Unit-1
SOA (Service Oriented Architecture) Principles:

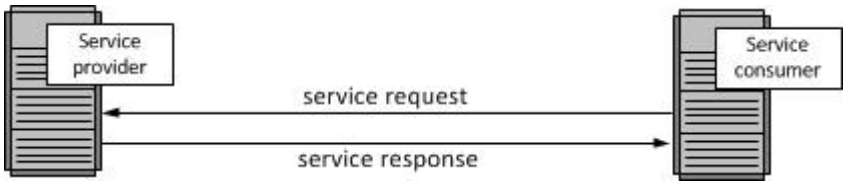
- A service-oriented architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network. The principles of service-orientation are independent of any product, vendor or technology.
- What is Service? A service is a well-defined, self-contained function that represents a unit of functionality. A service can exchange information from another service. It is not dependent on the state of another service. It uses a loosely coupled, message-based communication model to communicate with applications and other services.
- SOA just makes it easier for software components over various networks to work with each other.
- Web services which are built as per the SOA architecture tend to make web service more independent. The web services themselves can exchange data with each other and because of the underlying principles on which they are created, they don't need any sort of human interaction and also don't need any code modifications. It ensures that the web services on a network can interact with each other seamlessly.

- A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed.

- Service-oriented architectures are not a new thing. The first service-oriented architecture for many people in the past was with the use DCOM or Object Request Brokers (ORBs) based on the CORBA specification. For more on DCOM and CORBA.

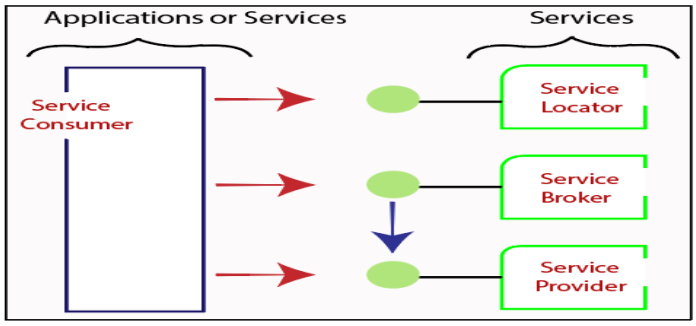
- **Services:**If a service-oriented architecture is to be effective, we need a clear understanding of the term service. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services.

- **Connections:The technology of Web Services is the most likely connection technology of service-oriented architectures. The following figure illustrates a basic service-oriented architecture. It shows a service consumer at the right sending a service request message to a service provider at the left. The service provider returns a response message to the service consumer. The request and subsequent response connections are defined in some way that is understandable to both the service consumer and service provider.**



Service-Oriented Terminologies:

Let's see some important service-oriented terminologies:



- **Services -** The services are the logical entities defined by one or more published interfaces.
- **Service provider -** It is a software entity that implements a service specification.
- **Service consumer -** It can be called as a requestor or client that calls a service provider. A service consumer can be another service or an end-user application.
- **Service locator -** It is a service provider that acts as a registry. It is responsible for examining service provider interfaces and service locations.
- **Service broker -** It is a service provider that pass service requests to one or more additional service providers.

SOA is based on some key principles which are mentioned below:

1. **Standardized Service Contract -** Services adhere to a service description. A service must have some sort of description which describes what the service is about. This makes it easier for client applications to understand what the service does.

2. **Loose Coupling** – Less dependency on each other. This is one of the main characteristics of web services which just states that there should be as less dependency as possible between the web services and the client invoking the web service. So if the service functionality changes at any point in time, it should not break the client application or stop it from working.

3. **Service Abstraction** - Services hide the logic they encapsulate from the outside world. The service should not expose how it executes its functionality; it should just tell the client application on what it does and not on how it does it.

4. **Service Reusability** - Logic is divided into services with the intent of maximizing reuse. In any development company re-usability is a big topic because obviously one wouldn't want to spend time and effort building the same code again and again across multiple applications which require them. Hence, once the code for a web service is written it should have the ability work with various application types.

5. **Service Autonomy** - Services should have control over the logic they encapsulate. The service knows everything on what functionality it offers and hence should also have complete control over the code it contains.

6. **Service Statelessness** - Ideally, services should be stateless. This means that services should not withhold information from one state to the other. This would need to be done from either the client application. An example can be an order placed on a shopping site. Now you can have a web service which gives you the price of a particular item. But if the items are added to a shopping cart and the web page navigates to the page where you do the payment, the responsibility of the price of the item to be transferred to the payment page should not be done by the web service. Instead, it needs to be done by the web application.

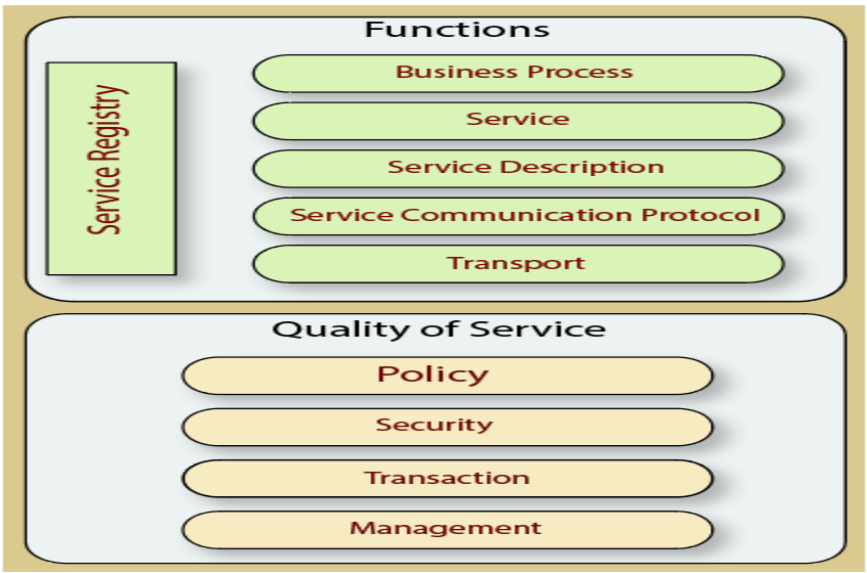
7. **Service Discoverability** - Services can be discovered (usually in a service registry). We have already seen this in the concept of the UDDI, which performs a registry which can hold information about the web service.

8. **Service Composability** - Services break big problems into little problems. One should never embed all functionality of an application into one single service but instead, break the service down into modules each with a separate business functionality.

9. **Service Interoperability** - Services should use standards that allow diverse subscribers to use the service. In web services, standards as XML and communication over HTTP is used to ensure it conforms to this principle.

Components of service-oriented architecture:

The service-oriented architecture stack can be categorized into two parts - functional aspects and quality of service aspects.



Functional aspects:

The functional aspect contains:

- **Transport -** It transports the service requests from the service consumer to the service provider and service responses from the service provider to the service consumer.
- **Service Communication Protocol** - It allows the service provider and the service consumer to communicate with each other.
- **Service Description** - It describes the service and data required to invoke it.
- Service - It is an actual service.
- **Business Process** - It represents the group of services called in a particular sequence associated with the particular rules to meet the business requirements.
- **Service Registry** - It contains the description of data which is used by service providers to publish their services.

Quality of Service aspects:

The quality of service aspects contains:

- **Policy** - It represents the set of protocols according to which a service provider make and provide the services to consumers.
- **Security** - It represents the set of protocols required for identification and authorization.
- **Transaction** - It provides the surety of consistent result. This means, if we use the group of services to complete a business function, either all must complete or none of the complete.
- **Management** - It defines the set of attributes used to manage the services.

Advantages of SOA:

SOA has the following advantages:

- **Easy to integrate** - In a service-oriented architecture, the integration is a service specification that provides implementation transparency.
- **Manage Complexity** - Due to service specification, the complexities get isolated, and integration becomes more manageable.
- **Platform Independence** - The services are platform-independent as they can communicate with other applications through a common language.
- **Loose coupling** - It facilitates to implement services without impacting other applications or services.
- **Parallel Development** - As SOA follows layer-based architecture, it provides parallel development.
- **Available** - The SOA services are easily available to any requester.
- **Reliable** - As services are small in size, it is easier to test and debug them.

