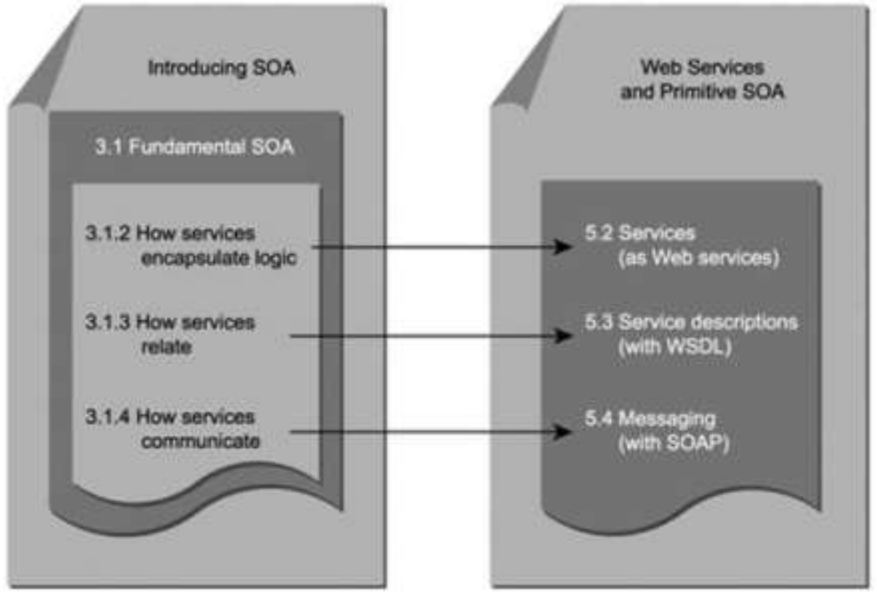
UNIT 4

The Web services framework

A technology framework is a collection of things. It can include one or more architectures, technologies, concepts, models, and even sub-frameworks. The framework established by Web services is comprised of all of these parts. Specifically, this framework is characterized by:

- an abstract (vendor-neutral) existence defined by standards organizations and implemented by (proprietary) technology platforms
- core building blocks that include Web services, service descriptions, and messages
- a communications agreement centered around service descriptions based on WSDL
- a messaging framework comprised of SOAP technology and concepts
- a service description registration and discovery architecture sometimes realized through UDDI
- a well-defined architecture that supports messaging patterns and a second generation of Web services extensions (also known as the WS-* specifications) continually broadening its underlying feature-set. It provides standards and best practices that govern the usage of WSDL, SOAP, and UDDI features. Therefore, much of what the Web services framework is comprised of can be standardized by the Basic Profile.


Introducing SOA
3.1 Fundamental SOA
3.1.2 How services encapsulate logic
3.1.3 How services relate
3.1.4 How services communicate
Web Services and Primitive SOA
5.2 Services (as Web services)
5.3 Service descriptions (with WSDL)
5.4 Messaging (with SOAP)

Services (as Web services)

The concept of services and how they provide a means of encapsulating various extents of logic. Manifesting services in real world automation solutions requires the use of a technology capable of preserving fundamental service-orientation, while implementing real world business functionality.

Web services provide the potential of fulfilling these primitive requirements, but they need to be intentionally designed to do so. This is because the Web services framework is flexible and adaptable. Web services can be designed to duplicate the behavior and functionality found in proprietary distributed systems, or they can be designed to be fully SOA-compliant. This flexibility has allowed Web services to become part of many existing application environments and has been one of the reasons behind their popularity. It also reveals the fact that Web services are not necessarily inherently service-oriented.

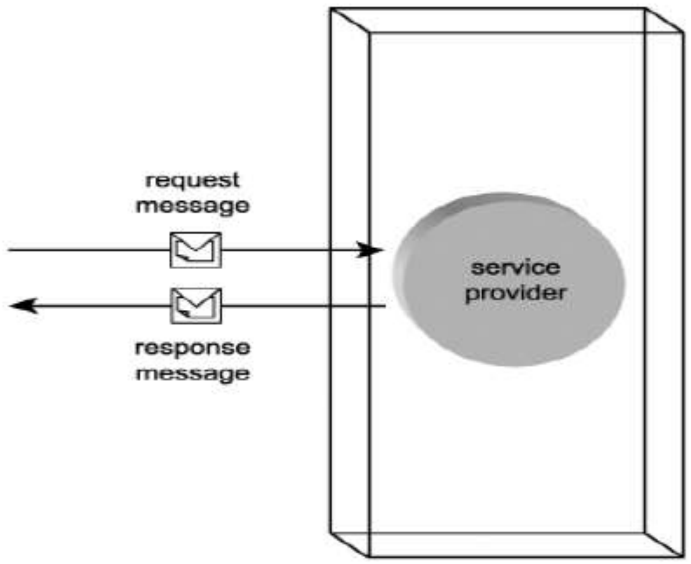
Service roles

A Web service is capable of assuming different roles, depending on the context within which it is used. For example, a service can act as the initiator, relayer, or the recipient of a message. A service is therefore not labeled exclusively as a client or server, but instead as a unit of software capable of altering its role, depending on its processing responsibility in a given scenario. It is not uncommon for a Web service to change its role more than once within a given business task. It is especially not uncommon for a Web service within an SOA to assume different roles in different business tasks.

