

# SOFTWARE ENGINEERING

## (Week-10)

USAMA MUSHARAF

MS-CS (Software Engineering)

*LECTURER (Department of Computer Science)*

*FAST-NUCES PESHAWAR*

# AGENDA OF WEEK #10

Architectural Styles (Cont..)

Event based Software Architecture

Distributed Software Architecture

Client Server

SOA

# CATEGORIES OF ARCHITECTURAL STYLES

## **Hierarchical Software Architecture**

Layered

## **Data Flow Software Architecture**

Pipe n Filter

Batch Sequential

## **Distributed Software Architecture**

Client Server

REST

SOA

Microservices

## **Event Based Software Architecture**

## **Data Centered Software Architecture**

Black board

Shared Repository

## **Component-Based Software Architecture**



# 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).

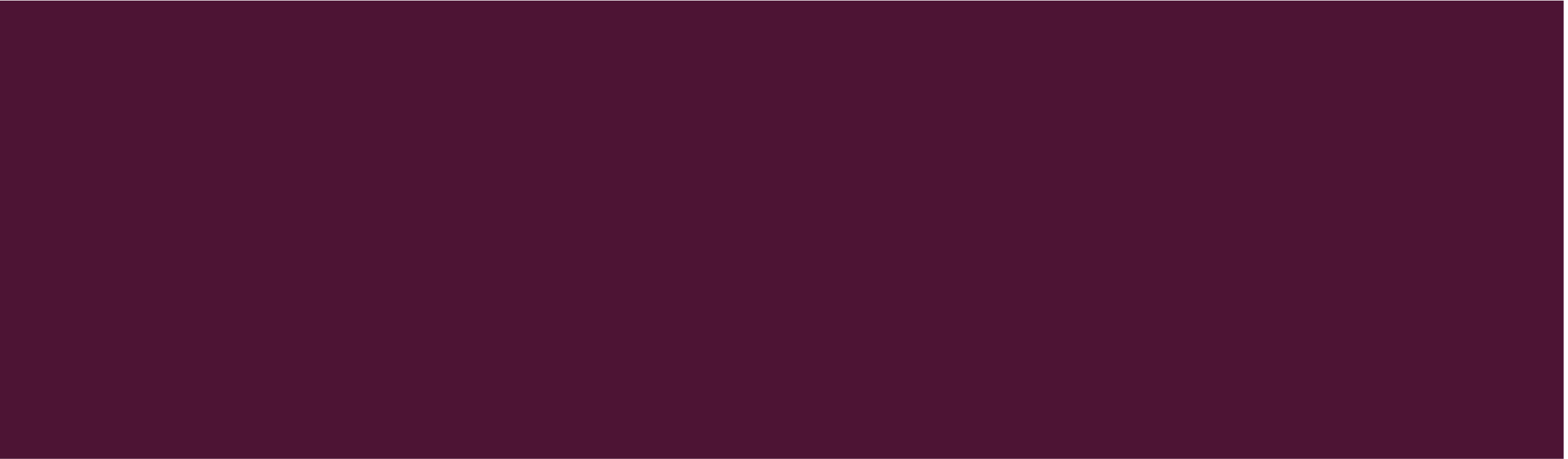