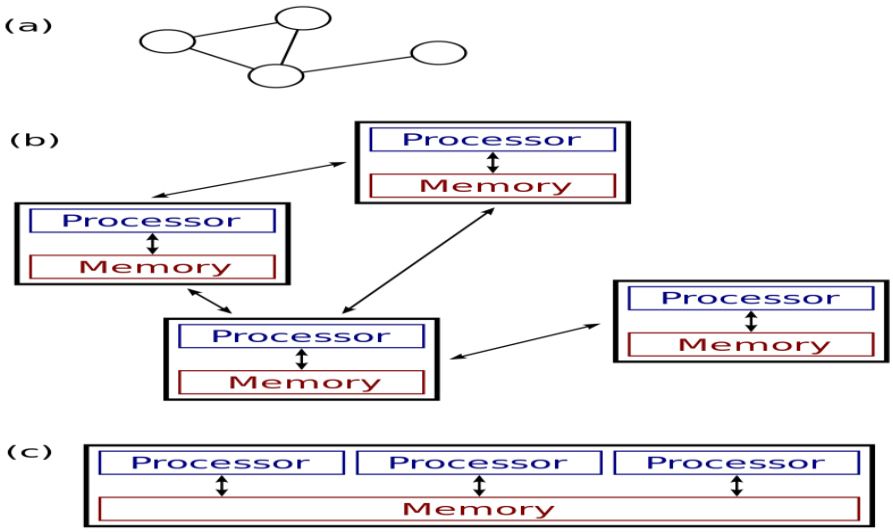
Distributed Computing?

- Distributed computing is a computing concept that, in its most general sense, refers to multiple computer systems working on a single problem. In distributed computing, a single problem is divided into many parts, and each part is solved by different computers. As long as the computers are networked, they can communicate with each other to solve the problem. If done properly, the computers perform like a single entity.
- **Distributed computing** is a field of computer science that studies distributed systems. A *distributed system* is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. The components interact with one another in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components. Examples of distributed systems vary from SOA-based systems to massively multiplayer online games to peer-to-peer applications.
- A computer program that runs within a distributed system is called a **distributed program** (and distributed programming is the process of writing such programs). There are many different types of implementations for the message passing mechanism, including pure HTTP, RPC-like connectors and message queues.
- Distributed computing also refers to the use of distributed systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other via message passing.
- The ultimate goal of distributed computing is to maximize performance by connecting users and IT resources in a cost-effective, transparent and reliable manner. It also ensures fault tolerance and enables resource accessibility in the event that one of the components fails.

Parallel and Distributed Computing:

Distributed systems are groups of networked computers, which have the same goal for their work. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them. The same system may be characterized both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel. Parallel computing may be seen as a particular tightly coupled form of distributed computing, and distributed computing may be seen as a loosely coupled form of parallel computing. Nevertheless, it is possible to roughly classify concurrent systems as "parallel" or "distributed" using the following criteria:

- In parallel computing, all processors may have access to a shared memory to exchange information between processors.
- In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.



The figure illustrates the difference between distributed and parallel systems. Figure (a) is a schematic view of a typical distributed system; the system is represented as a network topology in which each node is a computer and each line connecting the nodes is a communication link. Figure (b) shows the same distributed system in more detail: each computer has its own local memory, and information can be exchanged only by passing messages from one node to another by using the available communication links. Figure (c) shows a parallel system in which each processor has a direct access to a shared memory.

Distributed Computing in the large/ Basics for Big Data:

If your company is considering a big data project, it's important that you understand some distributed computing basics first. There isn't a single distributed computing model because computing resources can be distributed in many ways.

For example, you can distribute a set of programs on the same physical server and use messaging services to enable them to communicate and pass information. It is also possible to have many different systems or servers, each with its own memory, that can work together to solve one problem.

Why distributed computing is needed for big data

Not all problems require distributed computing. If a big time constraint doesn't exist, complex processing can done via a specialized service remotely. When companies needed to do complex data analysis, IT would move data to an external service or entity where lots of spare resources were available for processing.

It wasn't that companies wanted to wait to get the results they needed; it just wasn't economically feasible to buy enough computing resources to handle these emerging requirements. In many situations, organizations would capture only selections of data rather than try to capture all the data because of costs. Analysts wanted all the data but had to settle for snapshots, hoping to capture the right data at the right time.

Key hardware and software breakthroughs revolutionized the data management industry. First, innovation and demand increased the power and decreased the price of hardware. New software emerged that understood how to take advantage of this hardware by automating processes like load balancing and optimization across a huge cluster of nodes.

The software included built-in rules that understood that certain workloads required a certain performance level. The software treated all the nodes as though they were simply one big pool of computing, storage, and networking assets, and moved processes to another node without interruption if a node failed, using the technology of virtualization.

The changing economics of computing and big data

Fast-forward and a lot has changed. Over the last several years, the cost to purchase computing and storage resources has decreased dramatically. Aided by virtualization, commodity servers that could be clustered and blades that could be networked in a rack changed the economics of computing. This change coincided with innovation in software automation solutions that dramatically improved the manageability of these systems.

The capability to leverage distributed computing and parallel processing techniques dramatically transformed the landscape and dramatically reduce latency. There are special cases, such as High Frequency Trading (HFT), in which low latency can only be achieved by physically locating servers in a single location.

The problem with latency for big data

One of the perennial problems with managing data — especially large quantities of data — has been the impact of latency. *Latency* is the delay within a system based on delays in execution of a task. Latency is an issue in every aspect of computing, including communications, data management, system performance, and more.

If you have ever used a wireless phone, you have experienced latency firsthand. It is the delay in the transmissions between you and your caller. At times, latency has little impact on customer satisfaction, such as if companies need to analyze results behind the scenes to plan for a new product release. This probably doesn't require instant response or access.

However, the closer that response is to a customer at the time of decision, the more that latency matters.

Note:Distributed computing and parallel processing techniques can make a significant difference in the latency experienced by customers, suppliers, and partners. Many big data applications are dependent on low latency because of the big data requirements for speed and the volume and variety of the data.

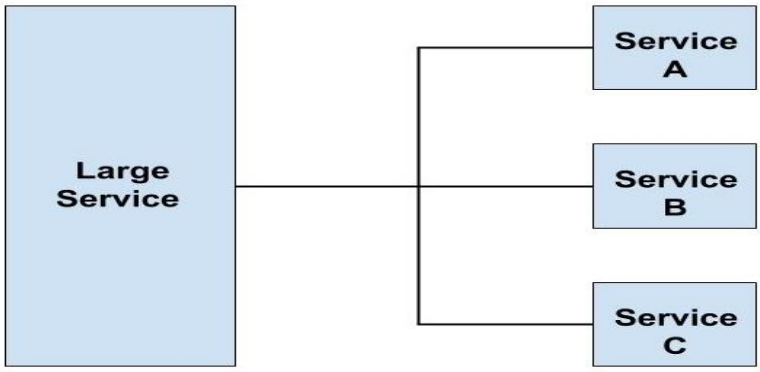
It may not be possible to construct a big data application in a high latency environment if high performance is needed. The need to verify the data in near real time can also be impacted by latency.

When you are dealing with real-time data, a high level of latency means the difference between success and failure.

Motivations for Composition of SOA(Service Oriented Architecture):

Service composition is a collection of services where, many smaller services are combined together to a larger service.

Below diagram illustrates the service composition:



- In the above diagram, Service A, Service B and Service C are smaller services.
- Large service is composed by combining services A,B and C together.

Service Composition Performance

The services communicate with each other through a network just like component composition where inter-service communication is too slow as compared to inter-component communication taking place in the same application. The performance will be bad if the services communicate internally through ESB (Enterprise Service Bus) and larger services are decomposed to many smaller services.

Service compositions can be categorized into primitive and complex variations. Simple logic was implemented through point-to-point exchanges or primitive compositions in early service-oriented solutions. As the technology developed, complex compositions became more familiar.

Challenges for Composition of SOA(Service Oriented Architecture):

- **Categorize and document services during the design phase.** It's important to identify service categories (and subcategories) as a means of establishing a framework for service definition, design and development. This also ensures that reviewers can quickly understand the scope of service design that is being undertaken. We often encourage our clients to identify their data services by subject area (e.g., customer, product, etc.) or functional capability (cleanse, augment, match, etc.). Processing services are the most common initial services to be developed. It's common to differentiate high-level (or business) services (Invoicing, Purchasing, etc.) from lower level (or functional) services (lookup/search, retrieving records, etc.).
- **Differentiate data from processing services.** It's important that data services are establishes defined separate from their processing services brethren. We often find folks implementing services based on the calling conventions and interfaces of an individual software development project. The end result tends to reflect a set of application-centric functional conventions, not services that simplify application communication and functionality development. There's rarely any attention paid to defining services that relate to retrieving, defining or modifying business

data. Without well defined data services, your SOA environment will be hampered in its ability to support different business areas or applications.

- **Identify a series of usage scenarios for each service.** Because of the visibility and buzz that web services and SOA receive from the media (and even internal to most companies), it's important to prevent the SOA development effort from being perceived as a "field of silver bullets and dreams." All too often, we find that services are designed, developed and delivered to an underwhelmed audience. The disappointment isn't caused by the technical staff diverging from the project plan or goal; the problem occurs because the folks relying in the individual services didn't fully understand the scope of functionality that was going to be delivered. The value of the usage scenarios is that it dramatically reduces the likelihood that there will be any misunderstanding regarding the capabilities of the SOA components.