Service provider

The *service provider* role is assumed by a Web service under the following conditions:

- The Web service is invoked via an external source, such as a service requestor



The term "service provider" also is used to identify the organization (or individual) responsible for actually providing the Web service. To help distinguish the service role from the service's actual provider, the following, more qualified terms are sometimes used:

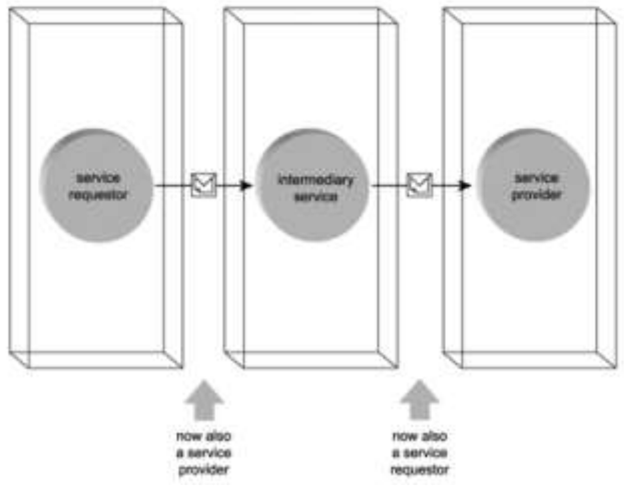
- *service provider entity* (the organization or individual providing the Web service)

- *service provider agent* (the Web service itself, acting as an agent on behalf of its owner)

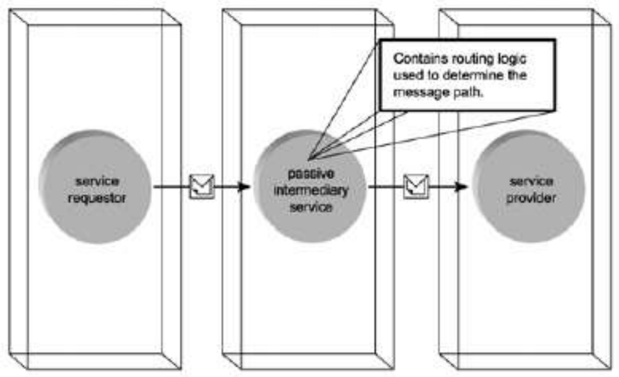
Intermediaries

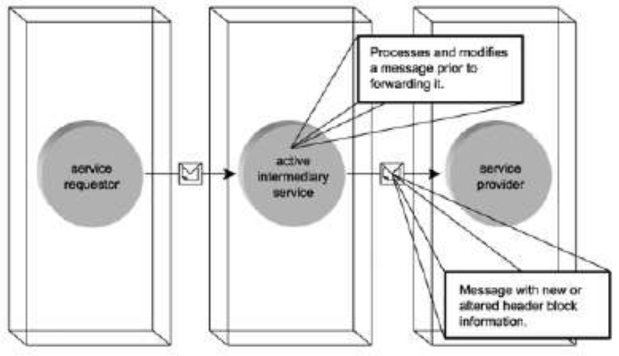
The communications framework established by Web services contrasts the predictable nature of traditional point-to-point communications channels. Though less flexible and less scalable, point-to-point communication was a great deal easier to design. Web services communication is based on the use of messaging paths, which can best be described as point-to-* paths. This is because once a service provider submits a message, it can be processed by multiple intermediate routing and processing agents before it arrives at its ultimate destination.

