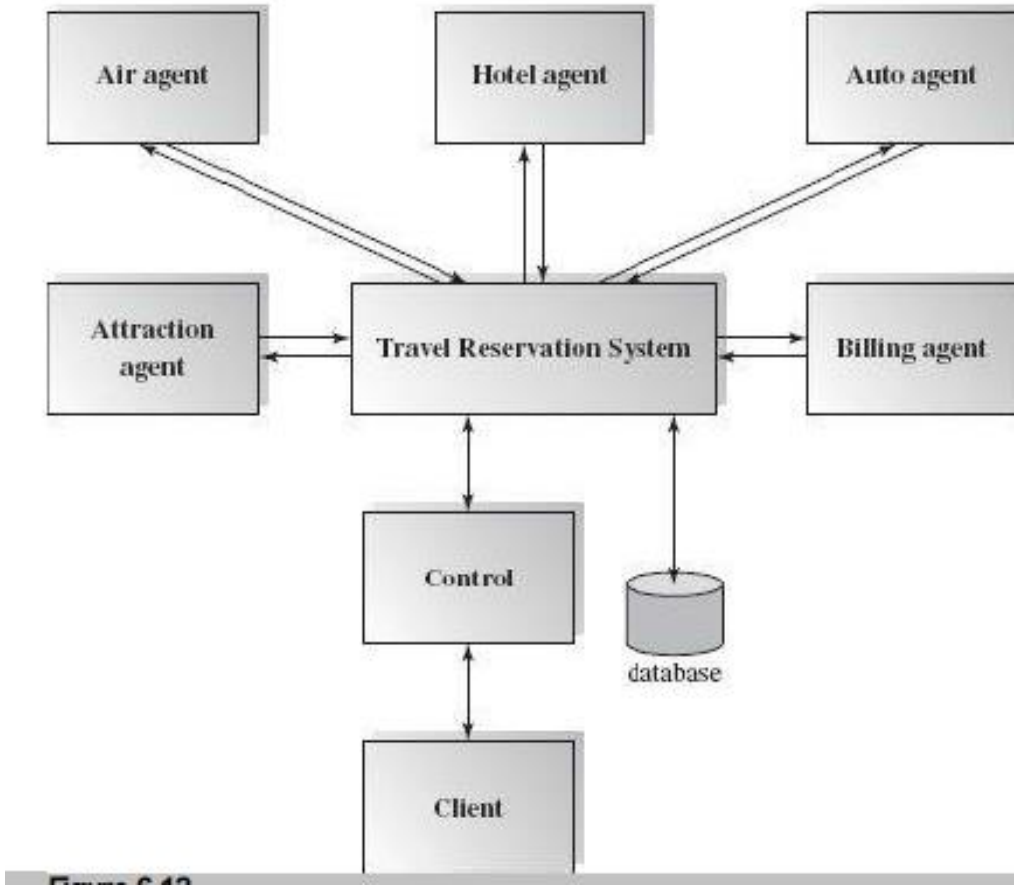
BLACKBOARD ARCHITECTURE - CONNECTIONS

- Data changes in the blackboard trigger one or more matched knowledge source to continue processing.
- This connection can be implemented in publish/subscribe mode.



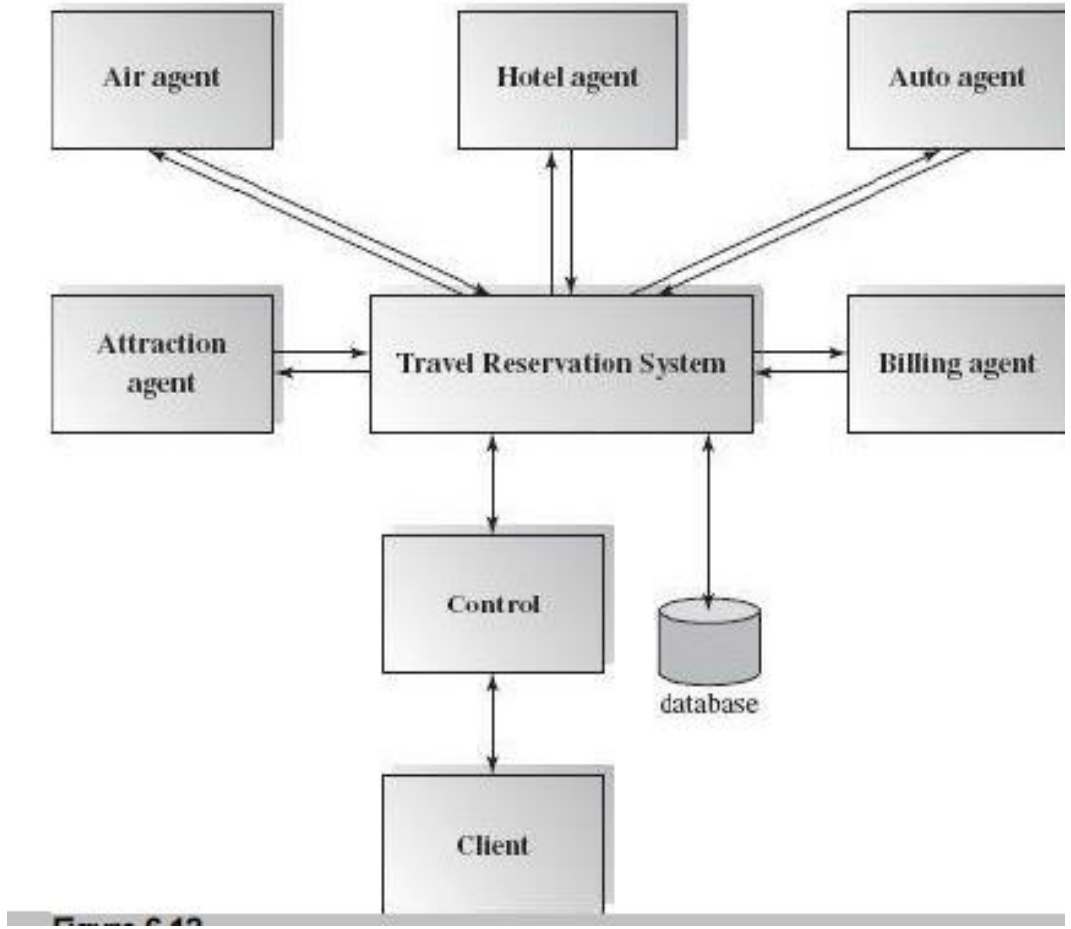
EXAMPLE - TRAVEL CONSULTING SYSTEM

There may be many air travel agencies, hotel reservation systems, car rental companies, or attraction reservation systems to subscribe to or register with through this travel planning system.



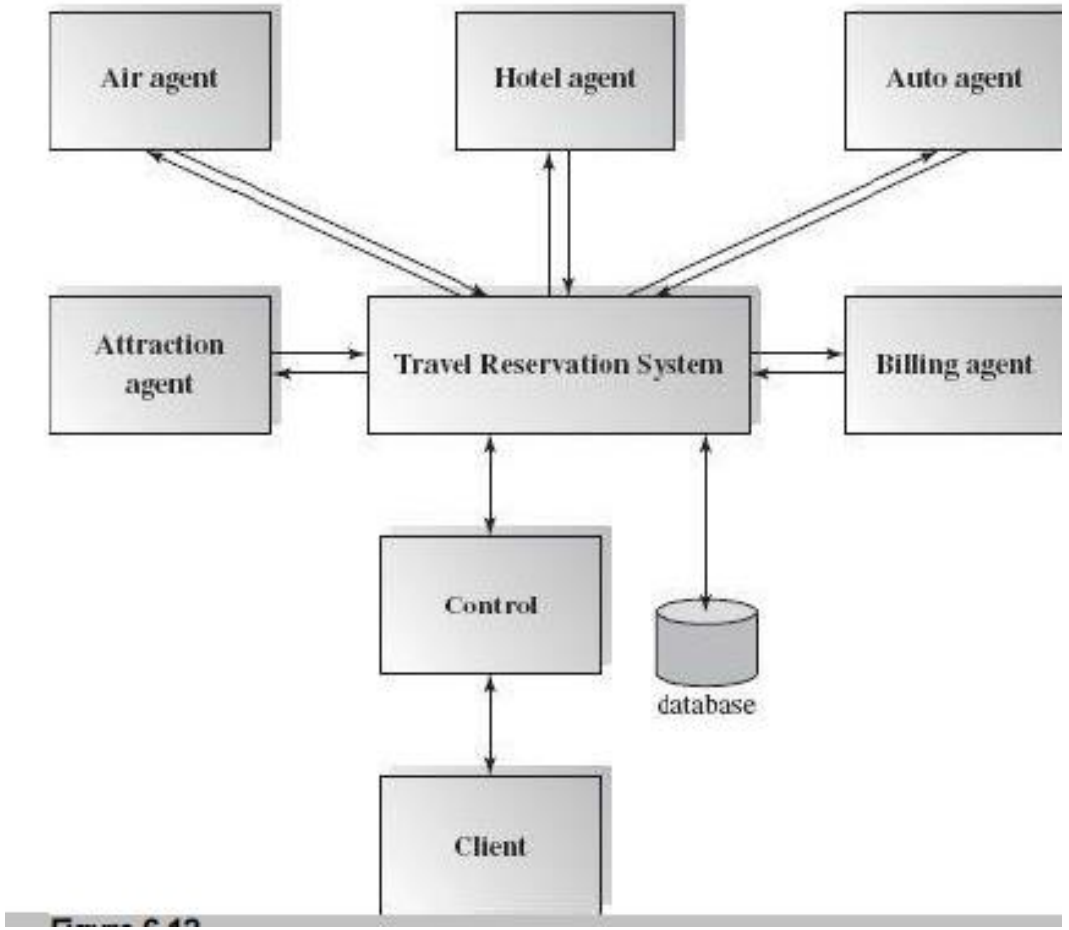
EXAMPLE - TRAVEL CONSULTING SYSTEM

- Once the system receives a client request, it publishes the request to all related agents and composes plan options for clients to choose from.



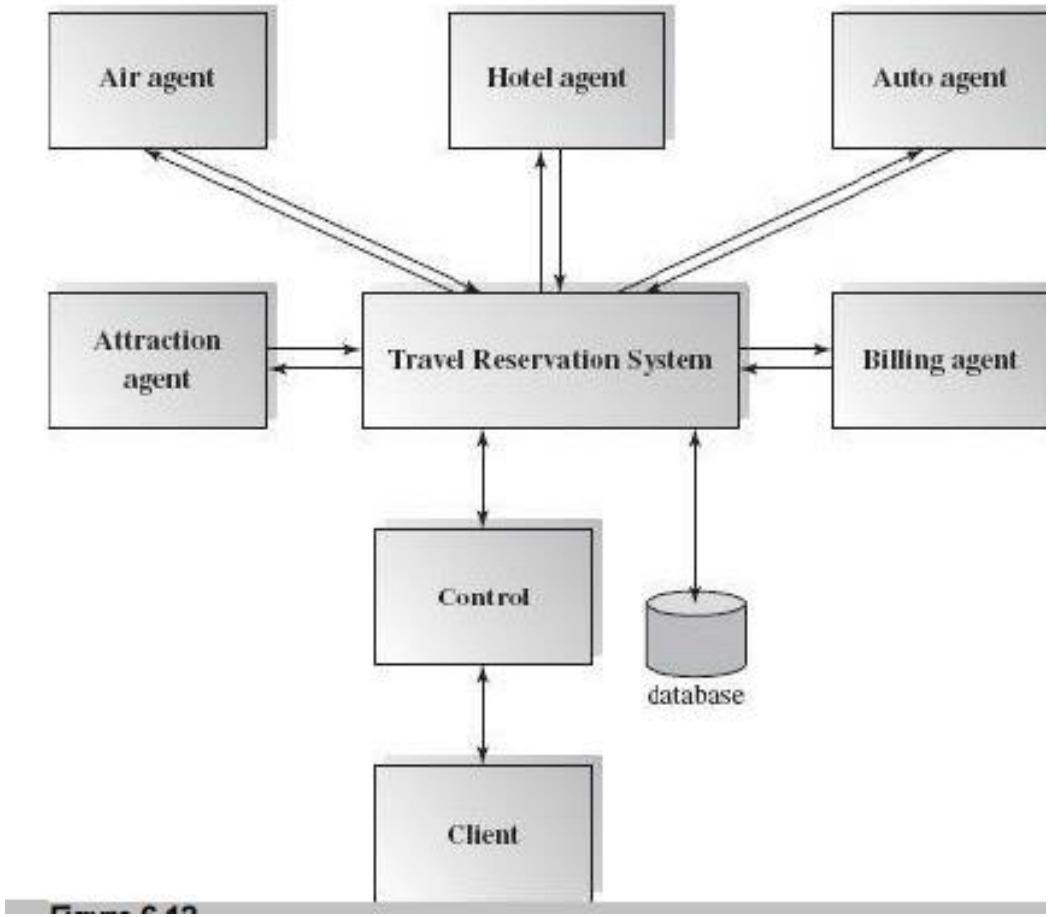
EXAMPLE - TRAVEL CONSULTING SYSTEM

- The system also stores all necessary data in the database.
- After the system receives a confirmation from the client, it invokes the financial billing system to verify credit background and to issue invoices.



EXAMPLE - TRAVEL CONSULTING SYSTEM

- The data in the data store plays an active role in this system.
- It does not require much user interaction after the system receives client requests since the request data will direct the computation and activate all related knowledge sources to solve the problem.



APPLICABLE DOMAIN:

- Suitable for solving complex problems such as artificial intelligence (AI) problems where no preset solutions exist.

BENEFITS:

- Scalability: easy to add or update knowledge source.
- Concurrency: all knowledge sources can work in parallel since they are independent of each other.
- Reusability of knowledge source agents.

COMPONENT BASED SOFTWARE ARCHITECTURE

COMPONENT-BASED SOFTWARE ENGINEERING

- CBSE is an approach to software development that relies on reuse
- CBSE emerged from the failure of object-oriented development to support reuse effectively
- Objects (classes) are too specific and too detailed to support design for reuse work

COMPONENT

- A software component is an independently deployable implementation of some functionality, to be reused as it is in a broad spectrum of applications.

- For a software component what it provides and requires should be clearly stated so that it can be used without any ambiguity.