# SYNCHRONOUS VS ASYNCHRONOUS

Synchronous

Asynchronous



# IMPLICIT ASYNCHRONOUS COMMUNICATION SOFTWARE ARCHITECTURE



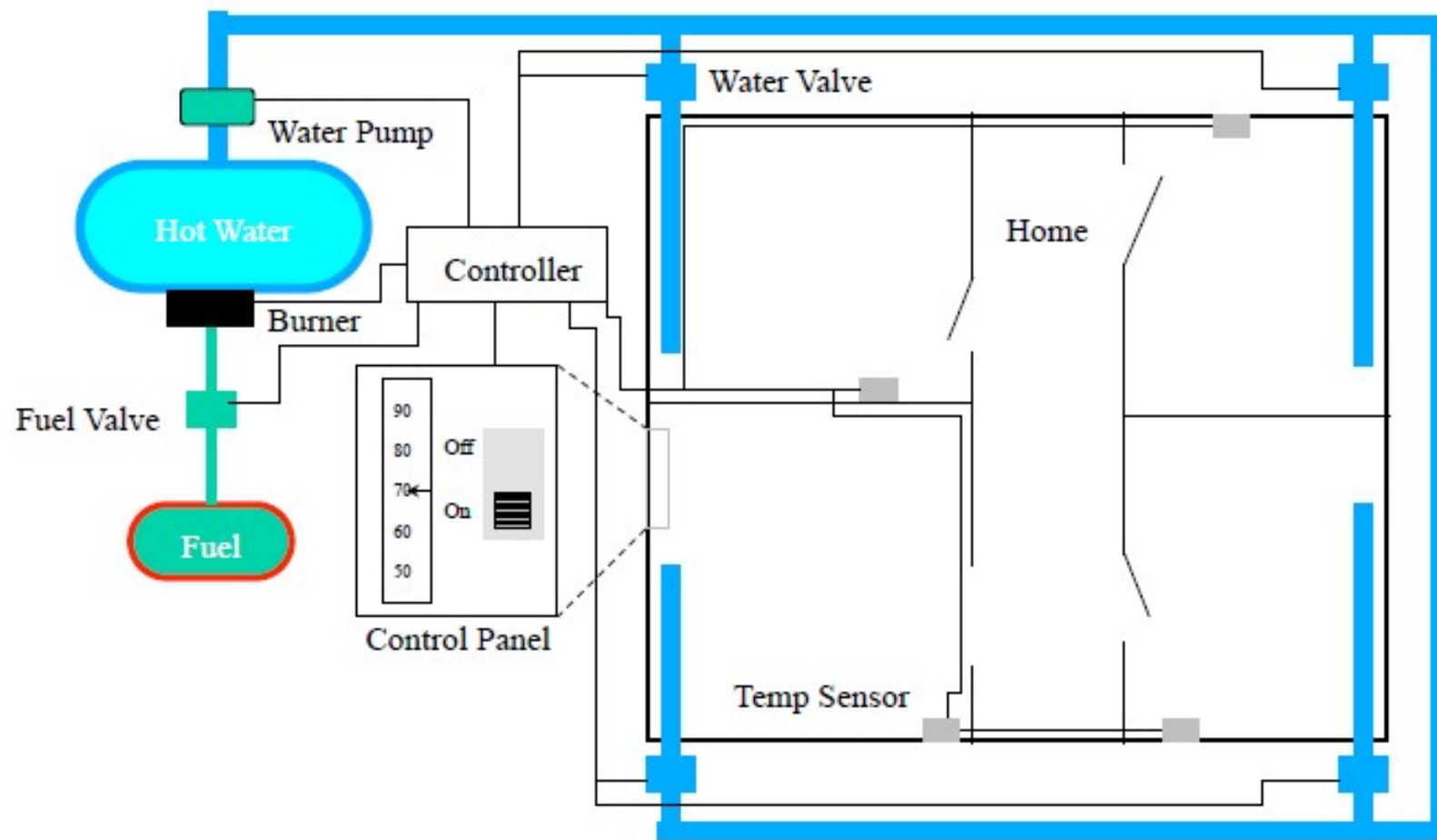


# IMPLICIT ASYNCHRONOUS COMMUNICATION SOFTWARE ARCHITECTURE

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.





# 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.

Reliability and overhead of indirect invocations may be an issue.





# BUFFERED MESSAGE-BASED SOFTWARE ARCHITECTURE



# THE BUFFERED MESSAGE-BASED SOFTWARE ARCHITECTURE

It breaks the software system into three partitions:

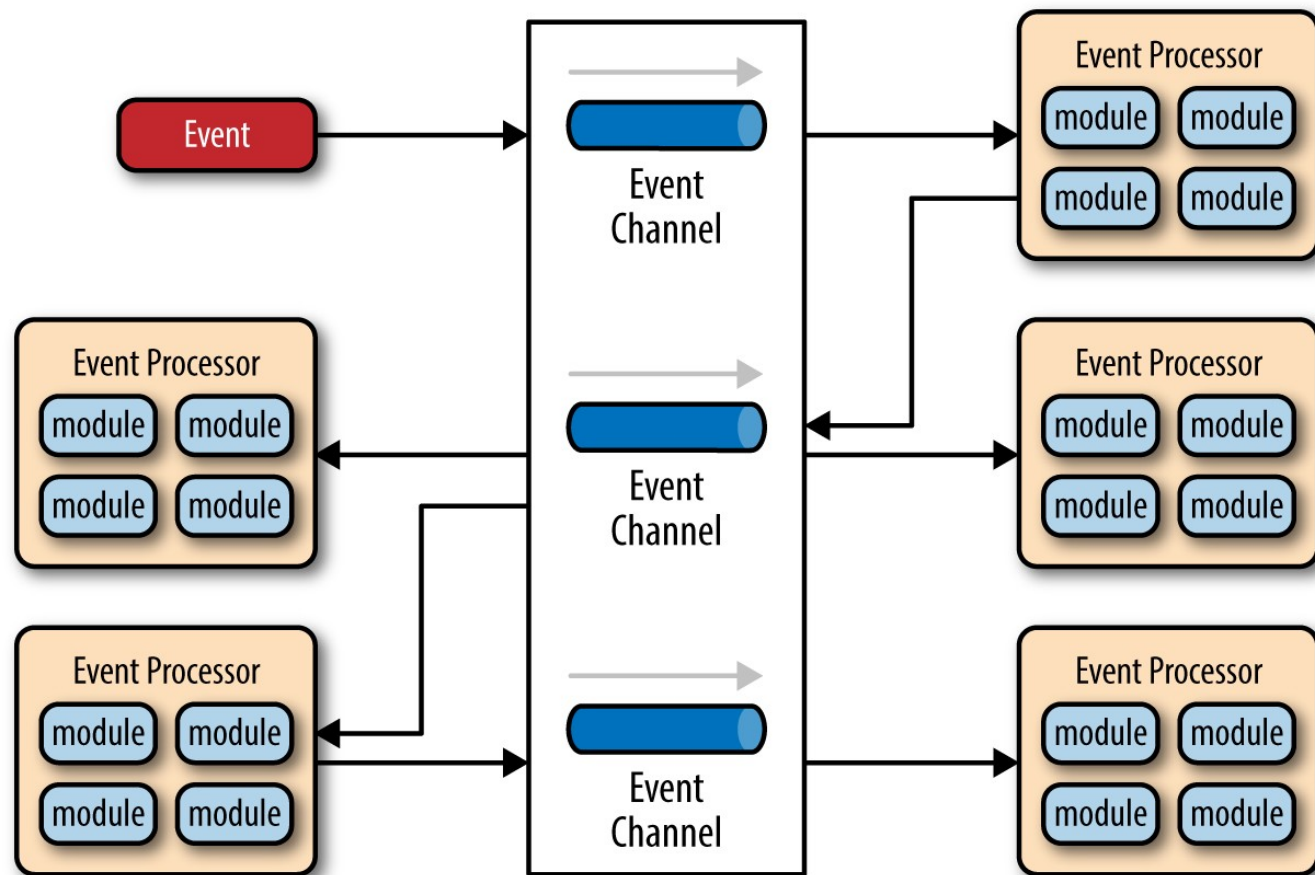
- message producers,

- 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.