- **Leverage existing data management rigor.** Since the premise of SOA is to establish a shared data and processing infrastructure, it's important that everyone that utilizes the services, either from a development or a usage perspective, understands the details. We're not talking about the internal workings of the service; we're talking about the inputs, outputs and processing results of the service. If data is manipulated within the service — like a zip code lookup — or delivered — like providing customer address details — it's important that the data and the processes are referred to using terms that are already in use within your company. If everyone uses zip code, calling a service a PostalCodeLookup is a bad idea.

As we know the benefits of SOA are well documented, but the challenges associated with an enterprise wide SOA deployment are not. While the distributed nature of a **SOA** encourages reuse and provides a high level of agility for the business, it can also give rise to real challenges in the delivery of SOA-based applications.

XML is the core technology enabler of SOA-based applications. Its verbose nature, inherent lack of security, and the increase in connections between applications and services required result in a number of challenges that are well-understood, and that have proven solutions.

The challenge in delivering SOA-based applications lies in identifying where potential problems will arise and addressing them as early in the deployment cycle as possible. Application delivery controllers are well suited to addressing both the well-understood and unanticipated issues associated with delivering SOA-based applications. Incorporating an application delivery **controller** (ADC) into the deployment plans for your SOA can prevent unnecessary delays and even the need to re-architect portions of your SOA infrastructure--saving time, expense, and headaches.

SOA is an architectural design pattern and a deployment methodology, a nice way of saying that no one can define what any particular SOA implementation should look like. While there have been attempts by various vendors in a number of vertical SOA markets to create best practices for SOA environments, these have largely failed due to the rather dynamic and organizational-specific nature of SOA implementations. Because the services that comprise an SOA are business-focused and encapsulate business specific entities, there is no single agreed upon definition of what must exist and how components are deployed in order to bear the title SOA.

There are, however, a set of guiding principles underlying SOA that can be considered a framework around which discussions on common SOA attributes and challenges can be based. **The distributed nature of services, for example, is a fundamental attribute of all SOA implementations and as such challenges associated with that deployment model can be applied to all SOA environments, regardless of their actual design or implementation**.

1. An SOA is comprised of distributed business services and achieves business value through the reuse of those services.
2. An SOA is based upon industry accepted open standards.
3. An SOA reduces time to market through loose coupling of service interfaces and its underlying implementation, enabling agility.
4. An SOA is heterogeneous and applications comprise n-tiers that may vary widely based on the business process activity around which its composite services are built.

These common attributes result in a common set of challenges that all organizations face and that must be addressed at some point in the implementation process. SOA guiding principles suggest that consideration be given to the **architecture** of the deployment environment and **infrastructure** upon which services will be deployed and delivered before said services are put into production, in parallel with service definition and development efforts.

Fortunately, most of the challenges associated with deploying SOA are common to all web-based application deployments. The difference between SOA-based applications and traditional applications is that these challenges are often faced by implementers earlier in the application lifecycle due to the challenges associated with its core technology enabler, XML.

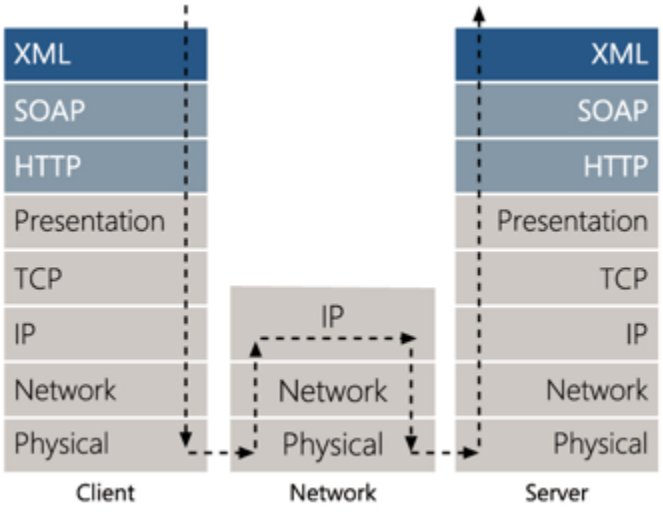
SOA basics

In most cases, an SOA will be implemented using a variety of standards, the most common being HTTP, WSDL (Web Services Definition Language), **SOAP** (Simple **Object** Access Protocol), and XML (eXtensible Markup Language). These latter three standards work together to deliver messages between services much in the same way as the post office.

WSDL--Entry in an address book HTTP--Postal carrier (transportation) SOAP--Envelope (encapsulation) XML--Letter (message)

If you're sending a letter then it is assumed you already know where to send it. In a SOA that information comes from the WSDL, which lists the transportation options (HTTP, FTP, SMTP) available, the possible addressees (remote functions available), and the **format** requirements for data sent to each addressee (XML).

The same way an envelope has an address, SOAP carries with it information about where the message should be delivered and who should open it. In the case of SOAP and SOA, the addressee is not a who, but a what, as the addressee is really the name of a remote **function** that should process the message inside. Just like a "real" envelope, SOAP has specific formatting requirements and failure to properly place information in the right place can cause delivery or **processing** issues.
With this information you can build a message based on the format requirements (XML-based), put it in an envelope address to the receiver (SOAP), and then drop it in the mailbox (HTTP) and wait for a response.
If you are the receiver of this message, the message must be delivered to you, you have to read (parse) the envelope and determine the correct addressee (remote function). The message then must be passed to the appropriate addressee (remote function) for processing. Once the correct addressee has the letter, they can then read (parse) the message and process it according to their designated function.



Web Services Architectures and Standards:

Modern day business applications use variety of programming platforms to develop web-based applications. Some applications may be developed in Java, others in .Net, while some other in Angular JS, Node.js, etc.

Most often than not, these heterogeneous applications need some sort of communication to happen between them. Since they are built using different development languages, it becomes really difficult to ensure accurate communication between applications.

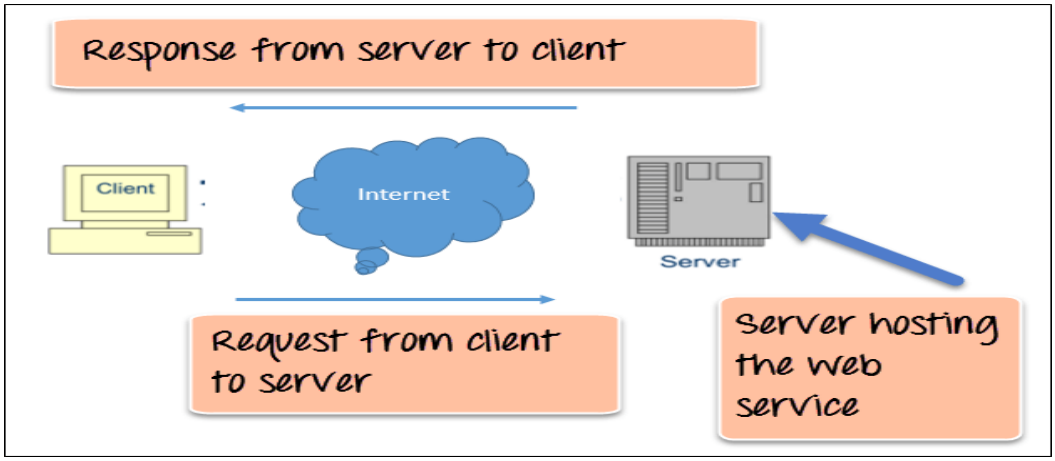
Here is where web services come in. Web services provide a common platform that allows multiple applications built on various programming languages to have the ability to communicate with each other.Here we will explain more about what web services are about, the different elements which constitute web services, and a little bit about SOA (Service Oriented Architecture) principles.

What is Web Service?

Web service is a standardized medium to propagate communication between the client and server applications on the World Wide Web.

A web service is a software module which is designed to perform a certain set of tasks.

- The web services can be searched for over the network and can also be invoked accordingly.
- When invoked the web service would be able to provide functionality to the client which invokes that web service.



(Web Service Architecture Diagram)

The above diagram shows a very simplistic view of how a web service would actually work. The client would invoke a series of web service calls via requests to a server which would host the actual web service.

These requests are made through what is known as remote procedure calls. Remote Procedure Calls(RPC) are calls made to methods which are hosted by the relevant web service.

As an example, Amazon provides a web service that provides prices for products sold online via amazon.com. The front end or presentation layer can be in .Net or Java but either programming language would have the ability to communicate with the web service.