COMPONENT BASED SOFTWARE ARCHITECTURE

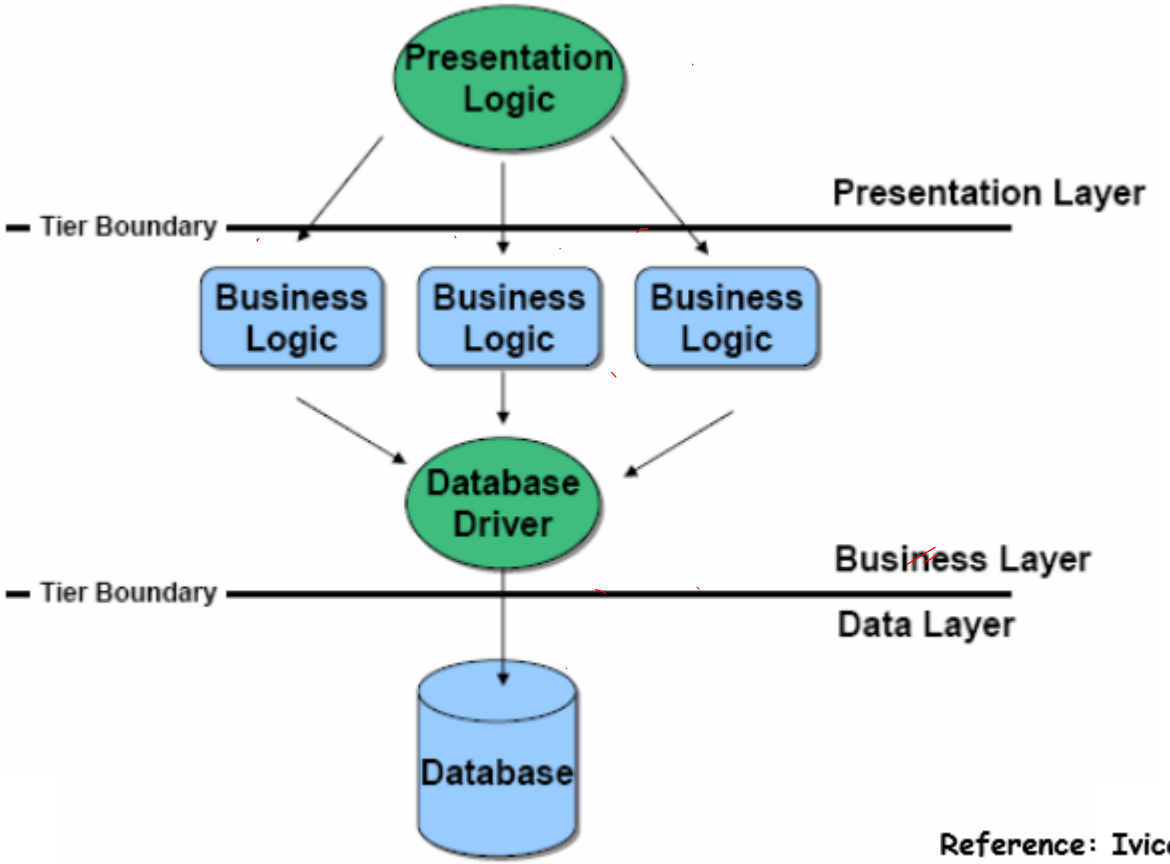
- The main motivation behind component-based design is component reusability.

- Designs can make use of existing reusable commercial off-the-shelf (COTS) components or ones developed in-house, and they may produce reusable components for future reuse.

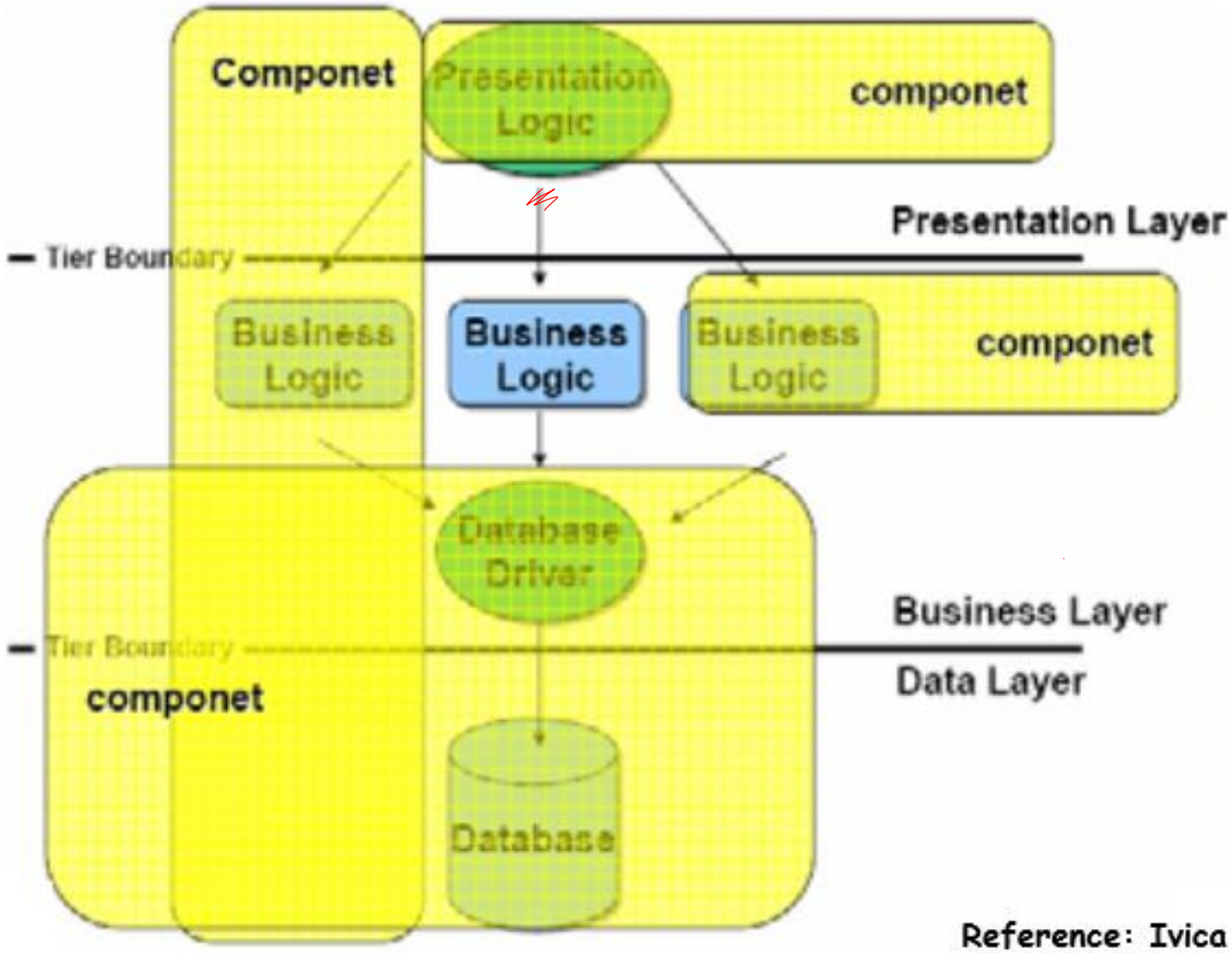
CATEGORIES OF COMPONENTS

- Commercial of-the-shelf components- complete application libraries readily available in the market.
- Qualified components—assessed by software engineers to ensure that not only functionality, but also performance, reliability, usability, and other quality factors conform to the requirements of the system/product to be built.
- Adapted components—adapted to modify (wrapping) unwanted or undesired characteristics.

N TIER ARCHITECTURE



Reference: Ivica Crnkovic



Reference: Ivica Crnkovic

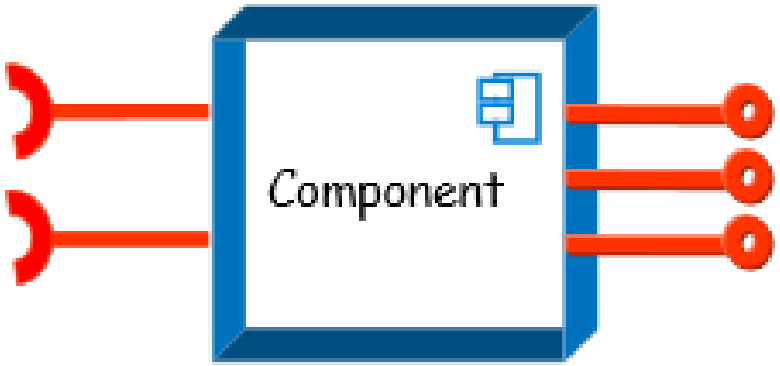
CONNECTORS

Connectors connect components, specifying and ruling their interaction.

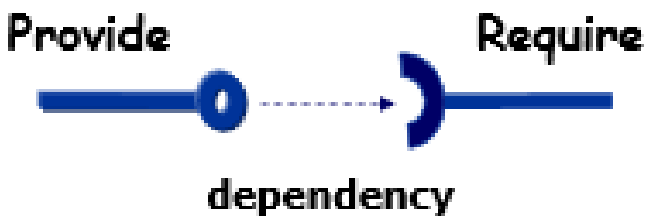
Component interaction can take the form of

- method invocations,
- asynchronous invocations such as
  - event listener and registrations,
  - broadcasting,

Requires interface
Defines the services that the component uses from the environment



Provides interface
Defines the services that are provided by the component to other components



COMPONENT DIAGRAM

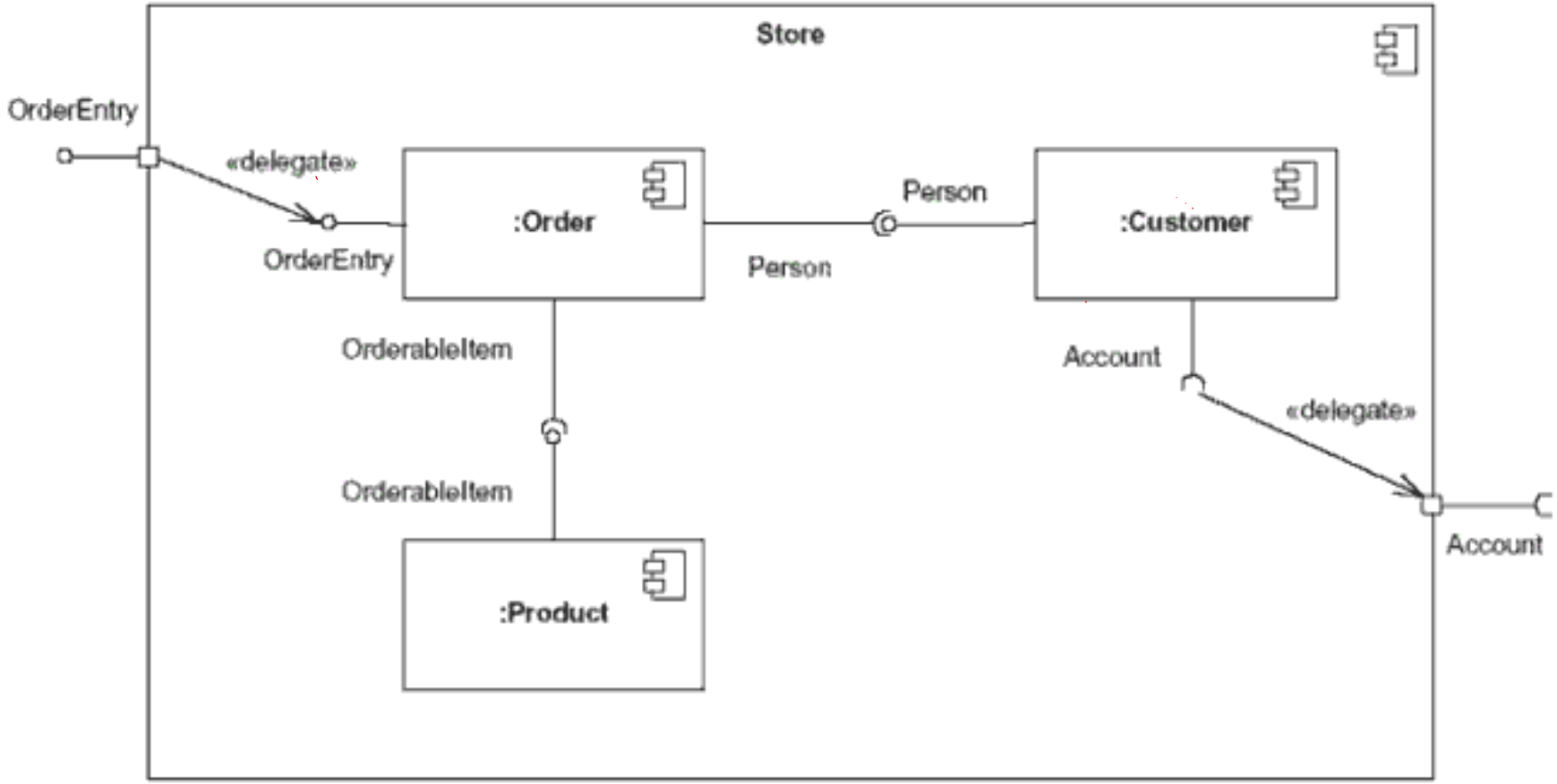
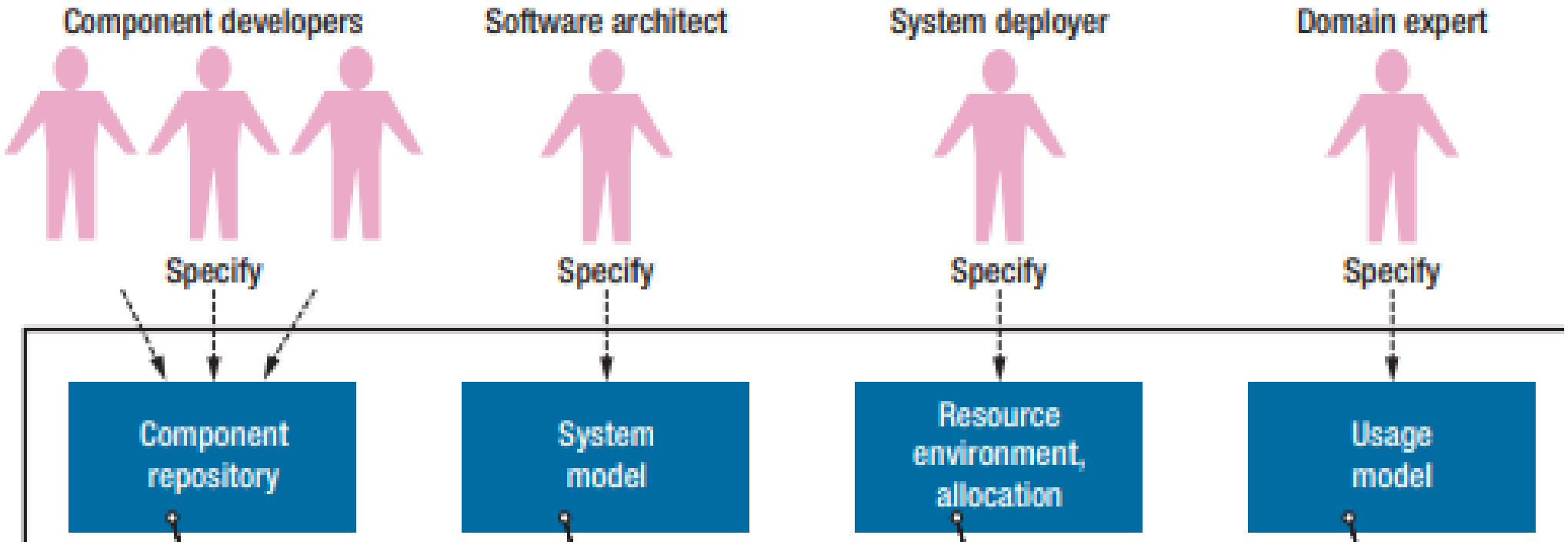
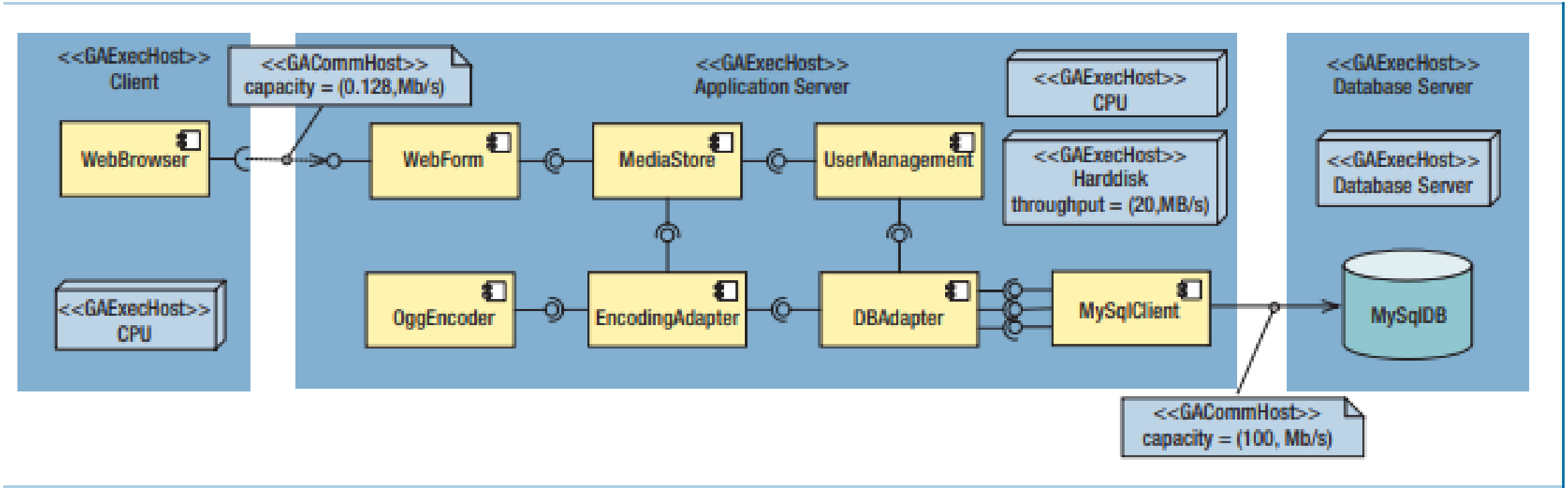