# 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.

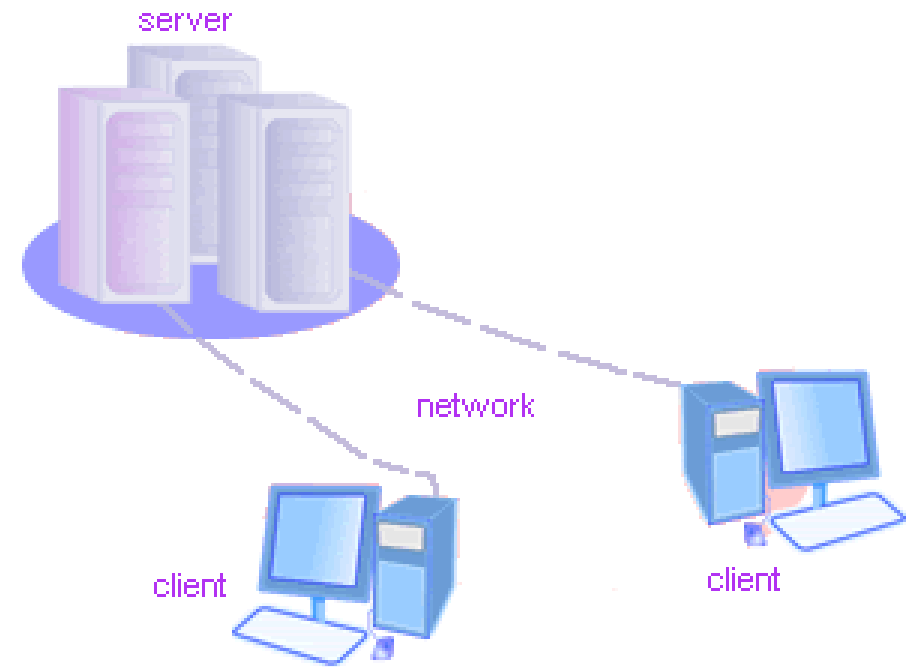
**Client** makes a service request to server.

**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.

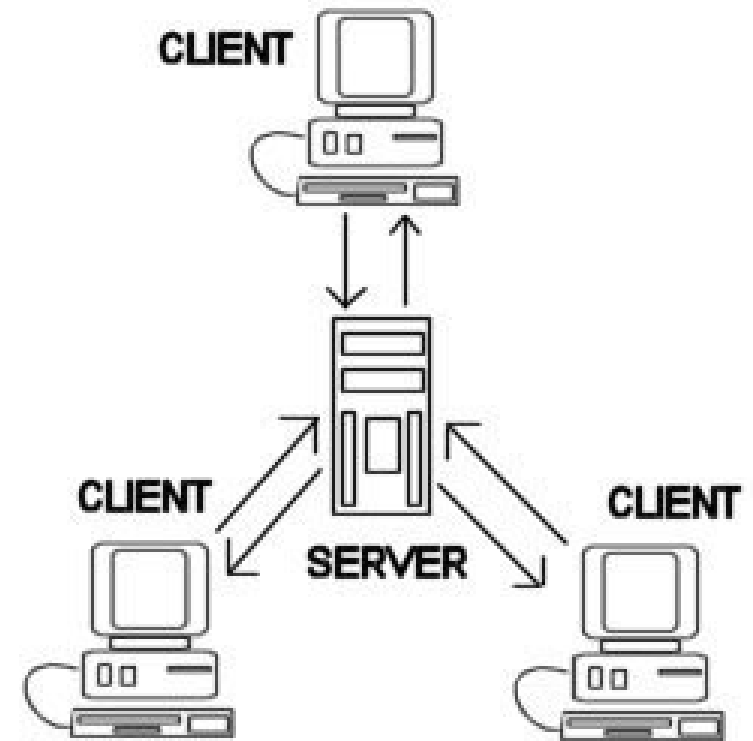
**Clients:** Components that call on the services provided by servers.

**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:

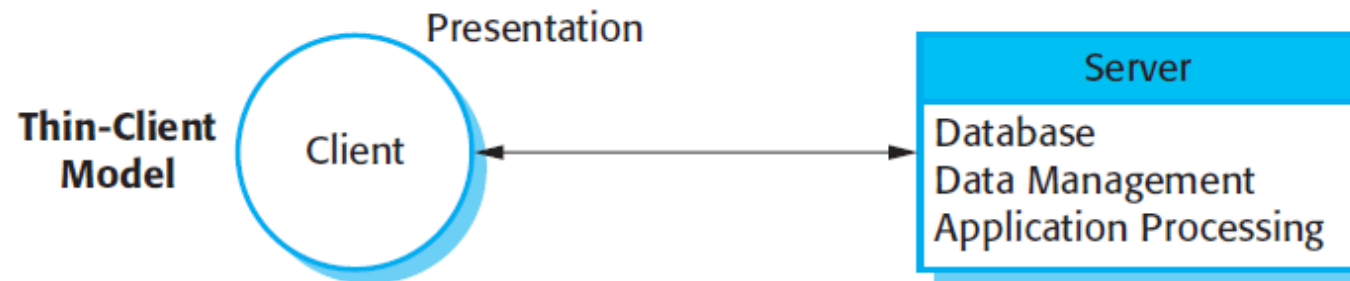
- A thin-client model,

- 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.