The main component of a web service is the data which is transferred between the client and the server, and that is XML. XML (Extensible markup language) is a counterpart to HTML and easy to understand the intermediate language that is understood by many programming languages.

So when applications talk to each other, they actually talk in XML. This provides a common platform for application developed in various programming languages to talk to each other.

Web services use something known as SOAP (Simple Object Access Protocol) for sending the XML data between applications. The data is sent over normal HTTP. The data which is sent from the web service to the application is called a SOAP message. The SOAP message is nothing but

an XML document. Since the document is written in XML, the client application calling the web service can be written in any programming language.

Type of Web Service

There are mainly two types of web services.

1.SOAP web services.

2.RESTful web services.

In order for a web service to be fully functional, there are certain components that need to be in place. These components need to be present irrespective of whatever development language is used for programming the web service.

Let's look at these components in more detail.

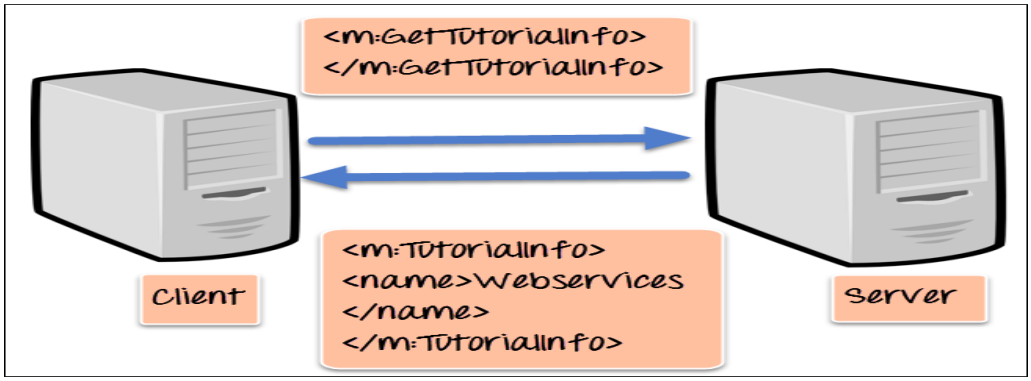
SOAP (Simple Object Access Protocol)

SOAP is known as a transport-independent messaging protocol. SOAP is based on transferring XML data as SOAP Messages. Each message has something which is known as an XML document. Only the structure of the XML document follows a specific pattern, but not the content. The best part of Web services and SOAP is that its all sent via HTTP, which is the standard web protocol.

Here is what a SOAP message consists of

- Each SOAP document needs to have a root element known as the <Envelope> element. The root element is the first element in an XML document.
- The "envelope" is in turn divided into 2 parts. The first is the header, and the next is the body.
- The header contains the routing data which is basically the information which tells the XML document to which client it needs to be sent to.
- The body will contain the actual message.

The diagram below shows a simple example of the communication via SOAP.



WSDL (Web services description language)

A web service cannot be used if it cannot be found. The client invoking the web service should know where the web service actually resides.

Secondly, the client application needs to know what the web service actually does, so that it can invoke the right web service. This is done with the help of the WSDL, known as the Web services description language. The WSDL file is again an XML-based file which basically tells the client application what the web service does. By using the WSDL document, the client application would be able to understand where the web service is located and how it can be utilized.

Web Service Example

An example of a WSDL file is given below.

```
<definitions>
  <message name="TutorialRequest">
    <part name="TutorialID" type="xsd:string"/>
  </message>

  <message name="TutorialResponse">
    <part name="TutorialName" type="xsd:string"/>
  </message>

  <portType name="Tutorial_PortType">
    <operation name="Tutorial">
      <input message="tns:TutorialRequest"/>
      <output message="tns:TutorialResponse"/>
    </operation>
  </portType>

  <binding name="Tutorial_Binding" type="tns:Tutorial_PortType">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="Tutorial">
      <soap:operation soapAction="Tutorial"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:Tutorialservice"
          use="encoded"/>
      </input>

            <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:Tutorialservice"
          use="encoded"/>
      </output>
    </operation>
```

```
  </binding>
</definitions>
```

The important aspects to note about the above WSDL declaration are as follows;

1. **<message>** - The message parameter in the WSDL definition is used to define the different data elements for each operation performed by the web service. So in the example above, we have 2 messages which can be exchanged between the web service and the client application, one is the "TutorialRequest", and the other is the "TutorialResponse" operation. The TutorialRequest contains an element called "TutorialID" which is of the type string. Similarly, the TutorialResponse operation contains an element called "TutorialName" which is also a type string.
2. **<portType>** - This actually describes the operation which can be performed by the web service, which in our case is called Tutorial. This operation can take 2 messages; one is an input message, and the other is the output message.
3. **<binding>** - This element contains the protocol which is used. So in our case, we are defining it to use http (**http://schemas.xmlsoap.org/soap/http**). We also specify other details for the body of the operation, like the namespace and whether the message should be encoded.

Universal Description, Discovery, and Integration (UDDI)

UDDI is a standard for describing, publishing, and discovering the web services that are provided by a particular service provider. It provides a specification which helps in hosting the information on web services.

Now we discussed in the previous topic about WSDL and how it contains information on what the Web service actually does. But how can a client application locate a WSDL file to understand the various operations offered by a web service? So UDDI is the answer to this and provides a repository on which WSDL files can be hosted. So the client application will have complete access to the UDDI, which acts as a database containing all the WSDL files.

Just as a telephone directory has the name, address and telephone number of a particular person, the same way the UDDI registry will have the relevant information for the web service. So that a client application knows, where it can be found.

Web Services Advantages

We already understand why web services came about in the first place, which was to provide a platform which could allow different applications to talk to each other.

But let's look at some other advantages of why it is important to use web services.

1.Exposing Business Functionality on the network - A web service is a unit of managed code that provides some sort of functionality to client applications or end users. This functionality can be invoked over the HTTP protocol which means that it can also be invoked over the internet. Nowadays all applications are on the internet which makes the purpose of Web services more useful. That means the web service can be anywhere on the internet and provide the necessary functionality as required.

2.Interoperability amongst applications - Web services allow various applications to talk to each other and share data and services among themselves. All types of applications can talk to each other. So instead of writing specific code which can only be understood by specific applications, you can now write generic code that can be understood by all applications

3.A Standardized Protocol which everybody understands - Web services use standardized industry protocol for the communication. All the four layers (Service Transport, XML Messaging, Service Description, and Service Discovery layers) uses well-defined protocols in the web services protocol stack.

4.Reduction in cost of communication - Web services use SOAP over HTTP protocol, so you can use your existing low-cost internet for implementing web services.

Web service Architecture

Every framework needs some sort of architecture to make sure the entire framework works as desired. Similarly, in web services, there is an architecture which consists of three distinct roles as given below

1.Provider - The provider creates the web service and makes it available to client application who want to use it.

2.Requestor - A requestor is nothing but the client application that needs to contact a web service. The client application can be a .Net, Java, or any other language based application which looks for some sort of functionality via a web service.

3.Broker - The broker is nothing but the application which provides access to the UDDI. The UDDI, as discussed in the earlier topic enables the client application to locate the web service.

The diagram below showcases how the Service provider, the Service requestor and Service registry interact with each other.

1.Publish - A provider informs the broker (service registry) about the existence of the web service by using the broker's publish interface to make the service accessible to clients

2.Find - The requestor consults the broker to locate a published web service

3.Bind - With the information it gained from the broker(service registry) about the web service, the requestor is able to bind, or invoke, the web service.

Web service Characteristics

Web services have the following special behavioral characteristics:

1.They are XML-Based - Web Services uses XML to represent the data at the representation and data transportation layers. Using XML eliminates any networking, operating system, or platform sort of dependency since XML is the common language understood by all.

2.Loosely Coupled – Loosely coupled means that the client and the web service are not bound to each other, which means that even if the web service changes over time, it should not change the way the client calls the web service. Adopting a loosely coupled architecture tends to make software systems more manageable and allows simpler integration between different systems.

3.Synchronous or Asynchronous functionality- Synchronicity refers to the binding of the client to the execution of the service. In synchronous operations, the client will actually wait for the web service to complete an operation. An example of this is probably a scenario wherein a database read and write operation are being performed. If data is read from one database and subsequently written to another, then the operations have to be done in a sequential manner. Asynchronous operations allow a client to invoke a service and then execute other functions in parallel. This is one of the common and probably the most preferred techniques for ensuring that other services are not stopped when a particular operation is being carried out.

4.Ability to support Remote Procedure Calls (RPCs) - Web services enable clients to invoke procedures, functions, and methods on remote objects using an XML-based protocol. Remote procedures expose input and output parameters that a web service must support.

5.Supports Document Exchange - One of the key benefits of XML is its generic way of representing not only data but also complex documents. These documents can be as simple as representing a current address, or they can be as complex as representing an entire book.