Figure: This component's inner structure is composed of other components


Component developers
Software architect
System deployer
Domain expert
Specify
Specify
Specify
Specify
Component repository
System model
Resource environment, allocation
Usage model


<<GAExecHost>>
Client
<<GACommHost>>
capacity = (0.128,Mb/s)
<<GAExecHost>>
Application Server
<<GAExecHost>>
CPU
<<GAExecHost>>
Database Server
WebBrowser
WebForm
MediaStore
UserManagement
<<GAExecHost>>
Harddisk
throughput = (20,MB/s)
<<GAExecHost>>
Database Server
<<GAExecHost>>
CPU
OggEncoder
EncodingAdapter
DBAdapter
MySqlClient
MySqlDB
<<GACommHost>>
capacity = (100, Mb/s)

COMPONENT BASED DEVELOPMENT

COMPONENT BASED DEVELOPMENT

This process basically consists of three main stages namely

- qualification,
- adaptation and
- composition

COMPONENT QUALIFICATION

Component qualification ensures that a candidate component

- will perform the function required,
- will properly fit into the architectural style specified for the system, and
- will exhibit the quality characteristics (e.g., performance, reliability, usability) required for the application.

COMPONENT ADAPTION

- Most of the times even after the component has been qualified for use in the architecture it exhibits some conflicts.
- To soothe these conflicts certain techniques such as component wrapping is used.

WRAPPING

- White-Box wrapping – this wrapping involves code level modifications in the components to avoid conflicts.

  This is not widely used because the COTS products are not provided with the source code.

WRAPPING

- Grey-Box wrapping- applied when the component library provides a component extension language or API that enables conflicts to be removed.

- Black-box wrapping - introduction of pre- and post-processing at the component interface to remove or mask conflicts.

COMPONENT COMPOSITION

- Architectural style depends the connection between various components and their relationships.

- The design of the software system should be In such a way that most of the components are replaceable or can be reused in other systems.

BENEFITS

- Ideally the components for reuse would be verified and defect-free.
- As the component is reused in many software systems any defect if there would be detected and corrected. So after a couple of reuses the component will have no defects.

BENEFITS

- Productivity is increased as every time the code need not be re-written from the scratch.
- The time taken to design and write the code also decreases with the use of software components.

LIMITATIONS

- It can be difficult to find suitable available components to reuse.
- Adaptation of components is an issue.

DISTRIBUTED SOFTWARE ARCHITECTURE

DISTRIBUTED SOFTWARE ARCHITECTURE

- A distributed system is a collection of computational and storage devices connected through a communications network.

- Data, software, and users are distributed.

CLIENT SERVER ARCHITECTURAL STYLE

CLIENT SERVER ARCHITECTURAL STYLE

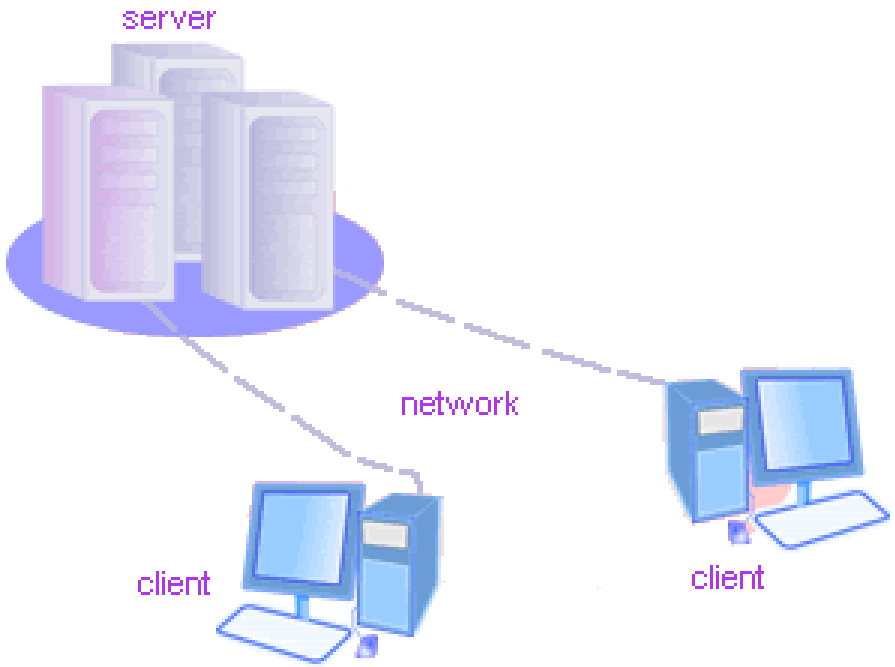
- Client/server architecture illustrates the relationship between two computer programs in which one program is a client, and the other is Server.

- **Client** makes a service request to server.
- **Server** provides service to the request.

CLIENT/SERVER

- Although the client/server architecture can be used within a single computer by programs, but it is a more important idea in a network.

- In a network, the client/server architecture allows efficient way to interconnect programs that are distributed efficiently across different locations.



CLIENT-SERVER STYLE

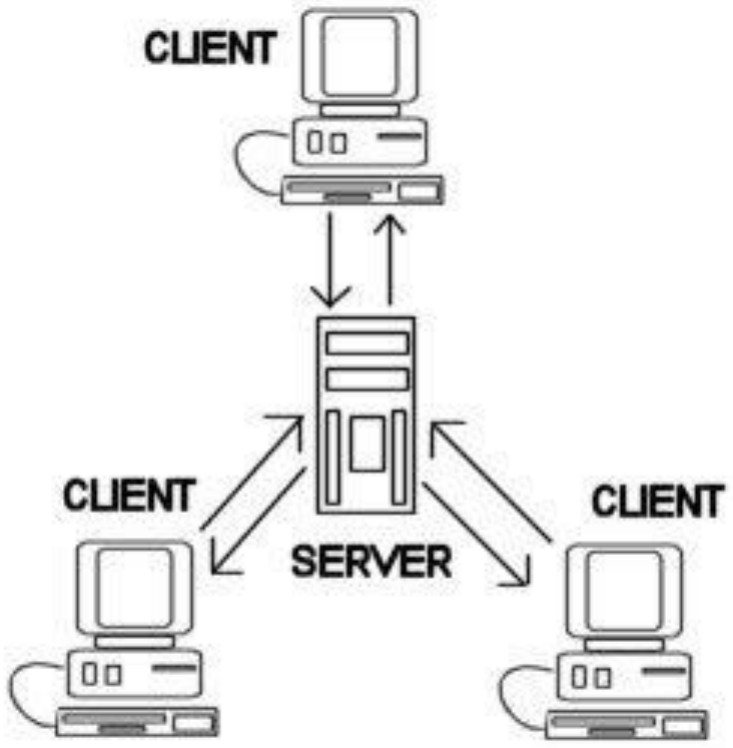
- Suitable for applications that involve distributed data and processing across a range of components.

Components:

- **Servers:** Stand-alone components that provide specific services such as printing, data management, etc.
- **Clients:** Components that call on the services provided by servers.
- **Connector:** The network, which allows clients to access remote servers.

COMMON EXAMPLE

- The World Wide Web is an example of client-server architecture.
- Each computer that uses a Web browser is a client, and the data on the various Web pages that those clients access is stored on multiple servers.



ANOTHER EXAMPLE

- If you have to check a bank account from your computer, you have to send a request to a server program at the bank.
- That program processes the request and forwards the request to its own client program that sends a request to a database server at another bank computer to retrieve client balance information.
- The balance is sent back to the bank data client, which in turn serves it back to your personal computer, which displays the information of balance on your computer.

TYPES OF SERVERS

- **File Servers:**
  - Useful for sharing files across a network.
  - The client passes requests for files over the network to the file server.
- **Database Servers:**
  - Client passes SQL requests as messages to the DB server; results are returned over the network to the client.
  - Query processing done by the server.
  - No need for large data transfers.

MULTI-TIER CLIENT SERVER ARCHITECTURE

TYPES OF CLIENT SERVER

Two-tier client–server architecture,

- which is used for simple client–server systems, and in situations where it is important to centralize the system for security reasons.
- In such cases, communication between the client and server is normally encrypted.

- Multitier client–server architecture,
  - which is used when there is a high volume of transactions to be processed by the server.

A TWO-TIER CLIENT–SERVER ARCHITECTURE

The system is implemented as a single logical server plus an indefinite number of clients that use that server.

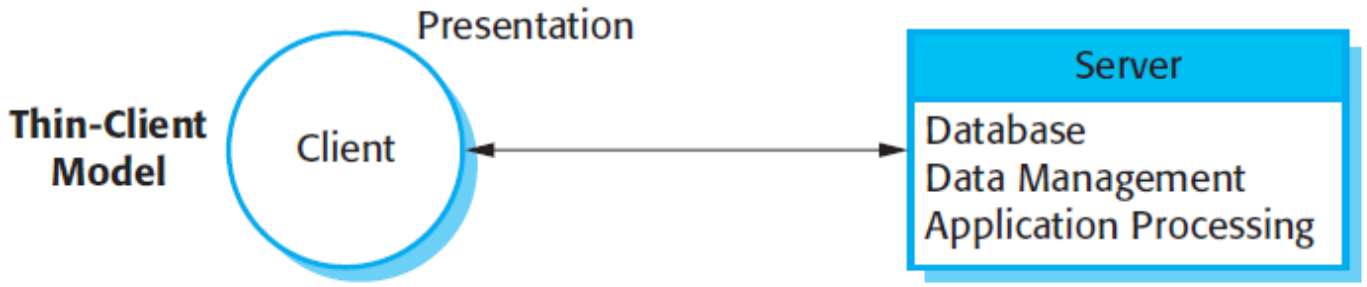
Two forms of this architectural model:

- A thin-client model,
- A fat-client model,

TWO-TIER CLIENT SERVER

A thin-client model,

- where the presentation layer is implemented on the client and all other layers (data management, application processing, and database) are implemented on a server.



ADVANTAGES

The advantage of the thin-client model is that it is simple to manage the clients.

- This is a major issue if there are a large number of clients, as it may be difficult and expensive to install new software on all of them. If a web browser is used as the client, there is no need to install any software.

DISADVANTAGES

The disadvantage of the thin-client approach, however is that it may place a heavy processing load on both the server and the network.

- The server is responsible for all computation and this may lead to the generation of significant network traffic between the client and the server.

TWO-TIER CLIENT SERVER

A fat-client model,

- where some or all of the application processing is carried out on the client.
- Data management and database functions are implemented on the server.

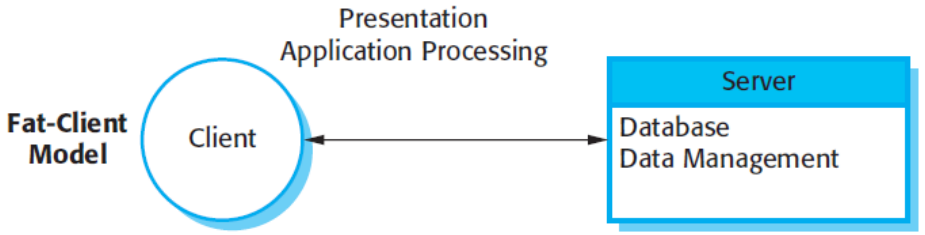
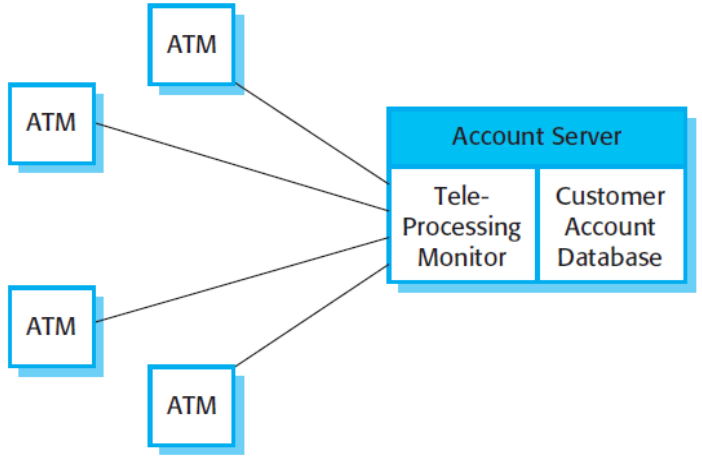


Figure 18.9 A fat-client architecture for an ATM system



MULTI-TIER CLIENT–SERVER ARCHITECTURES

- The fundamental problem with a two-tier client–server approach is that the logical layers in the system—presentation, application processing, data management, and database—must be mapped onto two computer systems: the client and the server.

- This may lead to problems with scalability and performance if the thin-client model is chosen, or problems of system management if the fat-client model is used.