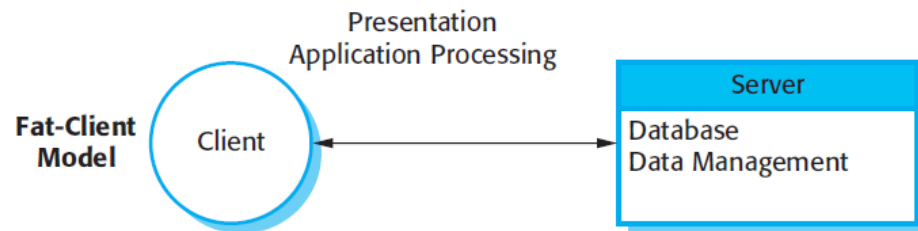


# TWO-TIER CLIENT SERVER

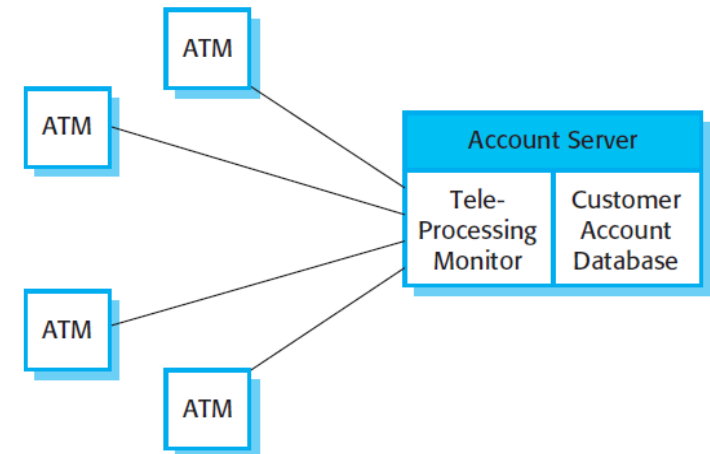
A fat-client model,

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



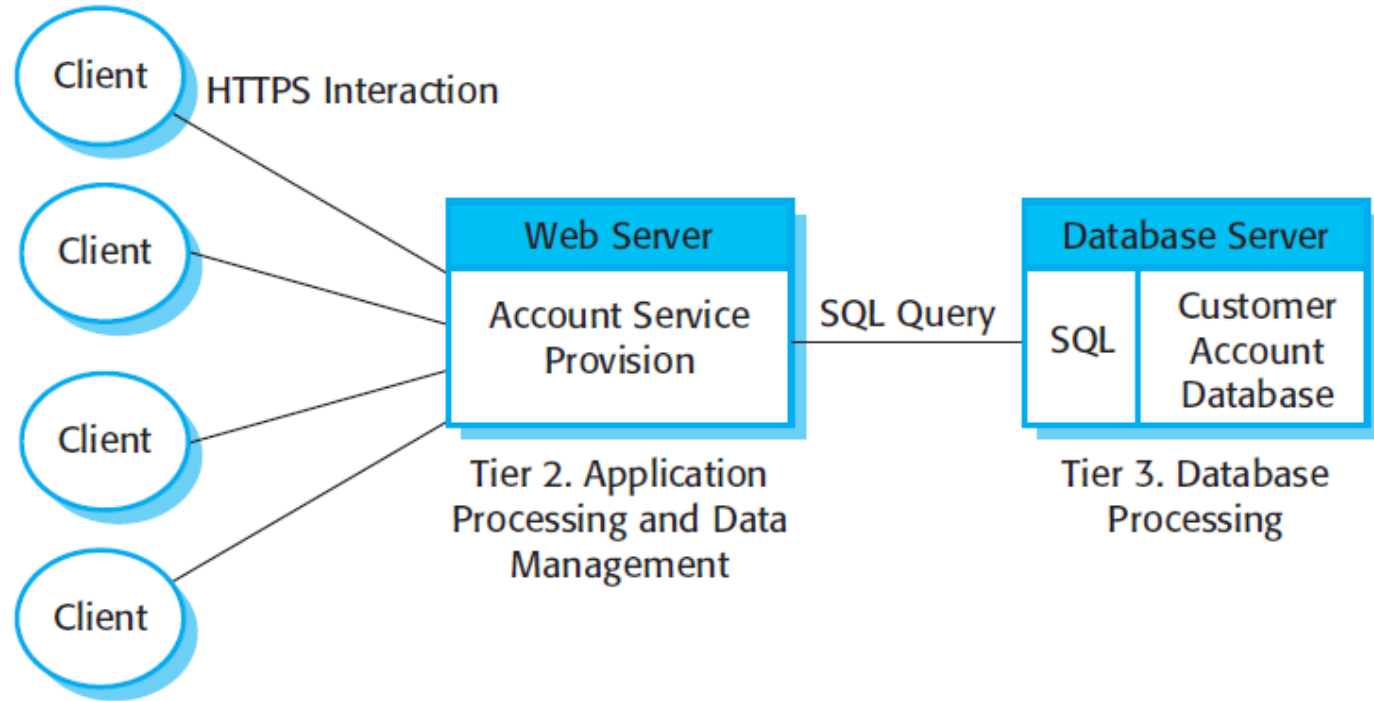
# MULTI-TIER CLIENT-SERVER ARCHITECTURES

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

Tier 1. Presentation



**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.



# SERVERLESS COMPUTING



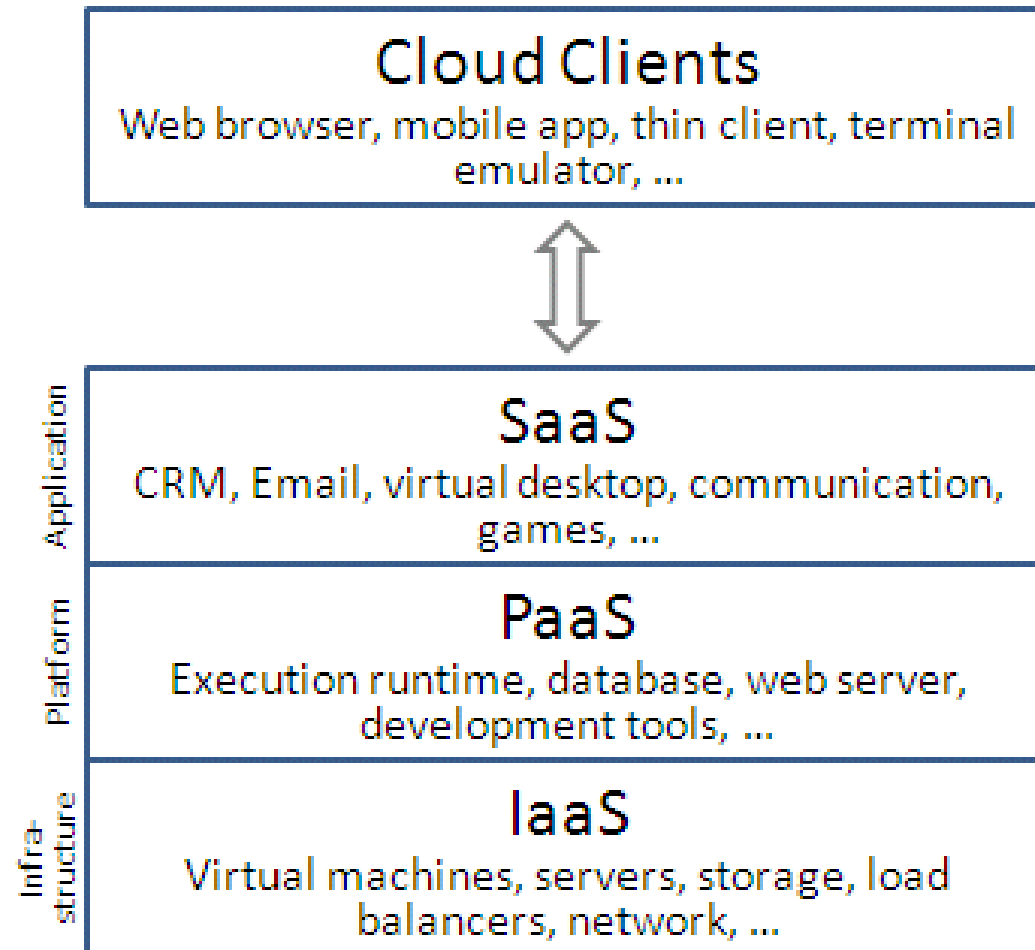
# SERVERLESS COMPUTING

Serverless computing is a cloud computing execution model in which the cloud provider allocates machine resources on demand, taking care of the servers on behalf of their customers.