Computing with Services:

Services Computing has become a cross-discipline that covers the science and technology of bridging the gap between business services and IT services. The underlying technology suite includes Web services and service-oriented architecture (SOA), cloud computing, business consulting methodology and utilities, business process

modeling, transformation and integration. This scope of Services Computing covers the whole life-cycle of service provision that includes business componentization, services modeling, services creation, services realization, services annotation, services deployment, services discovery, services composition, services delivery, service-to-service collaboration, services monitoring, services optimization, as well as services management. The goal of Services Computing is to enable IT services and computing technology to perform business services more efficiently and effectively.

World Wide Web Consortium(W3C):

The **World Wide Web Consortium** (**W3C**) is the main international standards organization for the World Wide Web (abbreviated WWW or W3). Founded and currently led by Tim Berners-Lee, the consortium is made up of member organizations which maintain full-time staff for the purpose of working together in the development of standards for the World Wide Web. As of 29 May 2019, the World Wide Web Consortium (W3C) has 444 members.The W3C also engages in education and outreach, develops software and serves as an open forum for discussion about the Web.

History Of W3C:

The World Wide Web Consortium (W3C) was founded by Tim Berners-Lee after he left the European Organization for Nuclear Research (CERN) in October, 1994. It was founded at the Massachusetts Institute of Technology Laboratory for Computer Science (MIT/LCS) with support from the European Commission and the Defense Advanced Research Projects Agency (DARPA), which had pioneered the ARPANET, one of the predecessors to the Internet. It was located in Technology Square until 2004, when it moved, with CSAIL, to the Stata Center.

The organization tries to foster compatibility and agreement among industry members in the adoption of new standards defined by the W3C. Incompatible versions of HTML are offered by different vendors, causing inconsistency in how web pages are displayed. The consortium tries to get all those vendors to implement a set of core principles and components which are chosen by the consortium.

It was originally intended that CERN host the European branch of W3C; however, CERN wished to focus on particle physics, not information technology. In April 1995, the French Institute for Research in Computer Science and Automation (INRIA) became the European host of W3C, with Keio University Research Institute at SFC (KRIS) becoming the Asian host in September 1996. Starting in 1997, W3C created regional offices around the world. As of September 2009, it had eighteen World Offices covering Australia, the Benelux countries (Netherlands, Luxembourg, and Belgium), Brazil, China, Finland, Germany, Austria, Greece, Hong Kong, Hungary, India, Israel, Italy, South Korea, Morocco, South Africa, Spain, Sweden, and, as of 2016, the United Kingdom and Ireland.[6]

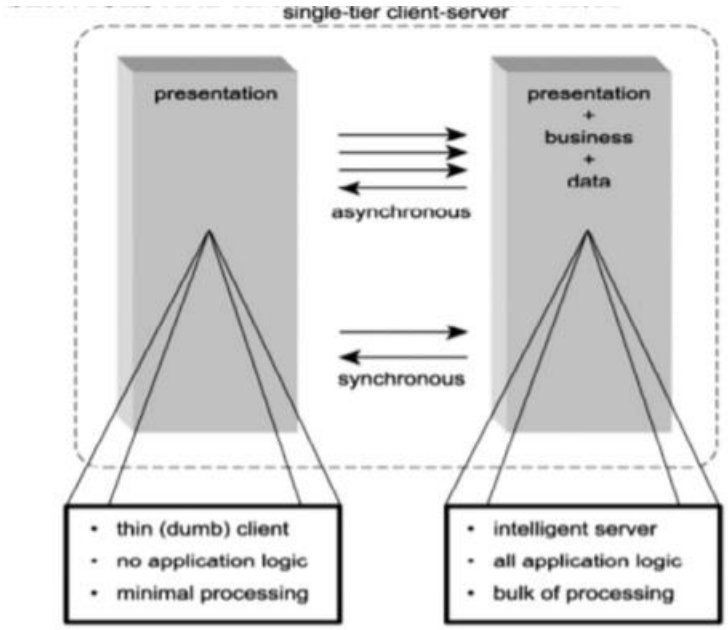
In October 2012, W3C convened a community of major web players and publishers to establish a MediaWiki wiki that seeks to document open web standards called the WebPlatform and WebPlatform Docs.

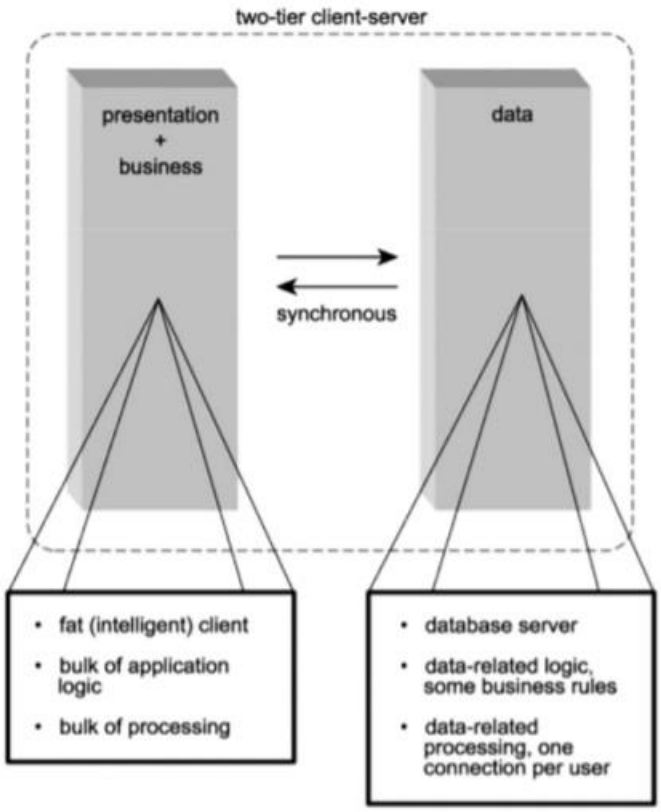
In January 2013, Beihang University became the Chinese host.

Comparing SOA to Client-Server and Distributed-Internet Architecture:

SOA vs. Client-Server Architecture The original monolithic mainframe systems that empowered organizations to get seriously computerized often are considered the first inception of client-server architecture. These environments, in which bulky mainframe back-ends served thin clients, are considered an implementation of the single-tier client-server architecture Mainframe systems natively supported both synchronous and asynchronous communication. The latter approach was used primarily to allow the server to continuously

receive characters from the terminal in response to individual key-strokes. Only upon certain conditions would the server actually respond. While its legacy still remains, the reign of the mainframe as the foremost computing platform began to decline when a two-tier variation of the client-server design merged in the late 80s. The common configuration of this architecture consisted of multiple fat clients, each with its own connection to a database on a central server. Client-side software performed the bulk of the processing, including all presentation-related and most data access logic. One or more servers facilitated these clients by hosting scalable RDBMSs. Let's look at the primary characteristics of the two-tier client-server architecture individually and compare them to the corresponding parts of SOA.

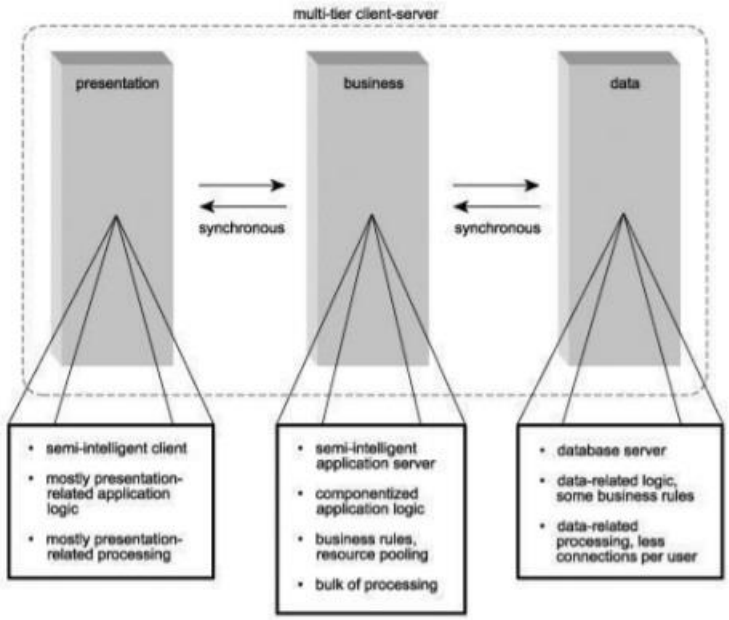




Application Processing A two-tier client-server solution with a large user-base generally requires that each client establish its own database connection. Communication is predictably synchronous, and these connections are often persistent (meaning that they are generated upon user login and kept active until the user exits the application). Proprietary database connections are expensive, and the resource demands sometimes overwhelm database servers, imposing processing latency on all users. Technology The emergence of client-server applications promoted the use of 4GL programming languages, such as Visual Basic and PowerBuilder. The technology set used by SOA actually has not changed as much as it has expanded. Newer versions of older programming languages, such as Visual Basic, still can be used to create Web services, and the use of relational databases still is commonplace. The technology landscape of SOA, though, has become increasingly diverse. In addition to the standard set of Web technologies (HTML, CSS, HTTP, etc.) contemporary SOA brings with it the absolute requirement that XML data representation architecture be established, along with a SOAP messaging framework, and a service architecture comprised of the ever-expanding Web services platform. Security Besides the storage and management of data and the business rules embedded in stored procedures and triggers, the one other part of client-server architecture that frequently is centralized at the server level is security. Databases are sufficiently sophisticated to manage user accounts and groups and to assign these to individual parts of the physical data model. Security also can be controlled within the client executable, especially when it relates to specific business rules that dictate the execution of application logic (such as limiting access to a part of a userinterface to select users). Additionally, operating system-level security can be incorporated to achieve a single sign-on, where application clearance is derived from the user's operating system login account information. Administration One of the main reasons the client-server era ended was the increasingly large maintenance costs associated with the distribution and maintenance of