MULTI-TIER CLIENT–SERVER ARCHITECTURES

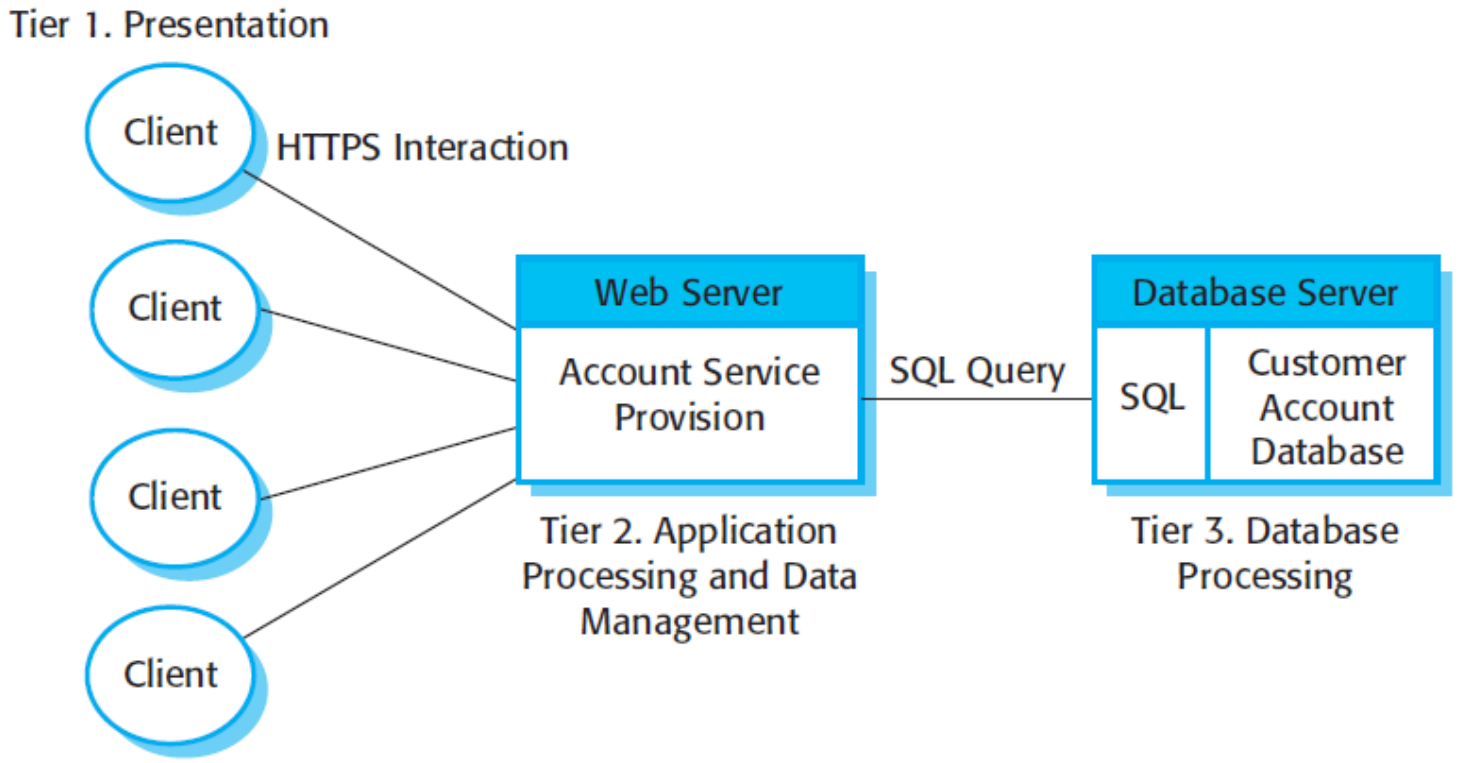


Figure 18.10 Three-tier architecture for an Internet banking system

MULTI-TIER CLIENT–SERVER ARCHITECTURES

- This system is scalable because it is relatively easy to add servers (scale out) as the number of customers increase.

- In this case, the use of a three-tier architecture allows the information transfer between the web server and the database server to be optimized.

SERVICE ORIENTED ARCHITECTURE (SOA)

SERVICE ORIENTED ARCHITECTURE (SOA)

- A service-oriented architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network.

- The principles of service-orientation are independent of any product, vendor or technology.

WHAT IS SERVICE?

A service is a self-contained, self-describing and modular piece of software that performs a specific business function such as validating a credit card or generating an invoice.

- The term self-contained implies services include all that is needed to get them working.
- Self-describing means they have interfaces that describe their business functionalities.
- Modular means services can be aggregated to form more complex applications.

EXAMPLE

A single service provides a collection of capabilities, often grouped together within a functional context as established by business requirements.

For example, the functional context of the service below is account. Therefore, this service provides the set of operations associated with a customer's account.

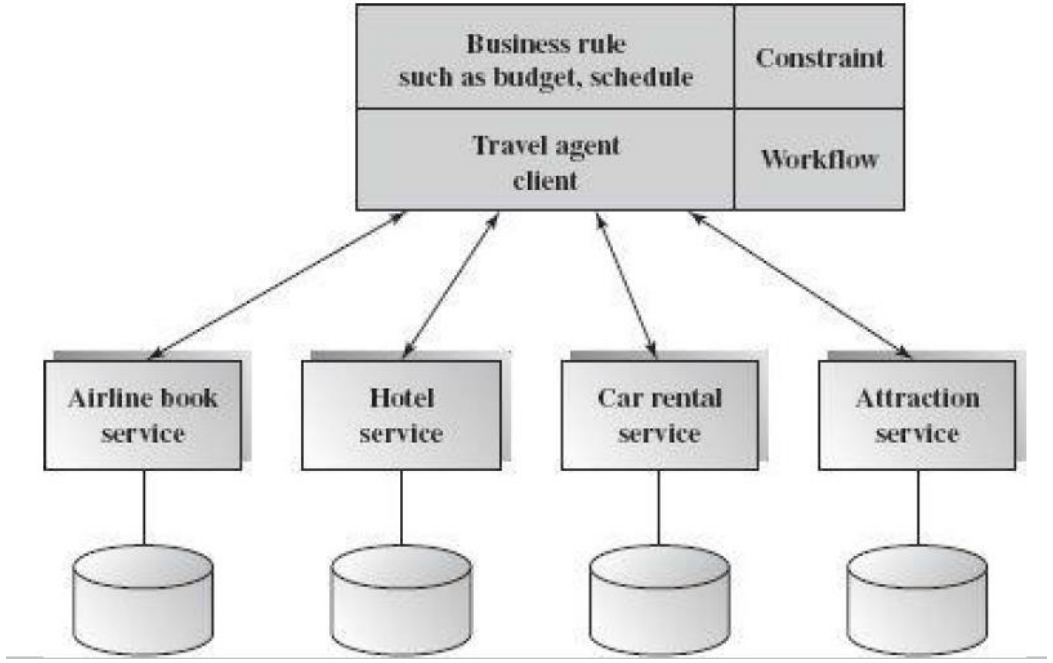
BANKING EXAMPLE

- Imagine that several areas of banking applications will deal with the current balance of an existing customer.
- More than often, the "get current balance" functionality is repeated in various applications within a banking environment.
- This gives rise to a redundant programming scenario.
- The focus should be toward finding this sort of common, reusable functionality and implement it as a service, so
  - that all banking applications can reuse the service as and when necessary.

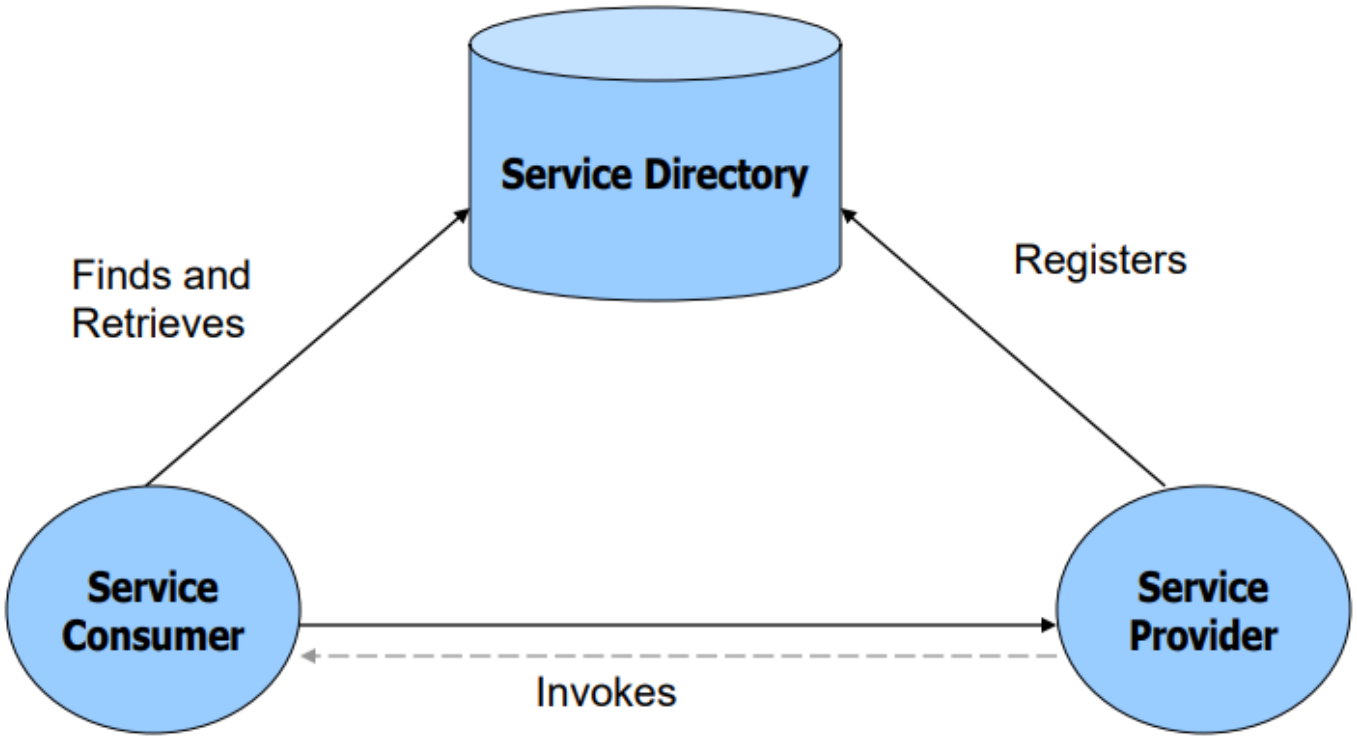
EXAMPLE

An online travel agency system

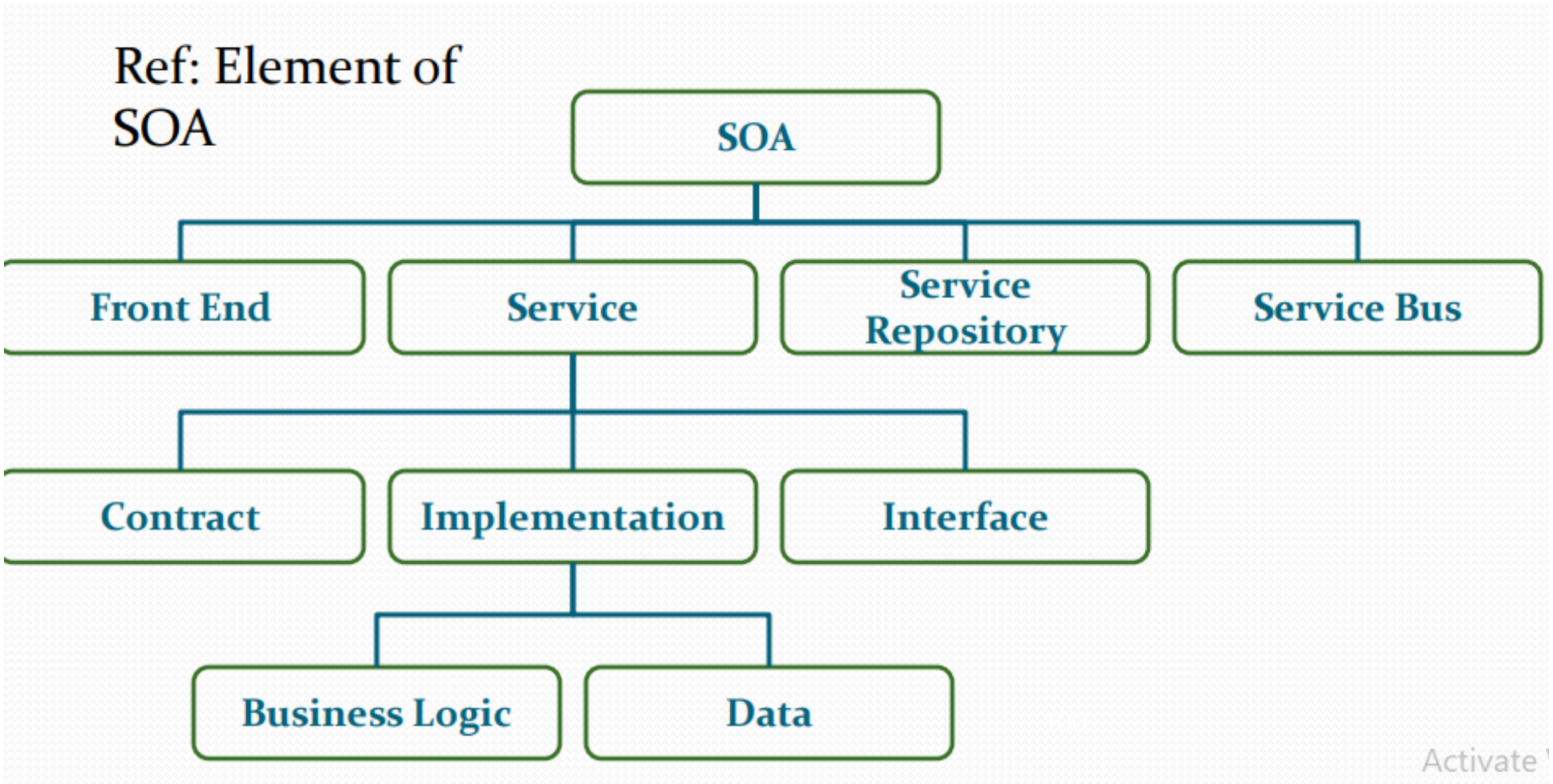
- It consists of four existing web services:
  - airline reservation,
  - car rental,
  - hotel reservation,
  - and attraction reservation.