# CLOUD COMPUTING

Cloud computing is the on-demand availability of computer system resources, especially data storage and computing power, without direct active management by the user.

# SERVICE MODELS







# 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

Account
<ul style="list-style-type: none"><li>• Balance</li><li>• Withdraw</li><li>• Deposit</li></ul>

## 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

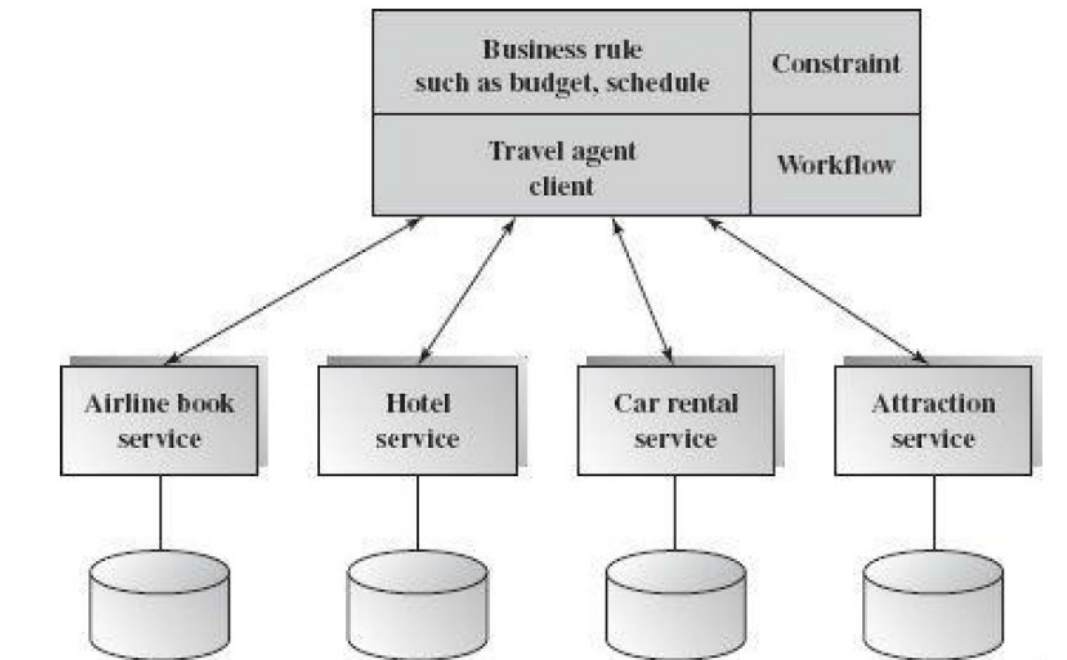
that all banking applications can reuse the service as and when necessary.

# EXAMPLE

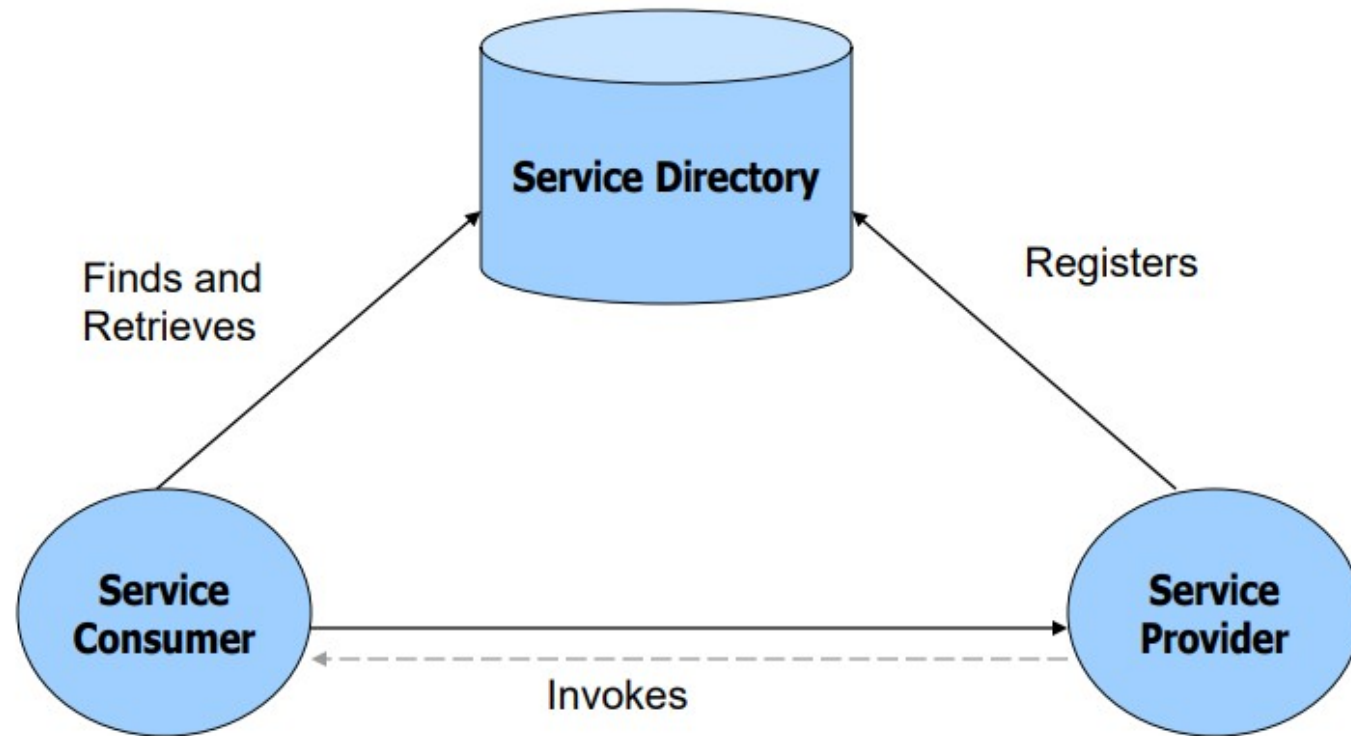
An online travel agency system

It consists of four existing web services:

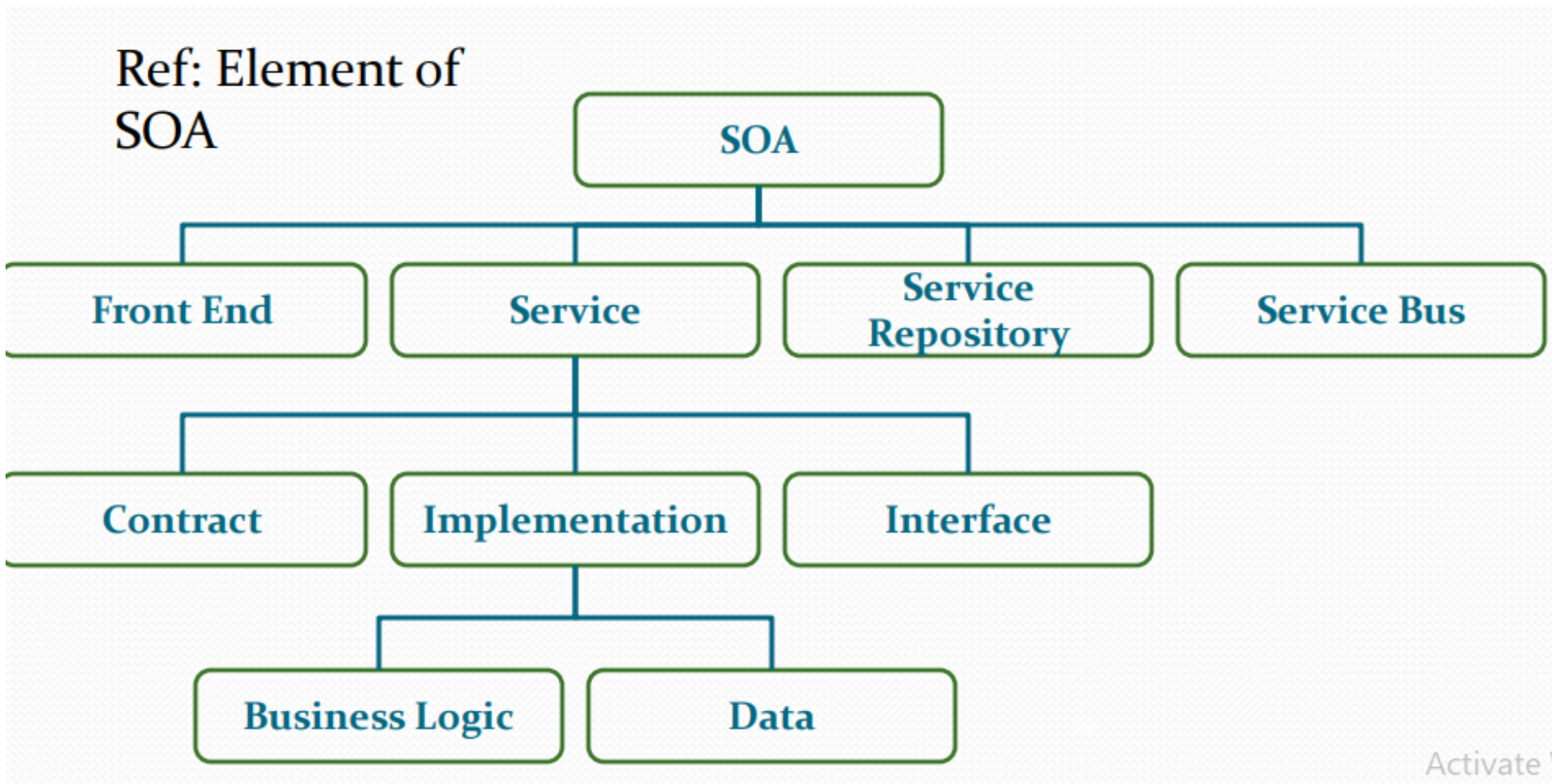
airline reservation,  
car rental,  
hotel reservation,  
and attraction reservation.