application logic across user workstations. Because each client housed the application code, each update to the application required a redistribution of the client software to all workstations. In larger environments, this resulted in a highly burdensome administration process

Maintenance issues spanned both client and server ends. Client workstations were subject to environment-specific problems because different workstations could have different software programs installed or may have been purchased from different hardware vendors. Further, there were increased server-side demands on databases, especially when a client-server application expanded to a larger user base. Because service-oriented solutions can have a variety of requestors, they are not necessarily immune to client-side maintenance challenges. While their distributed back-end does accommodate scalability for application and database servers, new administration demands can be introduced. SOA vs. Distributed Internet Architecture In response to the costs and limitations associated with the two-tier client server architecture, the concept of building component-based applications hit the mainstream. Multi-tier client-server architectures surfaced, breaking up the monolithic client executable into components designed to varying extents of compliance with object-orientation. Distributing application logic among multiple components (some residing on the client, others on the server) reduced deployment headaches by centralizing a greater amount of the logic on servers. Server-side components, now located on dedicated application servers, would then share and manage pools of database connections, alleviating the burden of concurrent usage on the database server.



- Multiple client-server architectures have appeared
- Client-server DB connections have been replaced with Remote Procedure Call connections (RPC) using CORBA or DCOM.
- Middleware application servers and transaction monitors require significant attention
- Distributed Internet architecture introduced the Web server as a new physical tier and HTTP replaced RPC protocols.

- Distributed Internet application put all the application logic on the server side.
- Even client-side scripts are downloaded from the server
- Entire solution is centralized o Emphasis is on:
- How application logic is partitioned
- Where partitioned units reside
- How units of logic should interact
- Difference lies in the principles used to determine the three primary design considerations
- Traditional systems create components that reside on one or more application servers
- Components have varying degrees of functional granularity
- Components on the same server communicate via proprietary APIs.
- RPC protocols are used across servers via proxy stubs
- Actual references to other physical components can be embedded in programming code (tight coupling)
- SOAs also rely on components
- Services encapsulate components
- Services expose specific sets of functionality
- Functionality can originate from legacy systems or other sources
- Functionality is wrapped in services
- Functionality is exposed via open, standardized interface, irrespective of technology providing the solution
- Services exchange information via SOAP messages. SOAP supports RPC-style and documentstyle messages
- Most applications rely on document-style
- Messages are structured to be self-sufficient
- Messages contain Meta information, processing instructions, policy rules
- SOA fosters reuse on a deep level by promoting solution-agnostic services

Comparing SOA to Client-Server and Distributed Internet Architectures Client-Server Application Technology

- The technology set for client-server applications included 4GLs like VB and PowerBuilder, RDBMSs
- The SOA technology set has expanded to include Web technologies (HTML, CSS, HTTP, etc)
- SOA requires the use of XML data representation architecture along with a SOAP messaging framework.

Client-Server Application Security

- Centralized at the Server level o Databases manage user accounts and groups and also controlled within the client executable
- Security for SOA is much more complex
- Security complexity is directly related to the degree of security measures required
- Multiple technologies are required, many in WS-Security framework

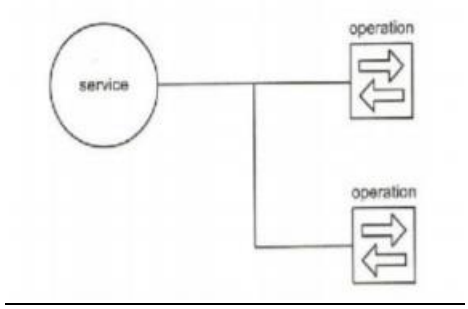
Client-Server Application Administration

- Significant maintenance costs associated with client-server
- Each client housed application code

- Each update required redistribution
- Client stations were subject to environment-specific problems
- Increased server-side demands on databases
- SOA solutions are not immune to client-side maintenance challenges
- Distributed back-end supports scalability, but new admin demands are introduced
- Management of server resources and service interfaces may require new admin tools and even a private registry
- Commitment to services and their maintenance may require cultural change in an organization

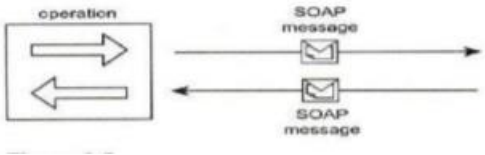
Logic Components of the Web Services Framework

- Web services contain one or more operations.



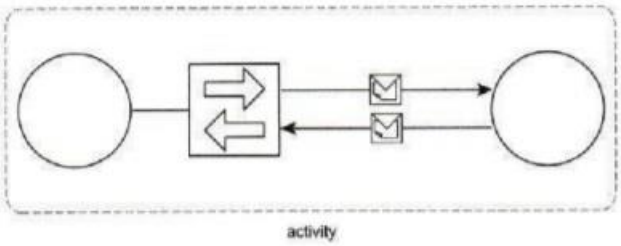
(A web service sporting two operations)

- Each operation governs the process of a specific function the web service is capable of performing.



(An operation processing outgoing and incoming SOAP messages)

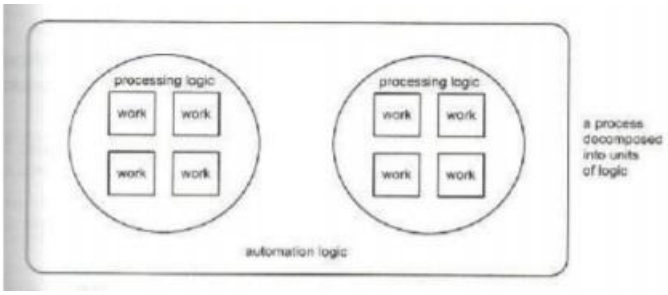
- Web services form an activity though which they can collectively automate a task.



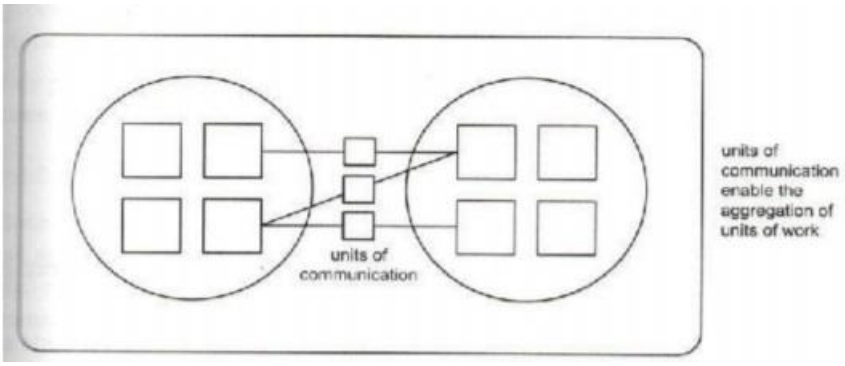
(A basic communications scenario between Web Services)

- Logic Components of Automation Logic
    - Fundamental parts of the framework
    - SOAP messages
    - Web service operations
    - Web services
    - Activities

The purpose of these views is to express the process, services and operations.
Is also provides a flexible means of partitioning and modularizing the logic.
These are the most basic concepts that underlie service-orientation



(a primitive view of how SOA modularizes automation logic into units)

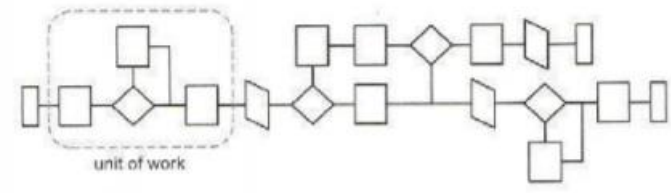


(a primitive view of how units of communication enable interaction between units of logic)

Components of an SOA

- Messages = units of communication
- Operations = units of work
- Services = units of processing logic
- Processes = units of automation logic
- **Message**: A message represents the data required to complete some or all parts of a unit of work.