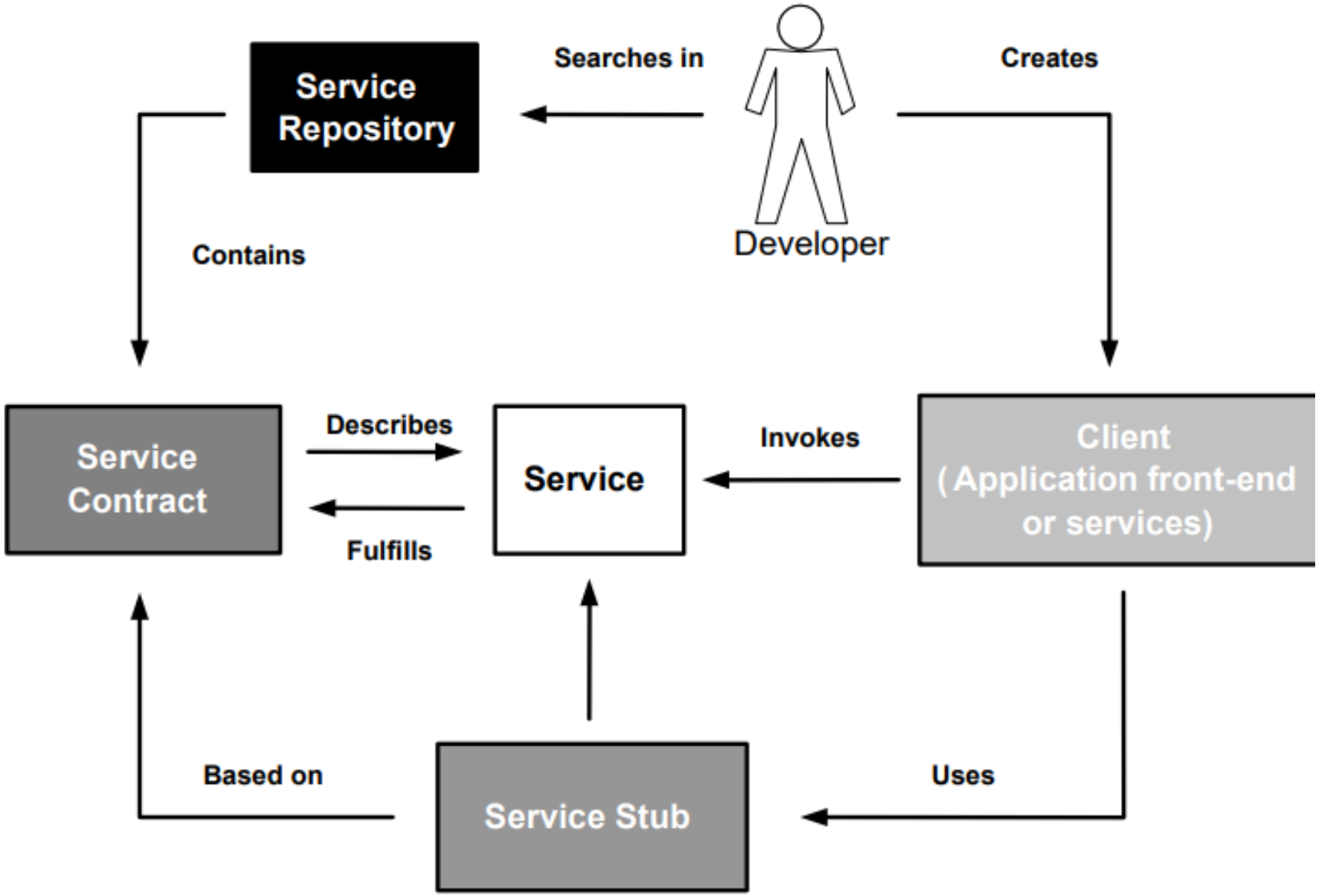


SOA ARCHITECTURE



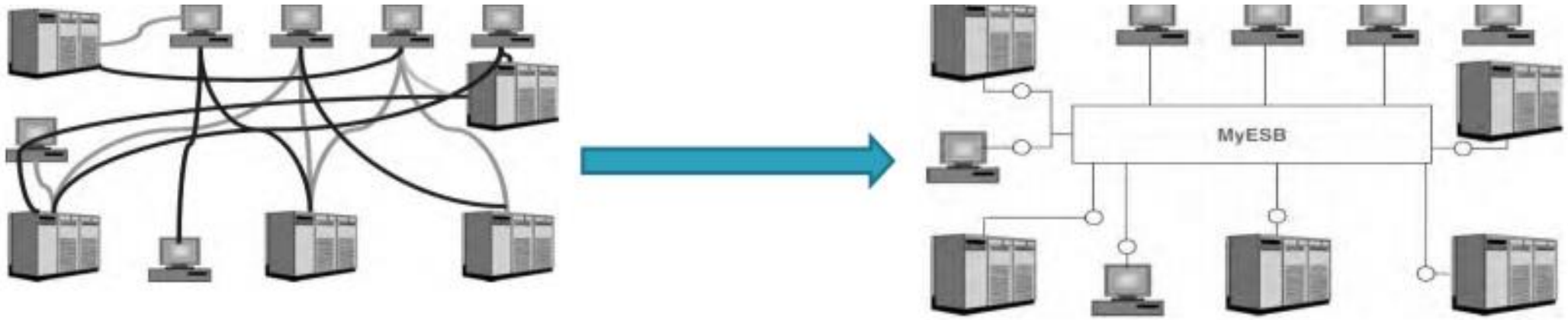
COMPONENTS OF SOA



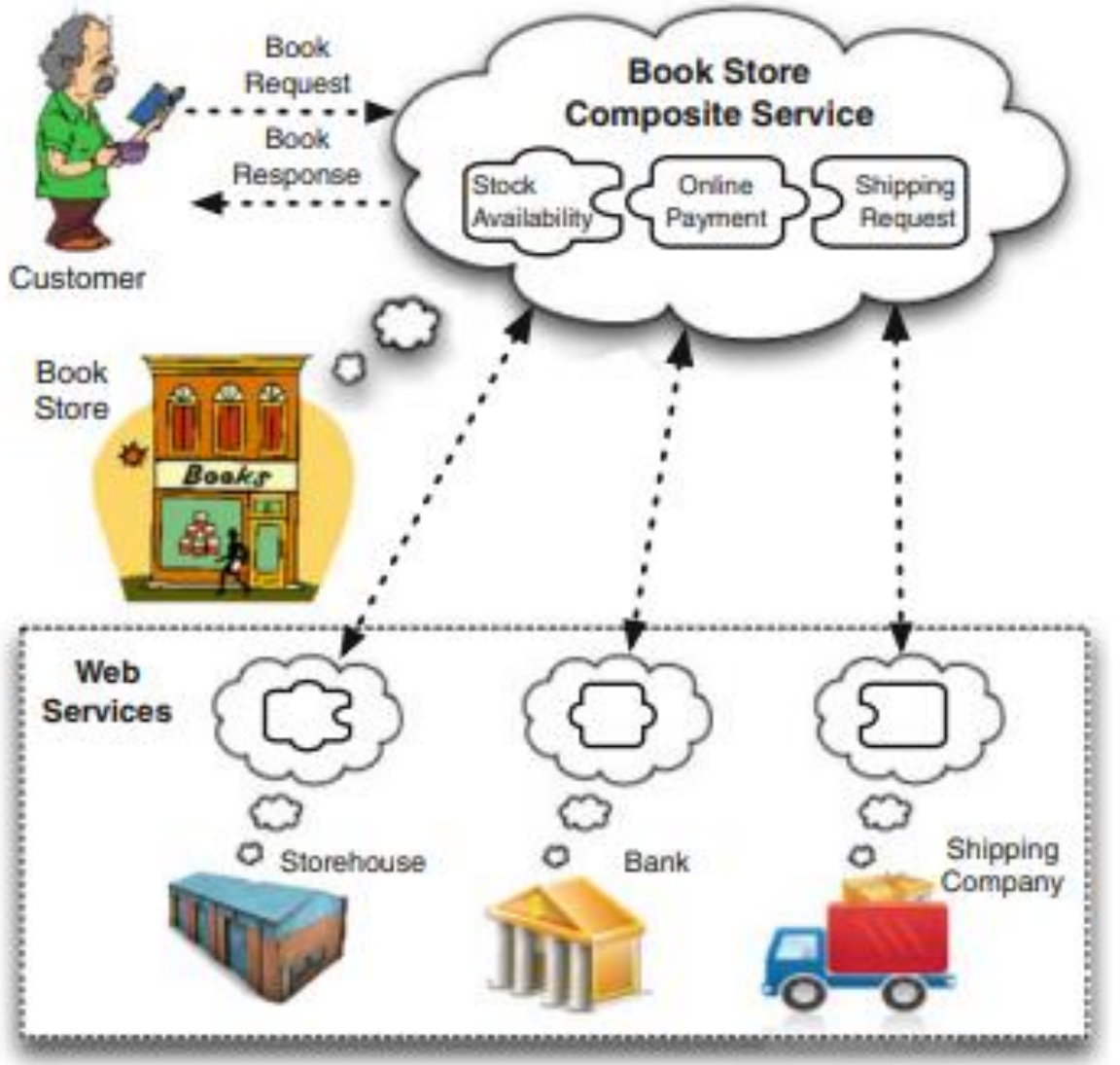

Service Repository
Searches in
Developer
Creates
Contains
Service Contract
Describes
Fulfills
Service
Invokes
Client (Application front-end or services)
Based on
Service Stub
Uses

ESB

- Enterprise Service Buses (ESBs) build on MOM (message-oriented middleware) to provide a flexible, scalable, standards-based integration technology for building a loosely coupled, highly-distributed SOA



SERVICE COMPOSITION

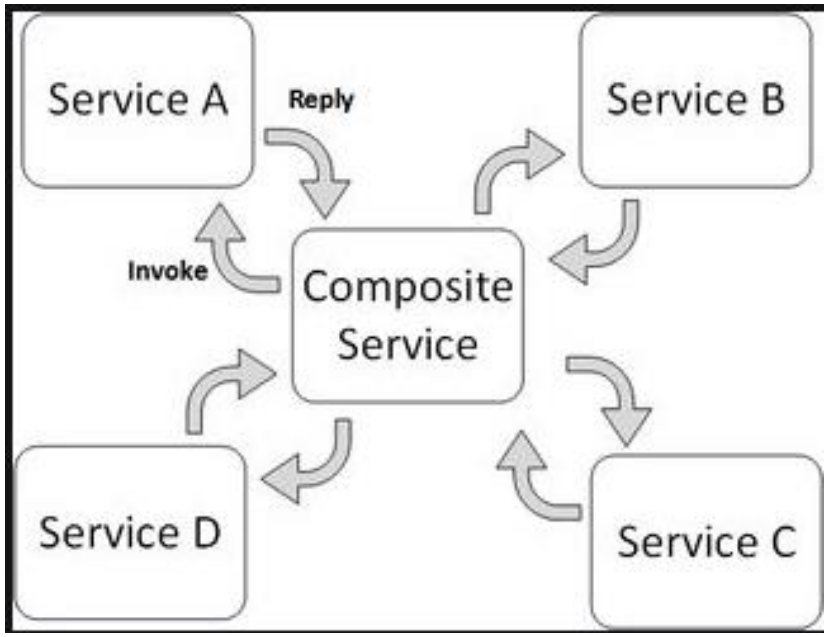


An example of Web service composition

SERVICE ORCHESTRATION VS SERVICE CHOREOGRAPHY

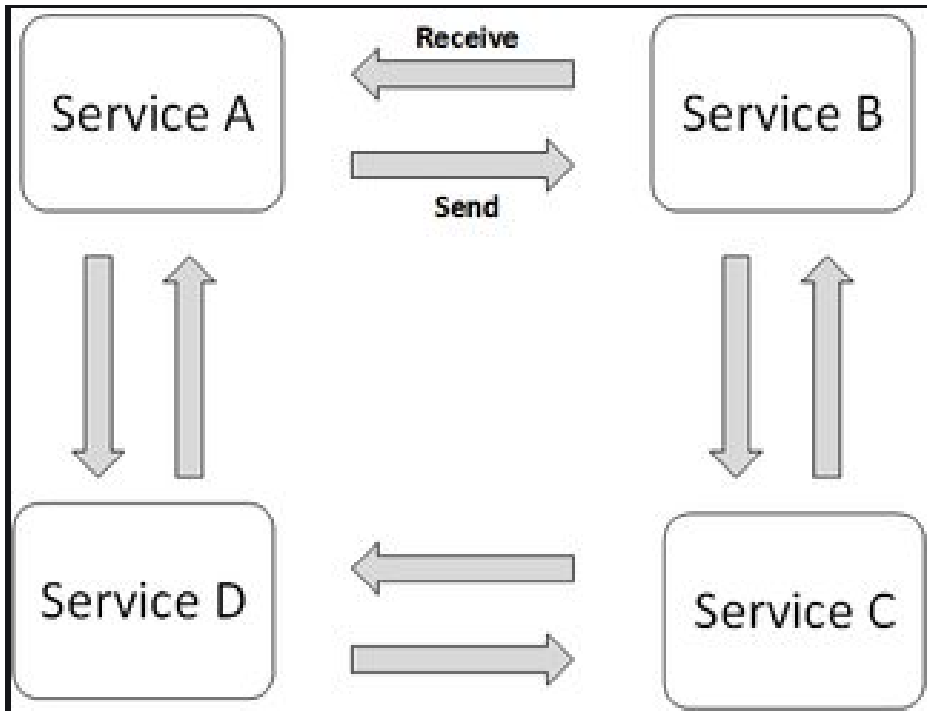
- Orchestration comes from orchestras; in an orchestra there is a conductor leading it, and there are musicians who play their instruments following the instructions of the conductor.

- The musicians know their role; they know how to play their instruments, but not necessarily directly interact with each other. That is, in an orchestra, there is a central system (the conductor) that coordinates and synchronizes the interactions of the components to perform a coherent piece of music.



SERVICE ORCHESTRATION VS SERVICE CHOREOGRAPHY

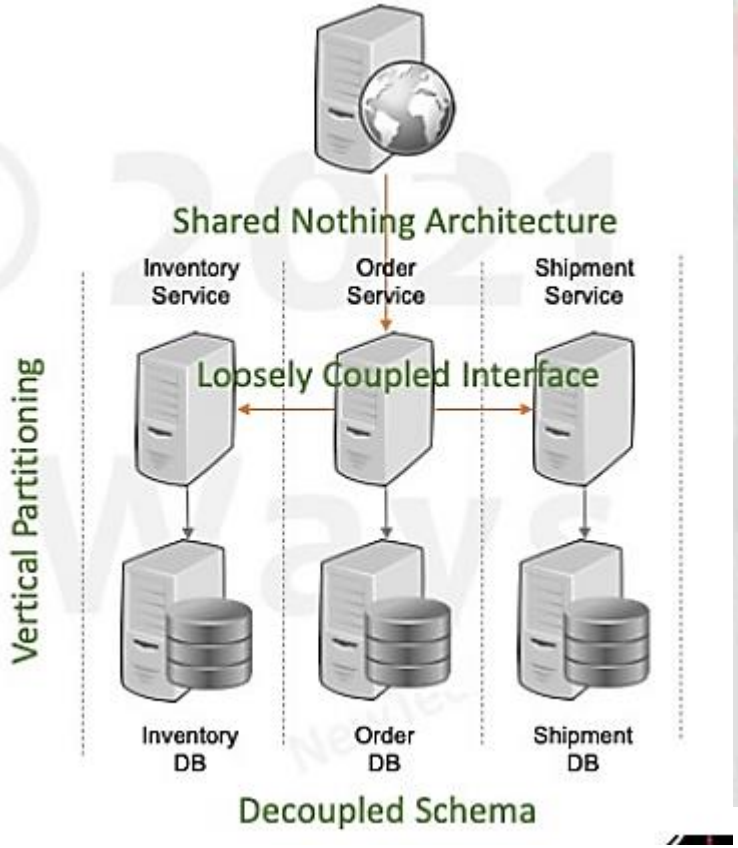
- Choreography is related to dance; on a dancing stage there are just dancers, but they know what they have to do and how to interact with the other dancers; each dancer is in charge of being synchronized with the rest.