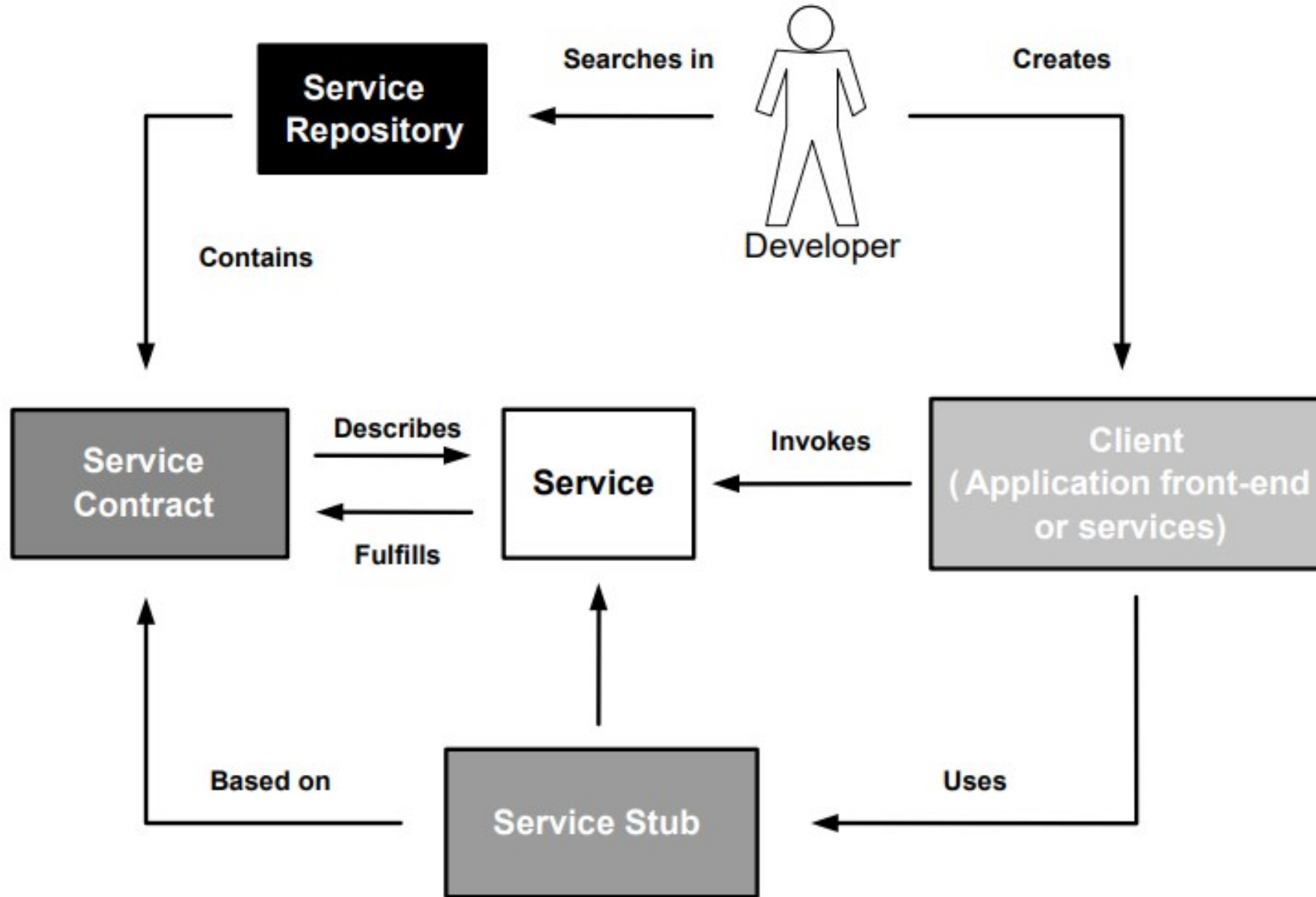
# SOA ARCHITECTURE



# COMPONENTS OF SOA

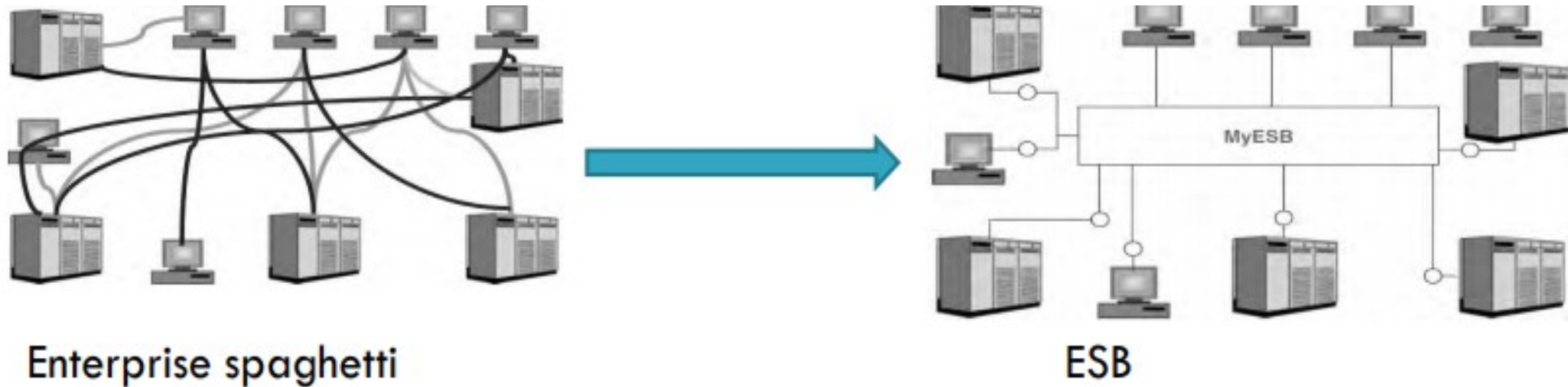






# ESB

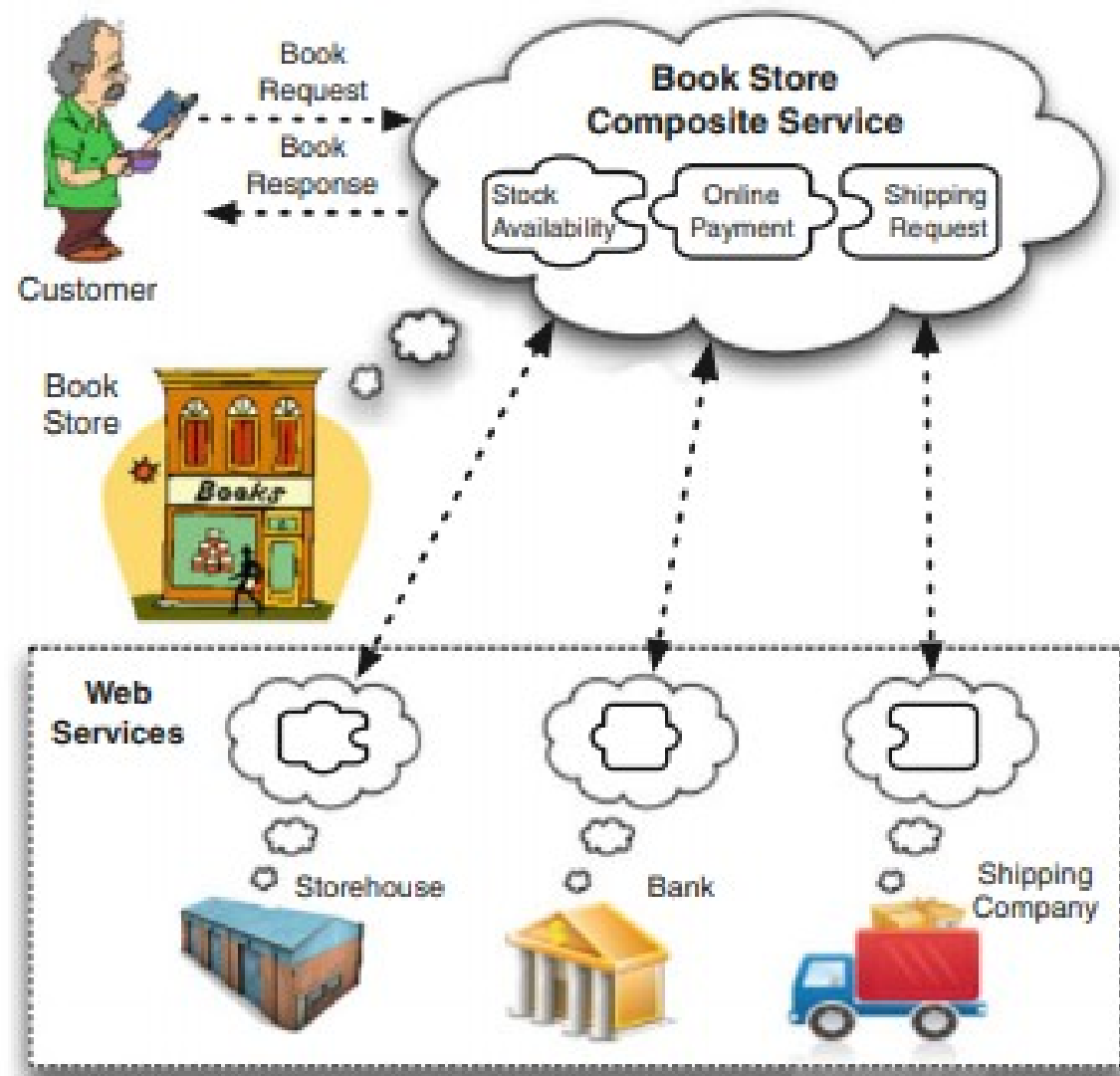
Enterprise Service Buses (ESBs) build on MOM (message-oriented middleware) to provide a flexible, scalable, standards-based integration technology for building a loosely coupled, highly-distributed SOA