- **Operation:**An operation represents the logic required to process messages in order to complete a unit of work

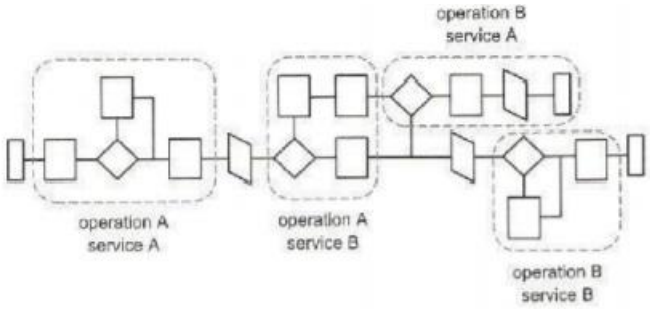


(the scope of an operation within a process)

- **Service** :A service represents a logically grouped set of operations capable of performing related units of work. Processes o A process contains the business

rules that determine which service operations are used to complete a unit of automation.

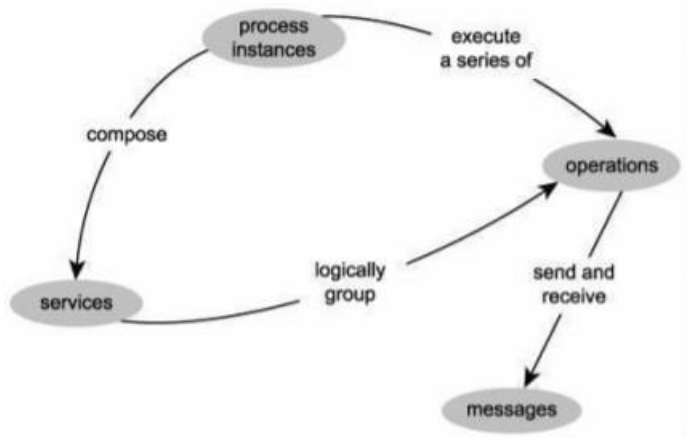
- A process represents a large piece of work that requires the completion of smaller units of work.



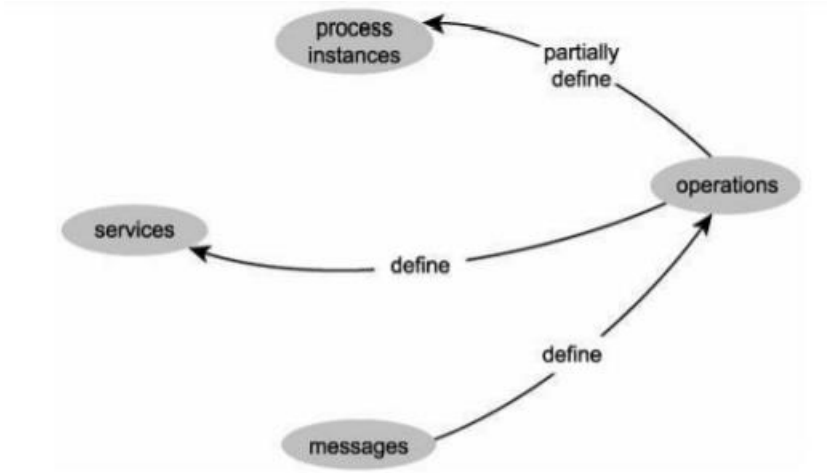
(operations belonging to different services representing various parts of process logic)

How Components in an SOA Inter-Relate:

Having established the core characteristics of our SOA components, let's now look at how these components are required to relate to each other:



(how the components of a service –oriented architecture relate)



(how the components of a service-oriented architecture define each other)

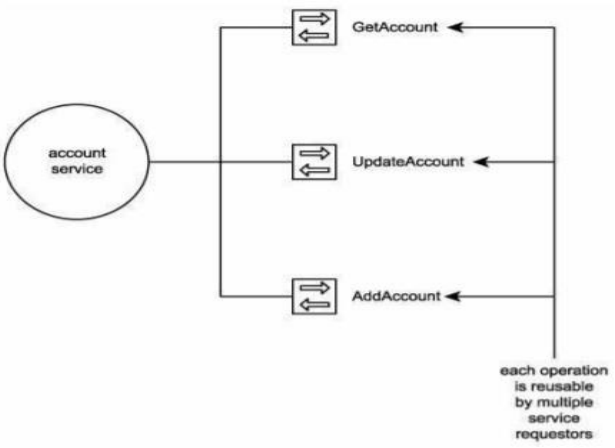
- An operation sends and receives messages to perform work.
- An operation is therefore mostly defined by the messages it processes.
- A service groups a collection of related operations. A service is therefore mostly defined by the operations that comprise it.
- A process instance can compose services.
- A process instance is not necessarily defined by its services because it may only require a subset of the functionality offered by the services.
- A process instance invokes a unique series of operations to complete its automation.
- Every process instance is therefore partially defined by the service operations it uses.

Common Principles of Service- Orientation

- Services are reusable
- Services share a formal contract
- Services are loosely coupled
- Services abstract underlying logic
- Services are composable
- Services are autonomous
- Services are stateless
- Services are discoverable Services Are Reusable
- Regardless of weather immediate reuse opportunities exist, services are designed to support potential reuse.
- Service-oriented encourages reuse in all services.
- By applying design standards that require reuse accommodate future requirements with less development effort.

Services Are Reusable:

- Regardless of weather immediate reuse opportunities exist, services are designed to support potential reuse.
- Service-oriented encourages reuse in all services.
- By applying design standards that require reuse accommodate future requirements with less development effort.



Case Study:

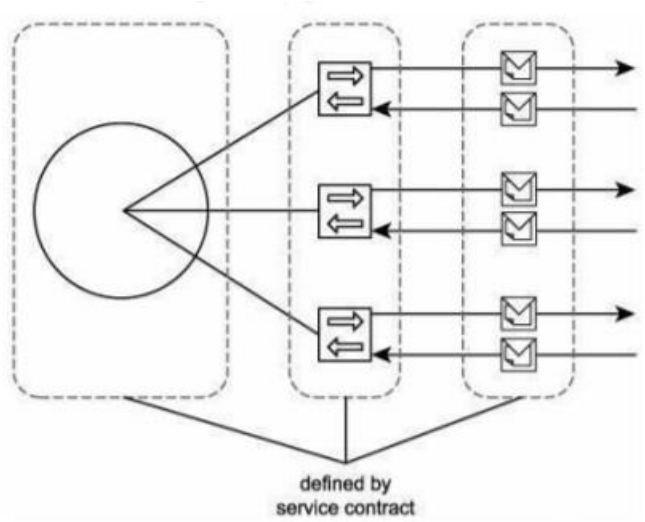
RailCo created the Invoice Submission Service which contains two operations

- SubmitInvoice
- GetTLSMetadata SubmitInvoice - Allows RailCo send electronic invoices to TLS Account Payable Service
- GetTLSMetadata – checks periodically for changes to TLS Account Payable Service

Services Share a Formal Contract For services to interact, they need not share anything but formal contract that describe each service and define the terms of information exchange.

Service contracts provide a formal definition of:

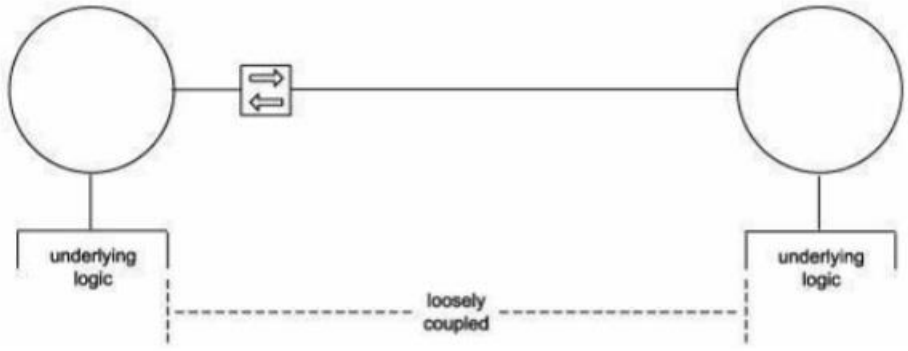
- The service endpoint
- Each service operation
- Every input and output message supported by each operation
- Rules and characteristics of the service and its operations Service contacts define almost all of the primary parts of an SOA.