MICROSERVICES

MICROSERVICES

- Microservices are small, individually deployable services performing different operations.
- Variant of SOA



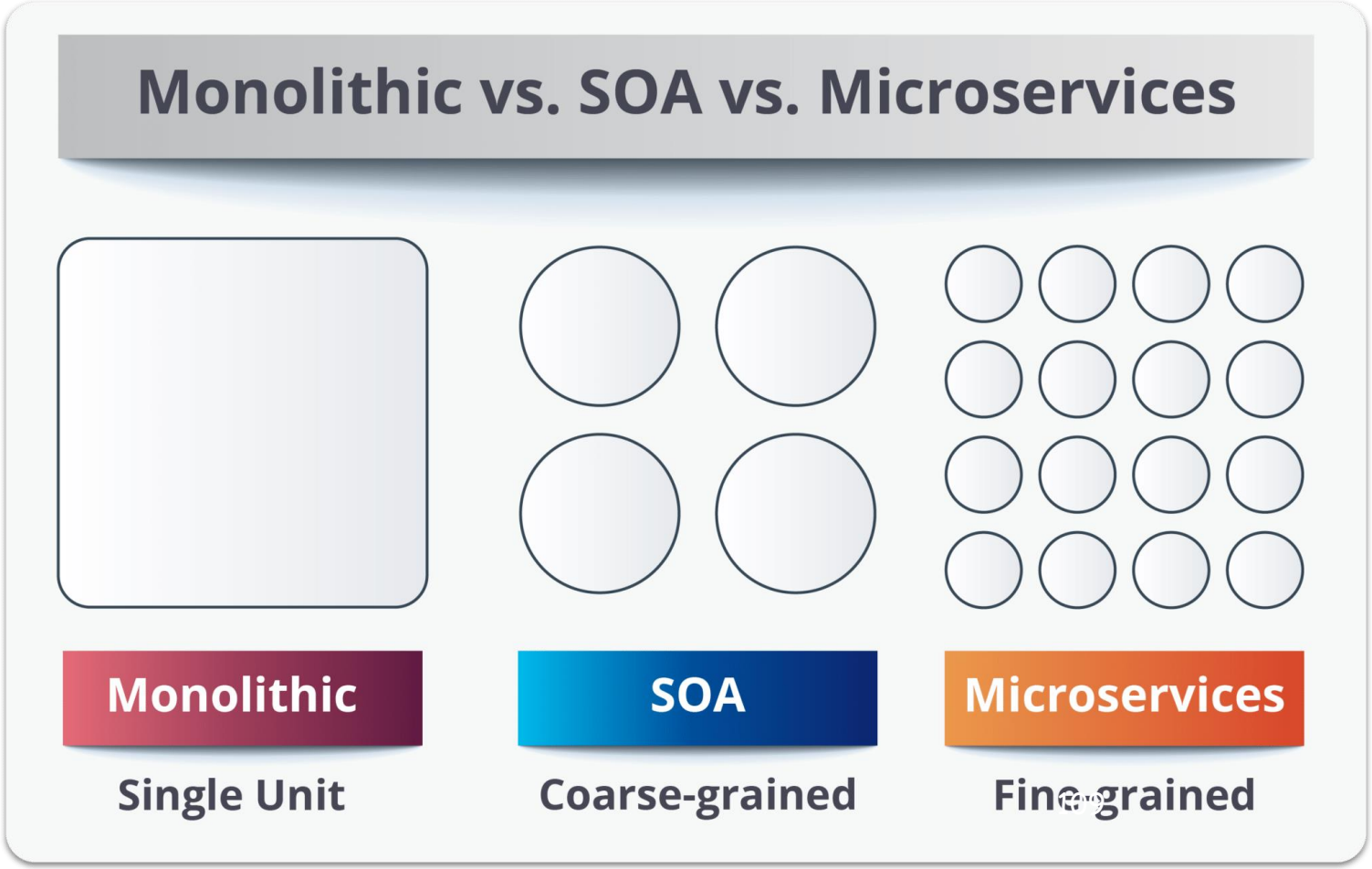
Frequent Deployment

Independent Deployment

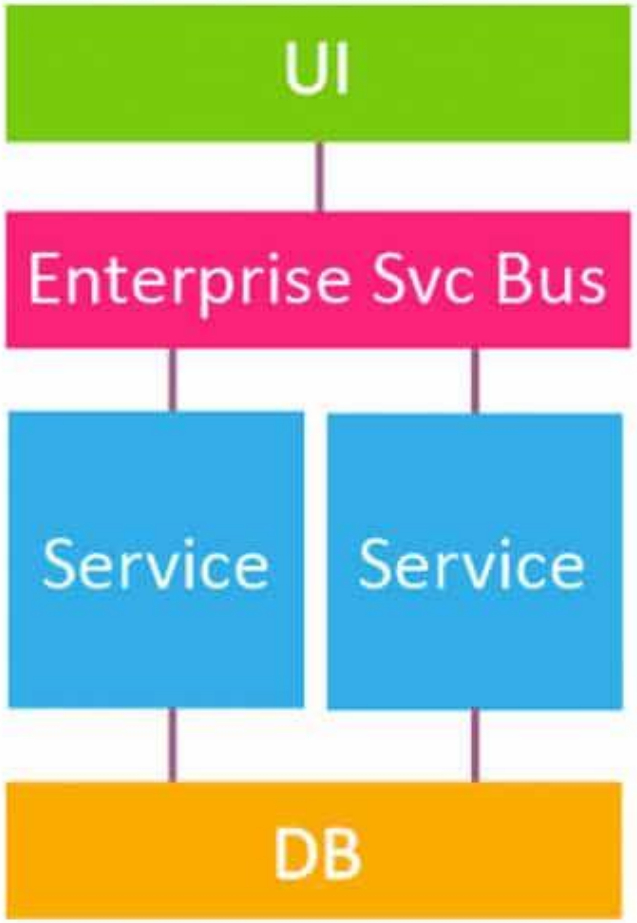
Independent Development

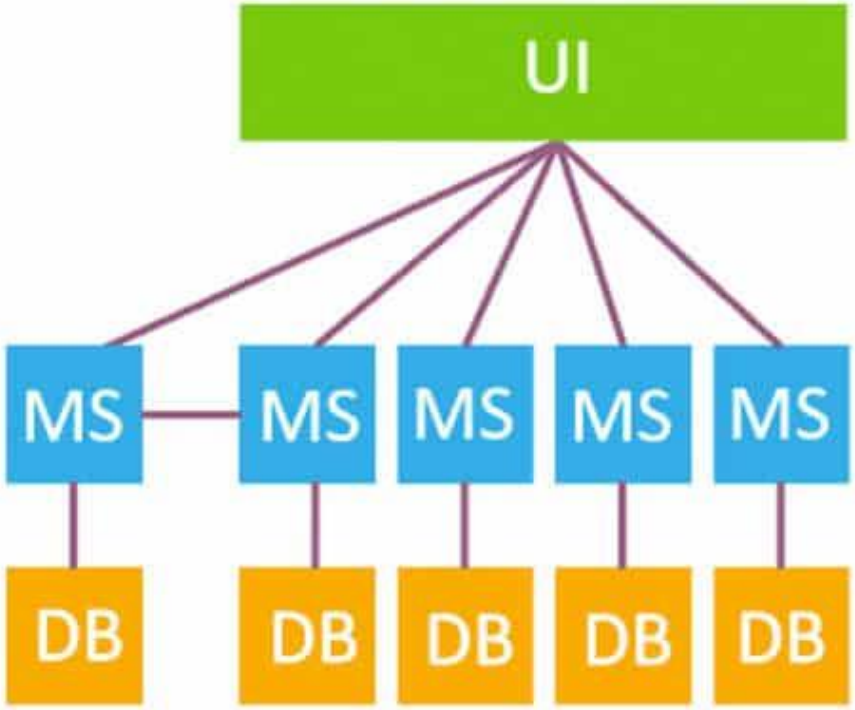
Independent Services

SOA VS MICROSERVICES

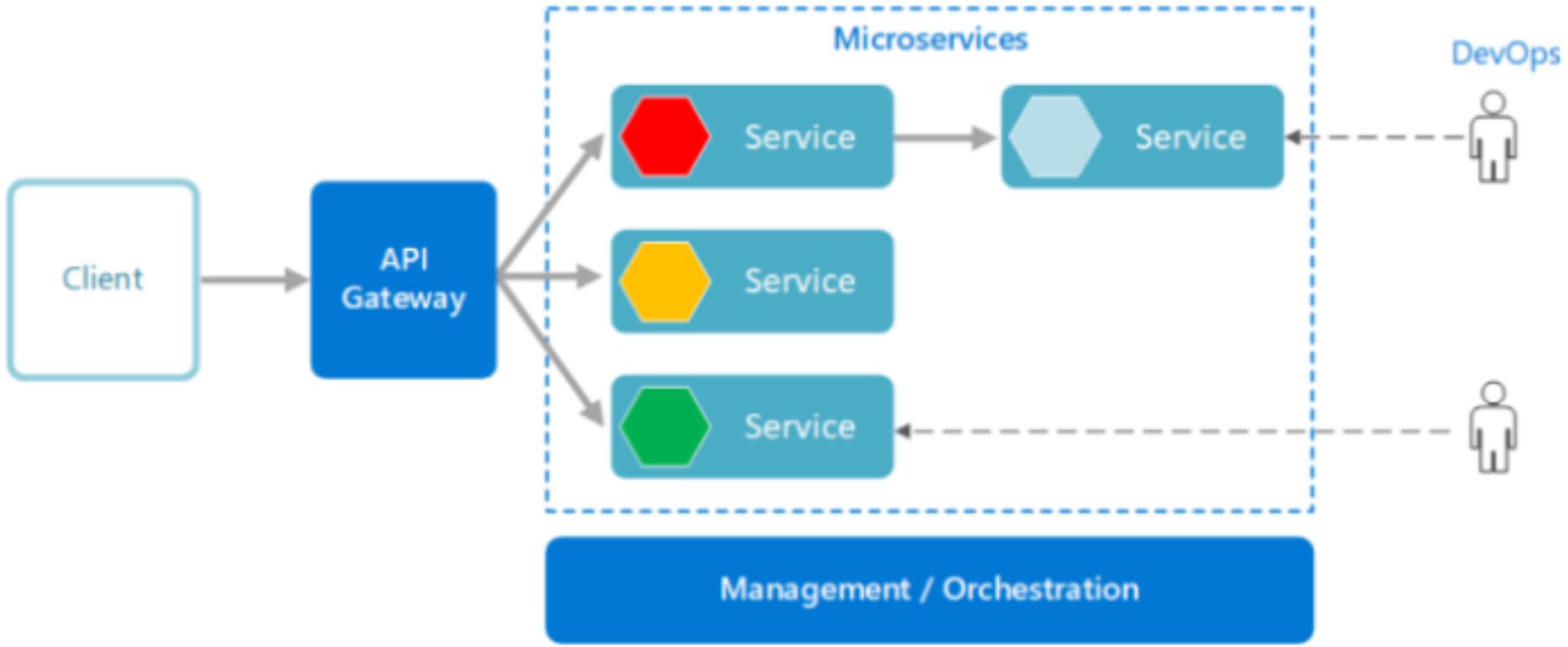


SOA VS MICROSERVICES





HOW DOES MICROSERVICES ARCHITECTURE WORK?

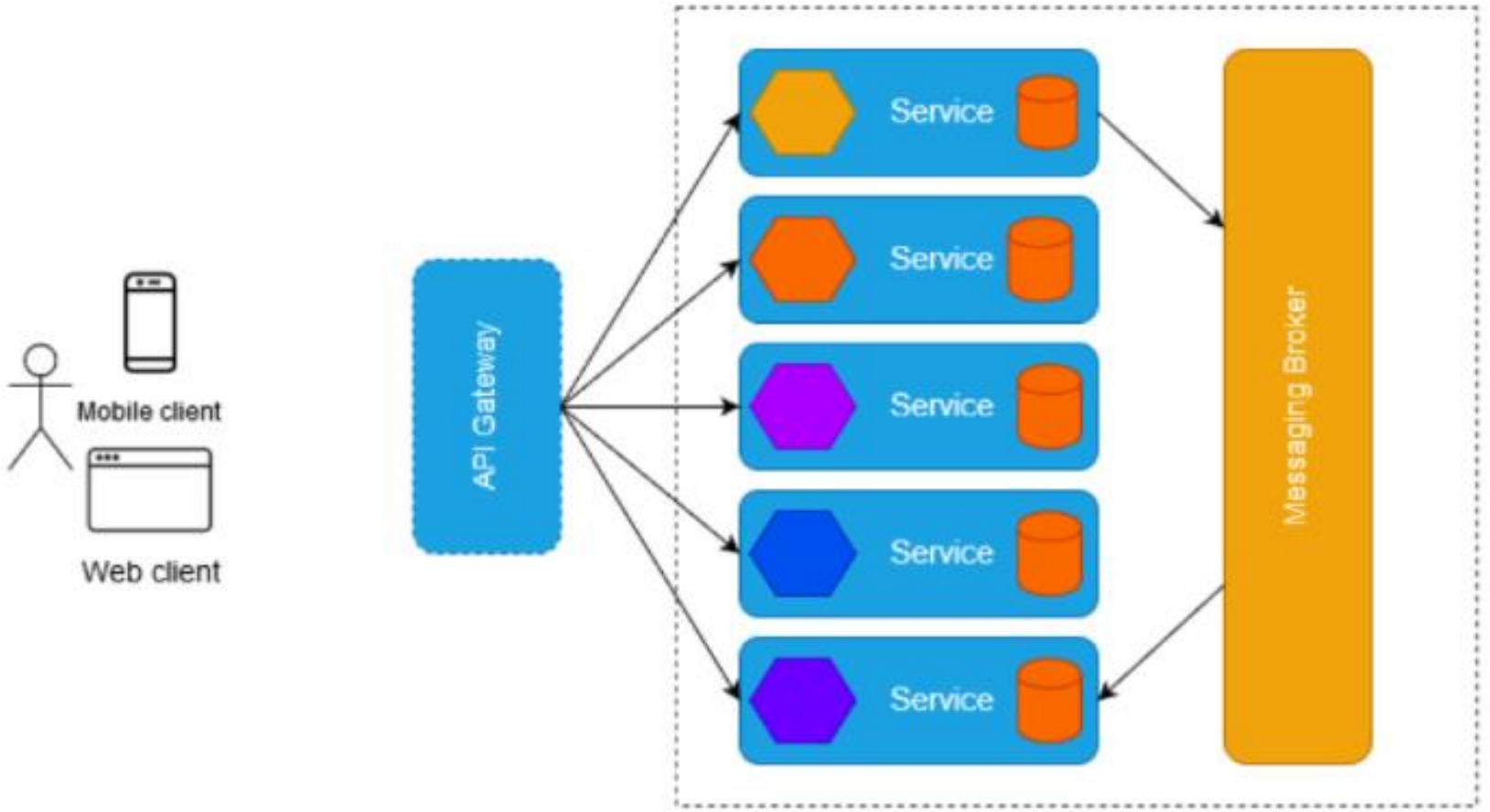


COMMUNICATION IN MICROSERVICES

Synchronous Messages:
In the situation where clients wait for the responses from a service, Microservices usually tend to use **REST (Representational State Transfer)** as it relies on a stateless, client-server, and the **HTTP protocol**. This protocol is used as it is a distributed environment each and every functionality is represented with a resource to carry out operations

Asynchronous Messages:
In the situation where clients do not wait for the responses from a service, Microservices usually tend to use protocols such as **AMQP, MQTT**.

REFERENCE ARCHITECTURE OF MICROSERVICES



UBER'S PREVIOUS ARCHITECTURE

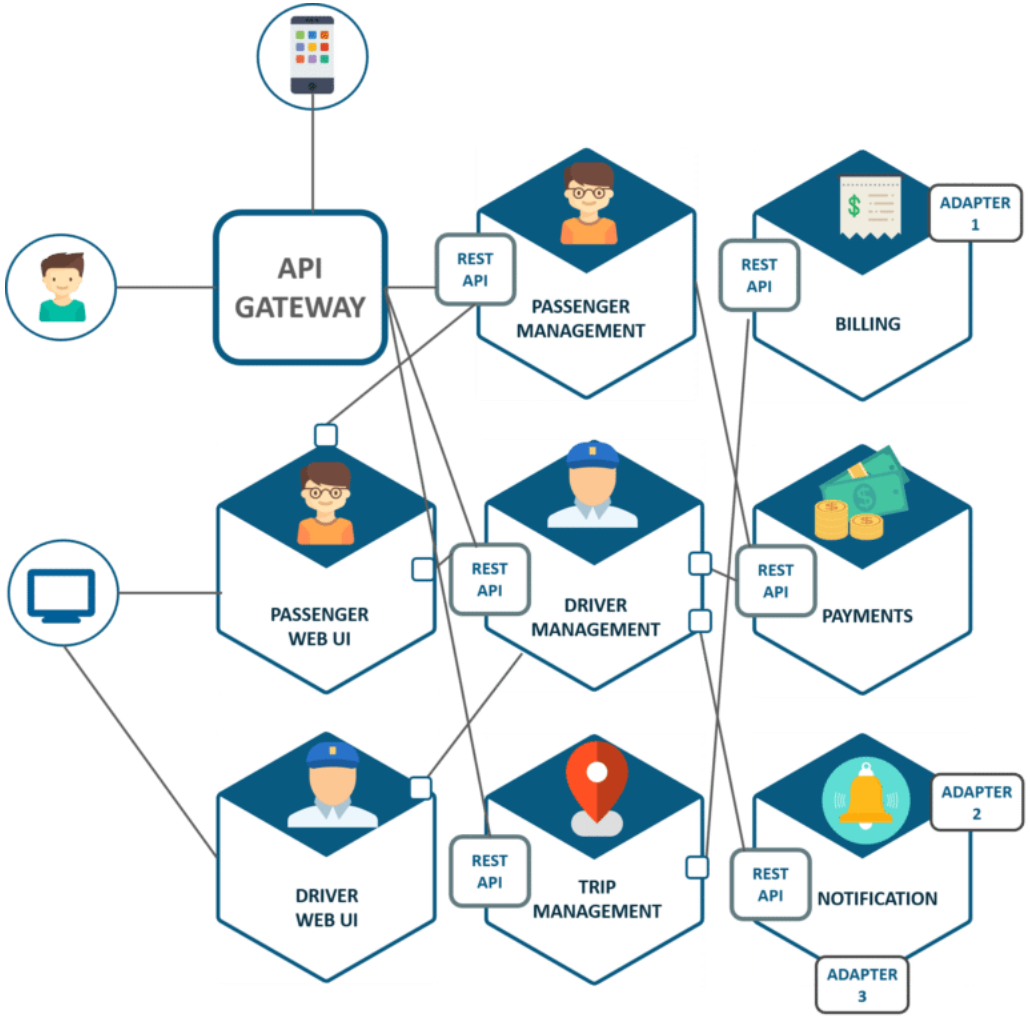
- A REST API is present with which the passenger and driver connect.
- Three different adapters are used with API within them, to perform actions such as billing, payments, sending emails/messages that we see when we book a cab.
- A MySQL database to store all their data.