# SERVICE COMPOSITION



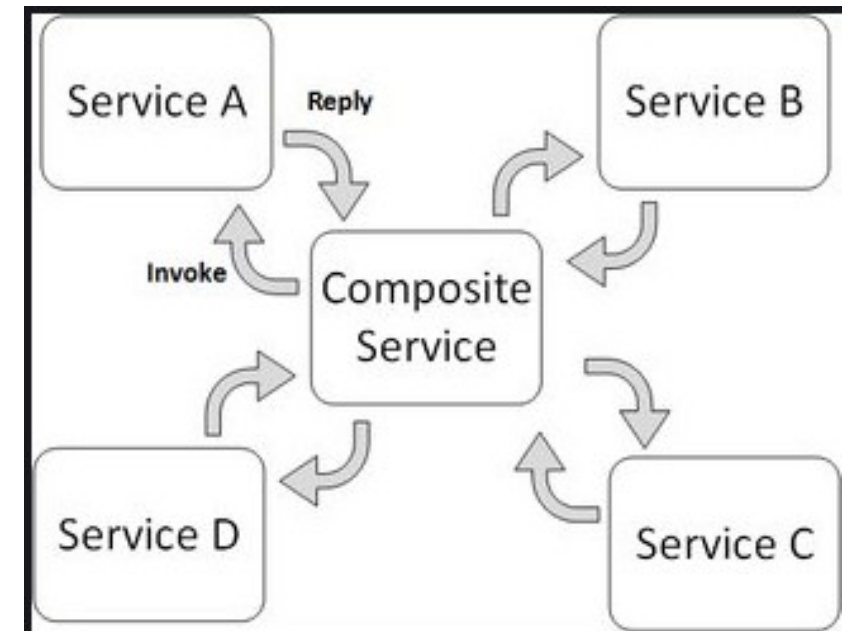


An example of Web service composition

# SERVICE ORCHESTRATION VS SERVICE CHOREOGRAPHY

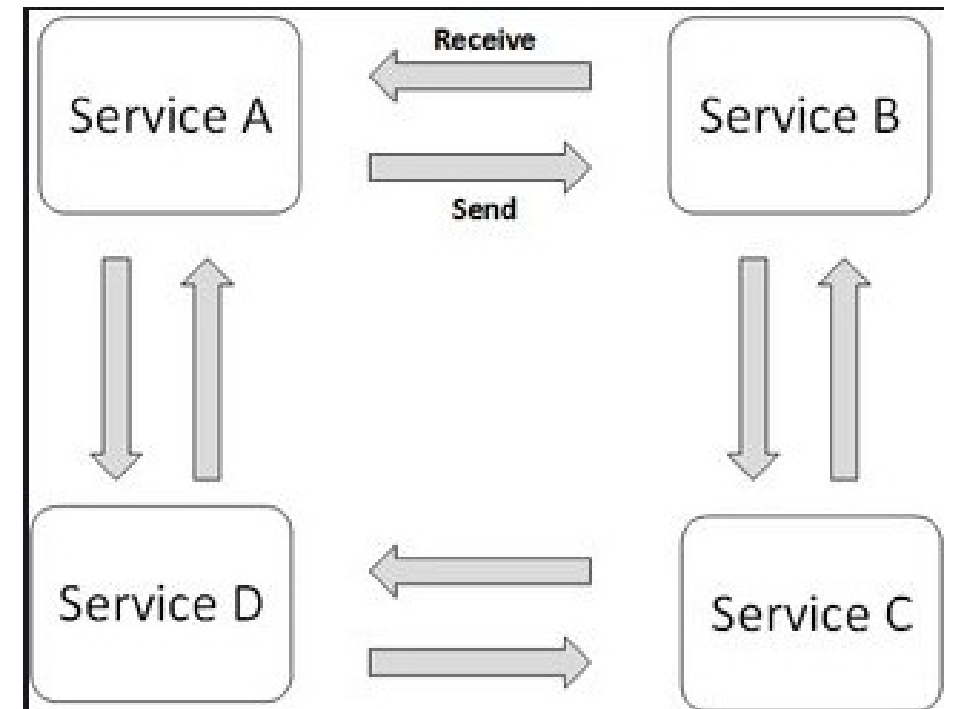
Orchestration comes from orchestras; in an orchestra there is a conductor leading it, and there are musicians who play their instruments following the instructions of the conductor.

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.





**HAVE A GOOD DAY!**