Case Study:

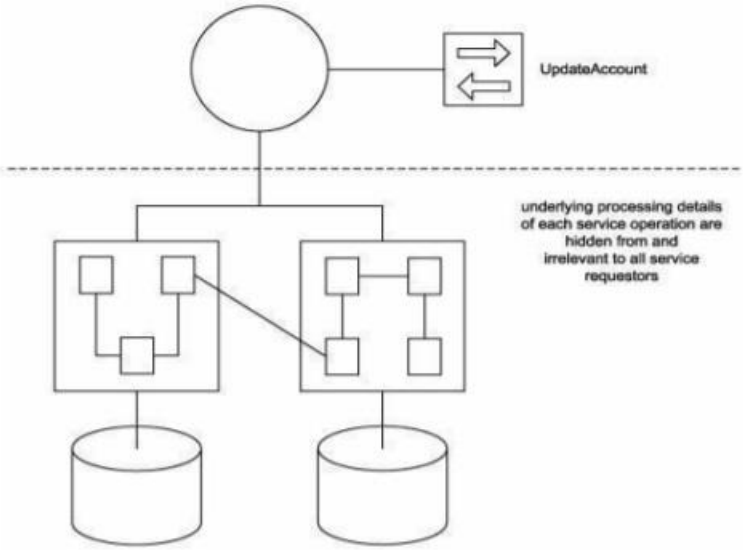
RailCo and TLS agreed to a service contract that allow the two companies to communicate TLS defined the definition of the associated service description documents TLS ensures a standardized level of conformance that applies to each of its online vendors TLS can change the service at any time

Services Are Loosely Coupled :Services must be designed to interact without the need for tight, cross-service dependencies.



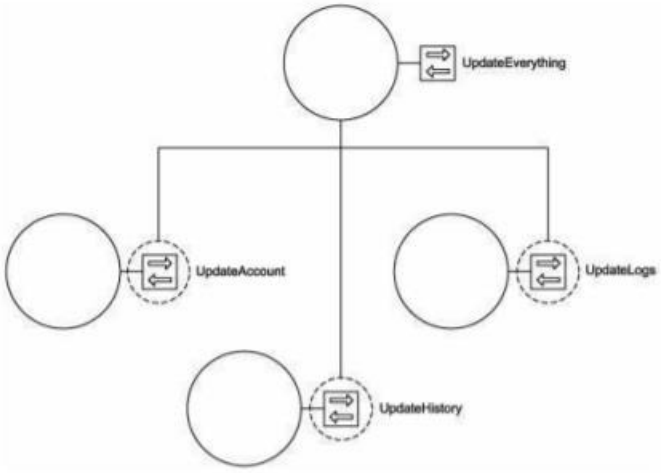
Case Study: TLS services are designed to communicate with multiple online vendors make it loosely coupled RailCo service are designed to communicate only with TLS B2B system so is considered less loosely coupled

Services Abstract Underlying Logic :The only part of a service that is visible to the outside world is what is exposed via the service contract. Underlying logic, beyond what is expressed in the descriptions that comprise the contract, is invisible and irrelevant to service requestors.



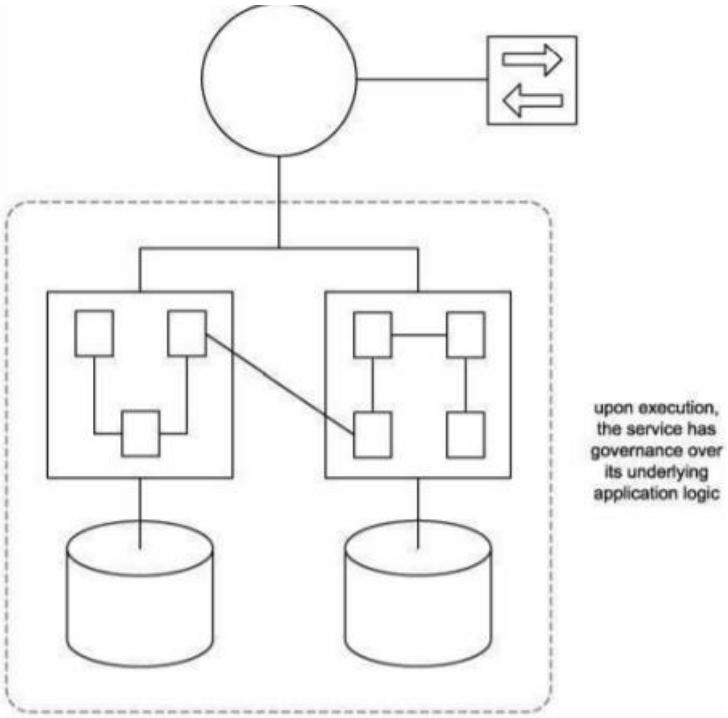
Case Study: RailCo Web services hide the underlying legacy code need to produce the invoices that are needed to sent to TLS TLS Web services hide the underlying services that process the invoices from multiple online vendors Neither service requestor require any knowledge of what processes are working on the other's service providers

Services Are Composable Services may be composing other services. This allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers



Case Study

- TLS accounts Payable Service is composed on three services o Accounts Payable Service Vendor Profile Service
- Ledger Service Services Are Autonomous The logic governed by a service resides within an explicit boundary. The service has control within this boundary and is not dependent on other services for it to execute its governance.



Note that autonomy does not necessarily grant a service exclusive ownership of the logic it encapsulates. It only guarantees that at the time of execution, the service has

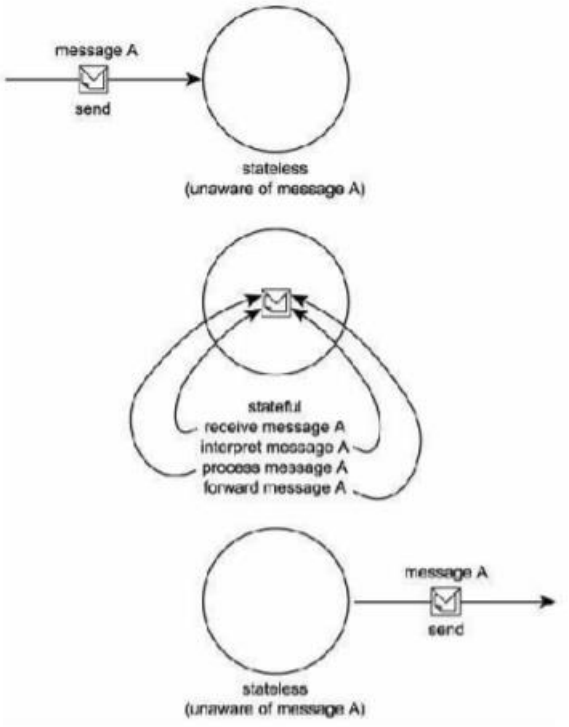
control over whatever logic it represents. We therefore can make a distinction between two types of autonomy.

Service-level autonomy: Service boundaries are distinct from each other, but the service may share underlying resources. For example, a wrapper service that encapsulates a legacy environment that also is used independently from the service has service-level autonomy. It governs the legacy system but also shares resources with other legacy clients.

Pure autonomy: The underlying logic is under complete control and ownership of the service. This is typically the case when the underlying logic is built from the ground up in support of the service.

Case Study: Given the distinct tasks they perform, the following three RailCo services all are autonomous: Invoice Submission Service Order Fulfillment Service TLS Subscription Service Each represents a specific boundary of application logic that does not overlap with the boundary of any other services. Autonomy in RailCo's services was achieved inadvertently. No conscious effort was made to avoid application overlap, as the services were delivered to simply meet specific connectivity requirements.

Services Are Stateless Services should not be required to manage state information, as that can impede their ability to remain loosely coupled. Services should be designed to maximize statelessness even if that means deferring state management elsewhere.



Statelessness is a preferred condition for services and one that promotes reusability and scalability. For a service to retain as little state as possible, its individual operations need to be designed with stateless processing considerations.

Case Study: As with loose coupling, statelessness is a quality that can be measured in degrees. The RailCo Order Fulfillment Service is required to perform extra runtime parsing and processing of various standard SOAP header blocks to successfully receive a purchase order document submitted by the TLS Purchase Order Service. This processing ties up the Order Fulfillment Service longer than, say, the Invoice Submission Service, which simply forwards a predefined SOAP message to the TLS Accounting Service.

Services are discoverable: Services should allow their descriptions to be discovered and understood by humans and service requestors that may be able to make use of their logic. Case Study RailCo provides no means of discovery for its services, either internally or to the outside world. Though outfitted with its own WSDL definition and fully capable of acting as a service provider, the Invoice Submission Service is primarily utilized as a service requestor and currently expects no communication outside of the TLS Accounts Payable Service