All the features such as passenger management, billing, notification features, payments, trip management, and driver management were composed within a single framework.

PROBLEM IN UBER'S ARCHITECTURE

- All the features had to be re-built, deployed and tested again and again to update a single feature.
- Fixing bugs became extremely difficult in a single repository as developers had to change the code again and again.
- Scaling the features simultaneously with the introduction of new features was quite tough to be handled together.

SOLUTION IS MICROSERVICES ARCHITECTURE



SOLUTION

- The units are individual separate deployable units performing separate functionalities.
- For Example: If you want to change anything in the billing Microservices, then you just have to deploy only billing Microservices and don't have to deploy the others.
- All the features were now scaled individually i.e. The interdependency between each and every feature was removed.

DECOMPOSITION OF MICROSERVICES

There are some Prerequisite of decomposition of microservices.

Services must be cohesive.

- A service should implement a small set of strongly related functions.

Services must be loosely coupled

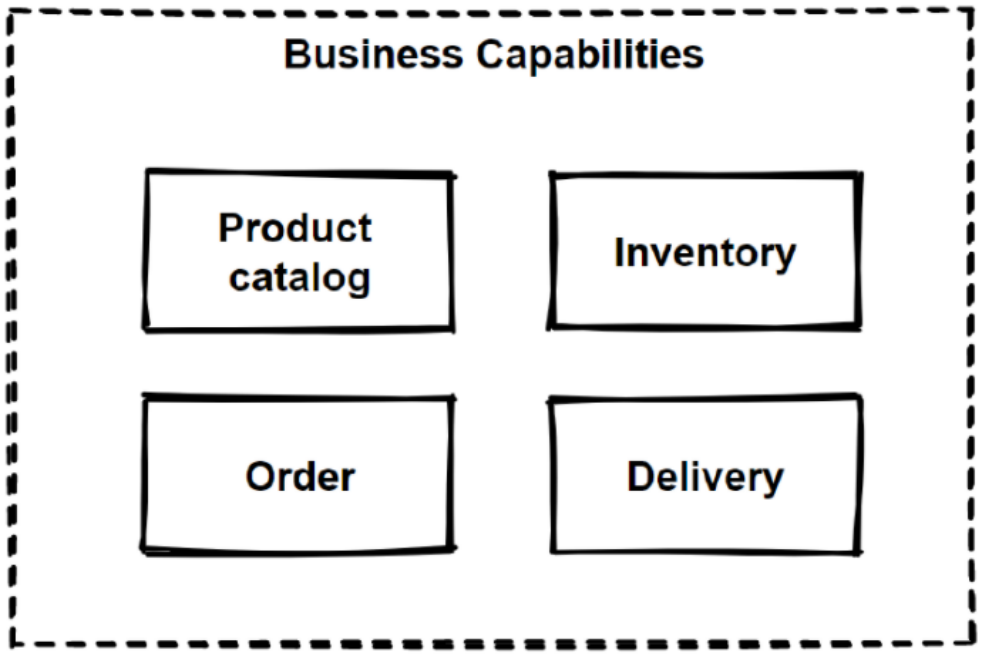
- Each service as an API that encapsulates its implementation.

DECOMPOSITION OF MICROSERVICES

Decompose by Business Capability" pattern offers that;

- Define services corresponding to business capabilities.
- A business capability is a concept from business architecture modeling.
- A business service should generate value.

For example; Order Management is responsible for orders, Customer Management is responsible for customers.



EVENT BASED SOFTWARE ARCHITECTURE

EVENT DRIVEN ARCHITECTURE

- Event-driven architecture refers to a system that exchange information between each other through the production and consumption of events.
- The main purpose of this type of communication architecture is to provide a decoupling between the event/message, the publishers/producers, and the subscribers/customers.
- These are very popular architectures in distributed applications.

EXAMPLE

- Uber's
- Fire Alarming System
- Used to enforce integrity constraints in **database management systems** (called triggers).

SYNCHRONOUS VS ASYNCHRONOUS

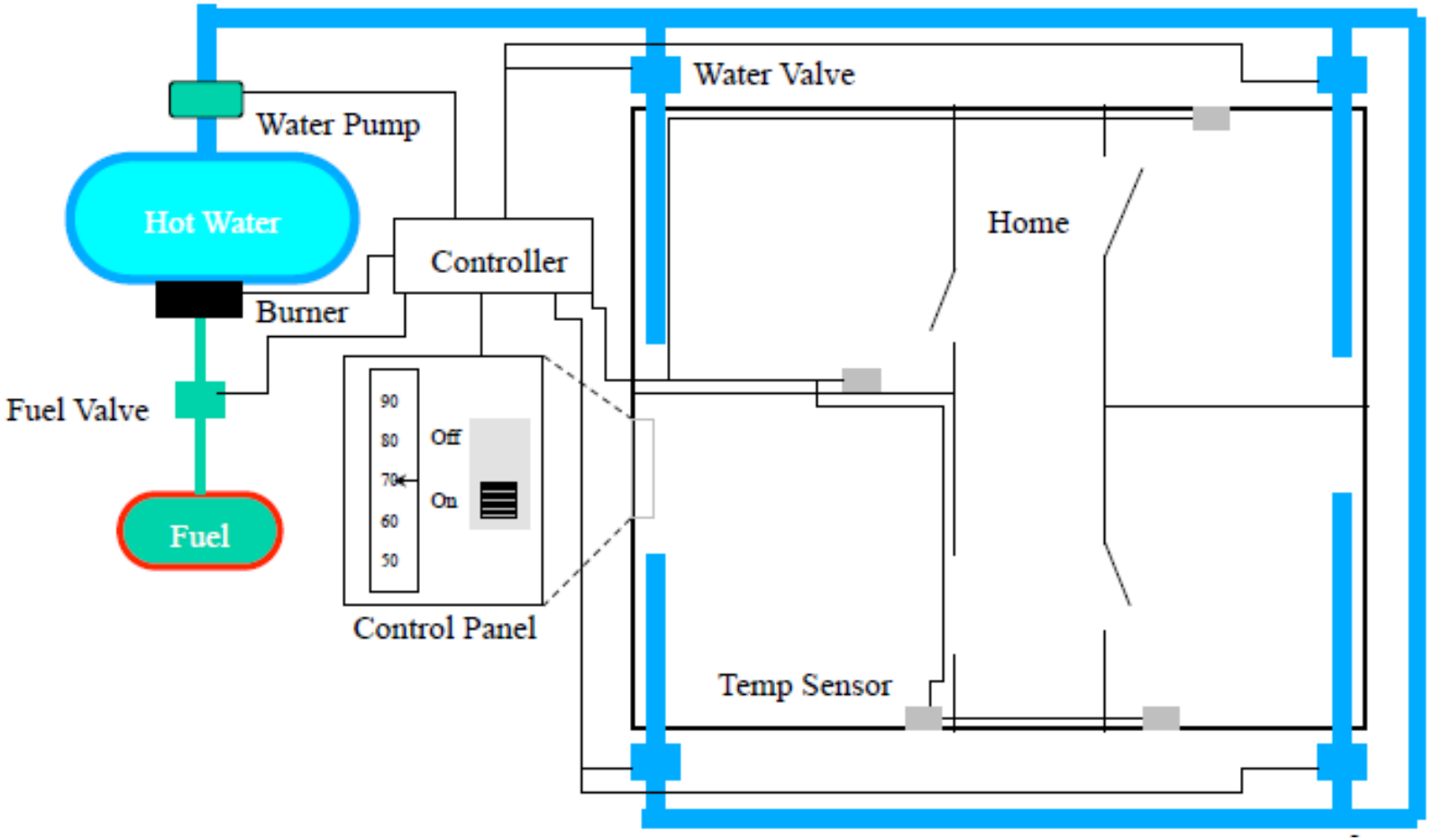
- Synchronous
- Asynchronous

IMPLICIT ASYNCHRONOUS COMMUNICATION SOFTWARE ARCHITECTURE

IMPLICIT ASYNCHRONOUS COMMUNICATION SOFTWARE ARCHITECTURE

Instead of invoking a procedure directly

- A **component** can announce (or broadcast) one or more events.
- When an event is announced, the broadcasting system (**connector**) itself invokes all of the procedures that have been registered for the event.


Water Valve
Water Pump
Hot Water
Controller
Home
Burner
Fuel Valve
90
80
70
60
50
Off
On
Fuel
Control Panel
Temp Sensor

IMPLICIT ASYNCHRONOUS COMMUNICATION SOFTWARE ARCHITECTURE

- Non-buffered Event-Based Implicit Invocations
- Buffered Message-Based Software Architecture

NON-BUFFERED EVENT-BASED IMPLICIT INVOCATIONS

NON-BUFFERED EVENT-BASED IMPLICIT INVOCATIONS

The non-buffered event-based implicit invocation architecture breaks the software system into two partitions:

- Event sources and
- Event listeners.

- The event registration process connects these two partitions. There is no buffer available between these two parties.

NON-BUFFERED EVENT-BASED IMPLICIT INVOCATIONS

- In the event-based implicit invocations (non-buffered) each object keeps its own dependency list.
- Any state changes of the object will impact its dependents.

BENEFITS:

- Reusability of components: It is easy to plug in new event handlers without affecting the rest of the system.
- System maintenance and evolution: Both event sources and targets are easy to update.
- Parallel execution of event handlings is possible.

LIMITATIONS:

- It is difficult to test and debug the system since it is hard to predict and verify responses and the order of responses from the listeners.
  - The event trigger cannot determine when a response has finished or the sequence of all responses.

- There is tighter coupling between event sources and their listeners than in message queue-based or message topic-based implicit invocation.

BUFFERED MESSAGE-BASED SOFTWARE ARCHITECTURE

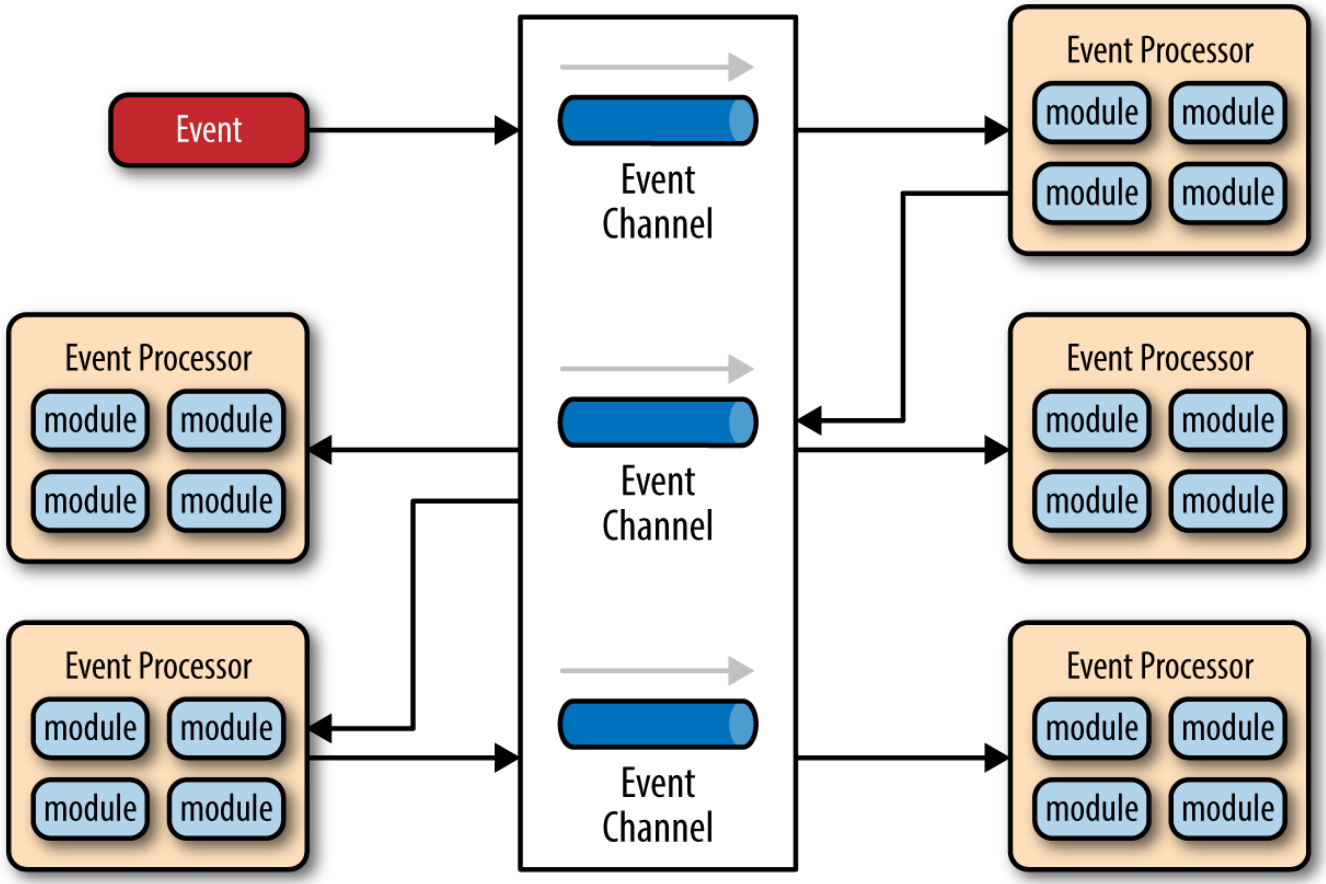
THE BUFFERED MESSAGE-BASED SOFTWARE ARCHITECTURE

It breaks the software system into three partitions:

- message producers,
- Message consumers, and
- message service providers.

They are connected asynchronously by either a message queue or a message topic.

THE BUFFERED MESSAGE-BASED SOFTWARE ARCHITECTURE



THE BUFFERED MESSAGE-BASED SOFTWARE ARCHITECTURE

- A messaging client can produce and send messages to other clients, and can also consume messages from other clients.

- Each client must register with a messaging destination in a connection session provided by a message service provider for creating, sending, receiving, reading, validating, and processing messages.

APPLICABLE DOMAINS OF MESSAGE-BASED ARCHITECTURE:

- Suitable for a software system where the communication between a producer and a receiver requires buffered message-based asynchronous implicit invocation for performance and distribution purposes.

- The provider wants components that function independently of information about other component interfaces so that components can be easily replaced.

BENEFITS:

- Anonymity: provides high degree of anonymity between message producer and consumer.
- Concurrency: supports concurrency both among consumers and between producer and consumers.
- Scalability

BENEFITS

- Provides strong support for **reuse** since any component can be introduced into a system simply by registering it for the events of that system.

- **Eases system evolution** since components may be replaced by other components without affecting the interfaces of other components in the system.

LIMITATIONS:

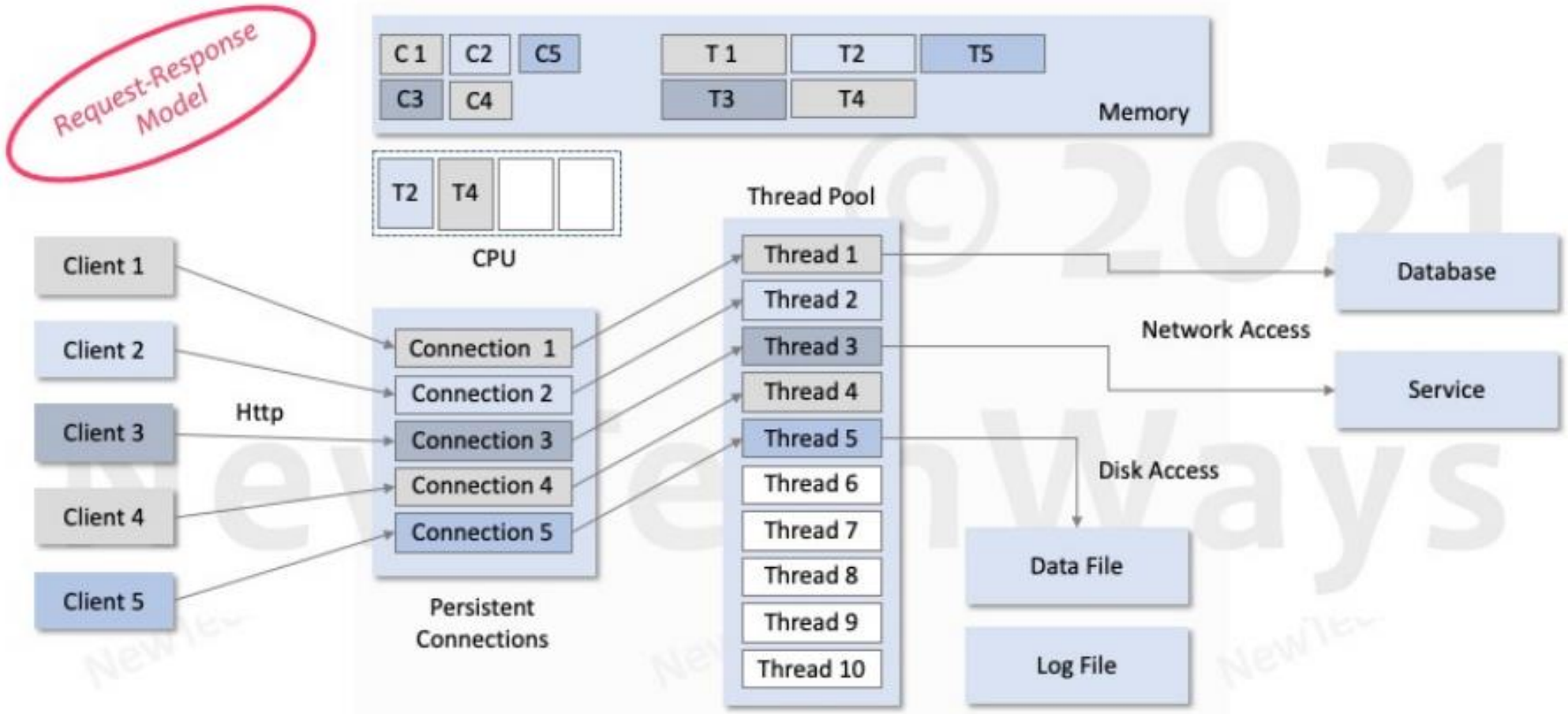
- Capacity limit of message queue:
- Increased complexity of the system design and implementation.

LIMITATIONS

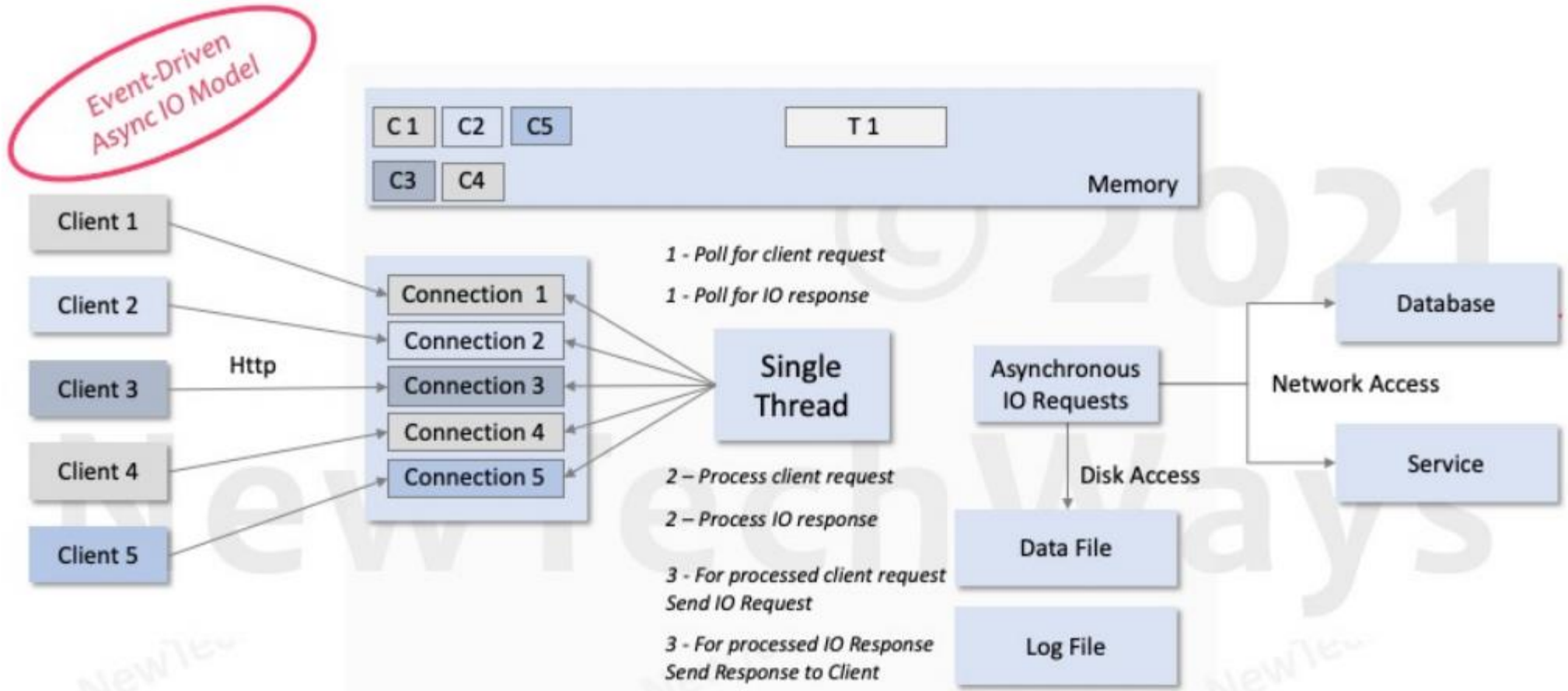
When a component announces an event:

- it has no idea how other components will respond to it,
- it cannot rely on the order in which the responses are invoked
- it cannot know when responses are finished.

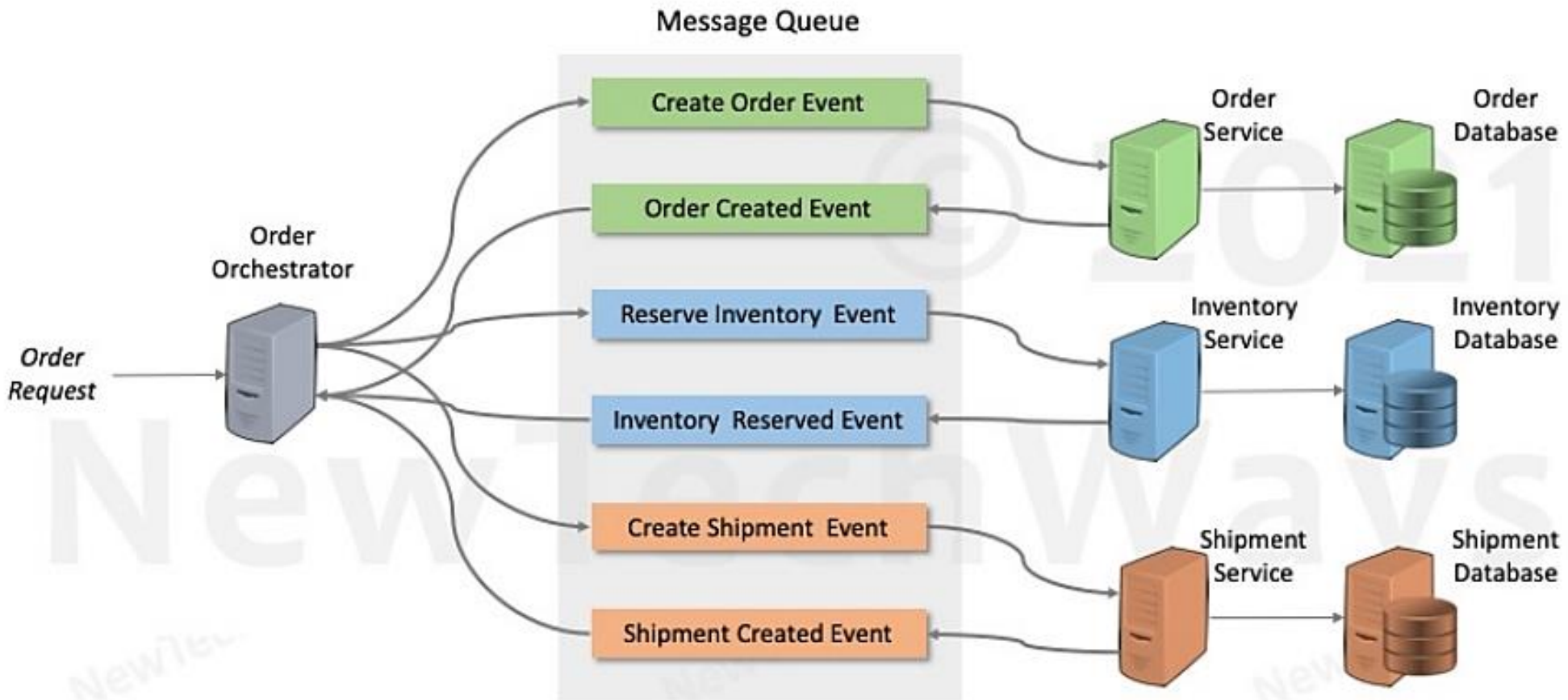
Apache Webserver Architecture

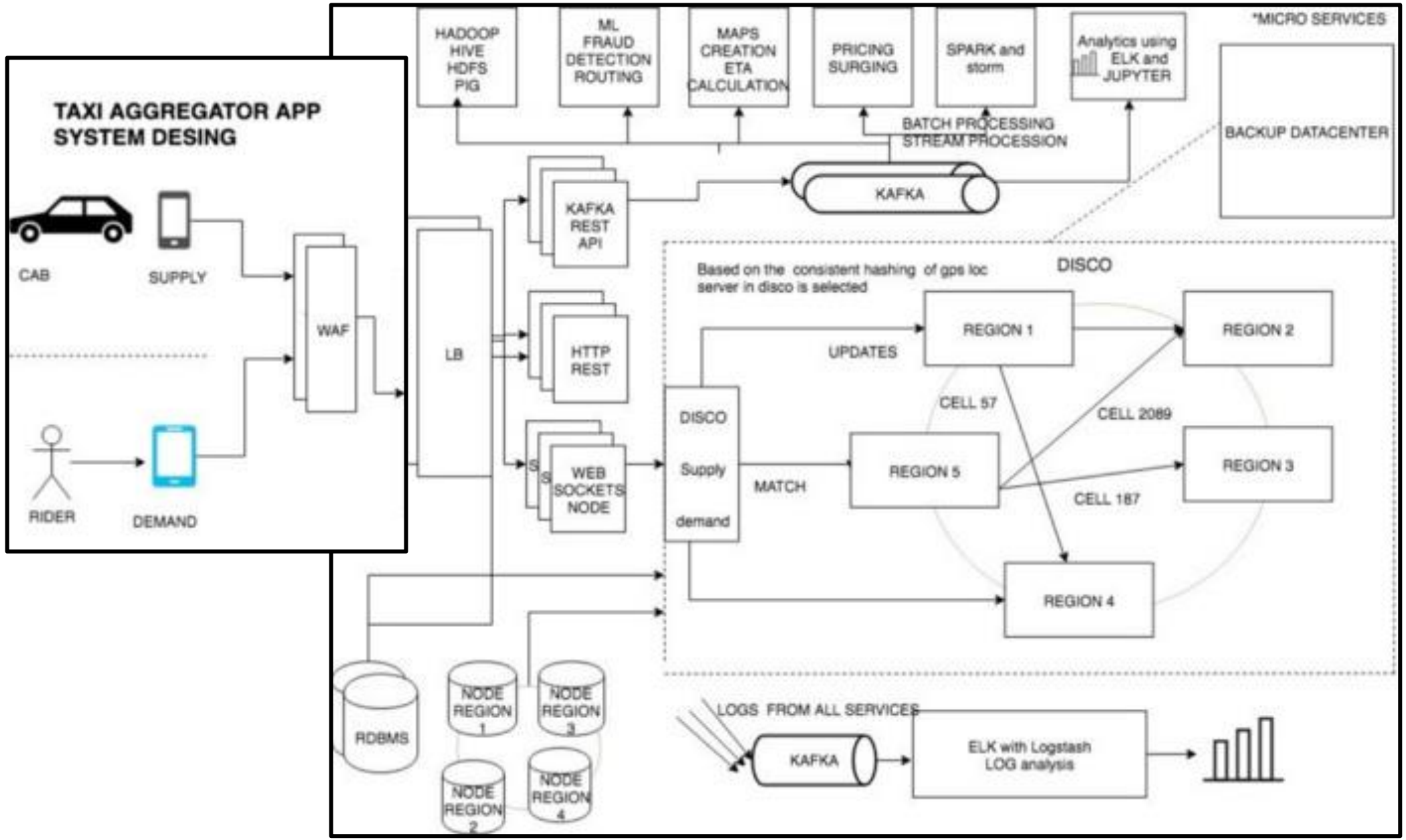


Nginx Architecture



Micro-Services Event Driven Transactions




TAXI AGGREGATOR APP
SYSTEM DESING
CAB
SUPPLY
RIDER
DEMAND
WAF
LB
HADOOP
HIVE
HDFS
PIG
ML
FRAUD
DETECTION
ROUTING
MAPS
CREATION
ETA
CALCULATION
PRICING
SURGING
SPARK and
storm
Analytics using
ELK and
JUPYTER
*MICRO SERVICES
BACKUP DATACENTER
BATCH PROCESSING
STREAM PROCESSION
KAFKA
KAFKA
REST
API
HTTP
REST
WEB
SOCKETS
NODE
DISCO
Based on the consistent hashing of gps loc
server in disco is selected
DISCO
Supply
demand
MATCH
UPDATES
REGION 1
REGION 2
REGION 3
REGION 4
REGION 5
CELL 57
CELL 2089
CELL 187
RDBMS
NODE
REGION
1
NODE
REGION
3
NODE
REGION
2
NODE
REGION
4
LOGS FROM ALL SERVICES
KAFKA
ELK with Logstash
LOG analysis

HAVE A GOOD DAY!