Web services and service agents that route and process a message after it is initially sent and before it arrives at its ultimate destination are referred to as *intermediaries* or *intermediary services*. Because an intermediary receives and submits messages, it always transitions through service provider and service requestor roles



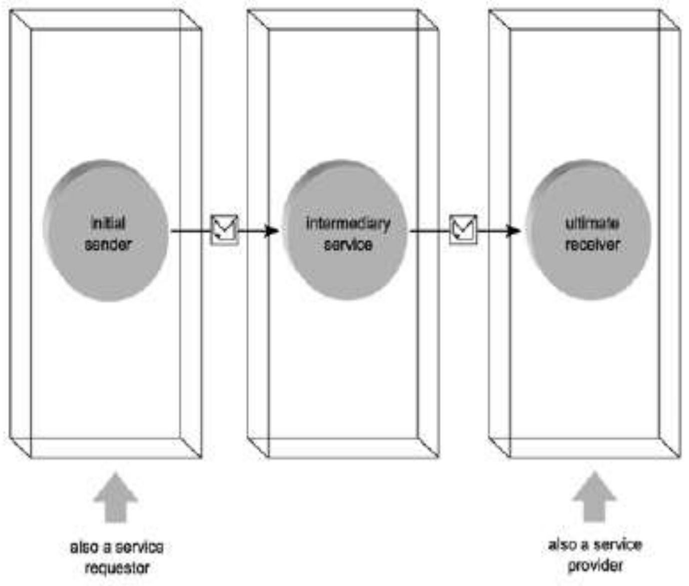
There are two types of intermediaries. The first, known as a *passive intermediary*, is typically responsible for routing messages to a subsequent location (Figure 5.6). It may use information in the SOAP message header to determine the routing path, or it may employ native routing logic to achieve some level of load balancing. Either way, what makes this type of intermediary passive is that it does not modify the message.


Contains routing logic used to determine the message path.
service requestor
passive intermediary service
service provider


Processes and modifies a message prior to forwarding it.
service requestor
active intermediary service
service provider
Message with new or altered header block information.

Initial sender and ultimate receiver

Initial senders are simply service requestors that initiate the transmission of a message. Therefore, the initial sender is always the first Web service in a message path. The counterpart to this role is the *ultimate receiver*. This label identifies service providers that exist as the last Web service along a message's path



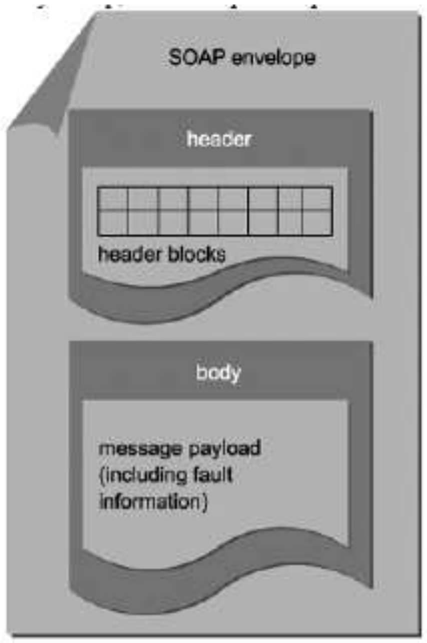
Service compositions

As the name suggests, this particular term does not apply to a single Web service, but to a composite relationship between a collection of services. Any service can enlist one or more additional services to complete a given task. Further, any of the enlisted services can call other services to complete a given sub-task. Therefore, each service that participates in a composition assumes an individual role of *service composition member*

Messaging (with SOAP)

The SOAP specification was chosen to meet all of these requirements and has since been universally accepted as the standard transport protocol for messages processed by Web services. **Messages** Even though it was originally named the Simple Object Access *Protocol*, the SOAP specification's main purpose is to define a standard message format. **Envelope, header, and body**

Every SOAP message is packaged into a container known as an *envelope*. Much like the metaphor this conjures up, the envelope is responsible for housing all parts of the message



Each message can contain a *header*, an area dedicated to hosting meta information. In most service-oriented solutions, this header section is a vital part of the overall architecture, and though optional, it is rarely omitted. Its importance relates to the use of header blocks through which numerous extensions can be implemented (as described next).

The actual message contents are hosted by the message *body*, which typically consists of XML formatted data. The contents of a message body are often referred to as the message *payload*

Header blocks

Message independence is implemented through the use of *header blocks*, packets of supplementary meta information stored in the envelope's header area. Header blocks outfit a message with all of the information required for any services with which the message comes in contact to process and route the message in accordance with its accompanying rules, instructions, and properties. What this means is that through the use of header blocks, SOAP messages are capable of containing a large variety of supplemental information related to the delivery and processing of message contents.

Message styles

The SOAP specification was originally designed to replace proprietary RPC protocols by allowing calls between distributed components to be serialized into XML documents, transported, and then deserialized into the native component format upon arrival. As a result, much in the original version of this specification centered around the structuring of messages to accommodate RPC data.

This *RPC-style* message runs contrary to the emphasis SOA places on independent, intelligence-heavy messages. SOA relies on *document-style* messages to enable larger payloads, coarser interface operations, and reduced message transmission volumes between services.

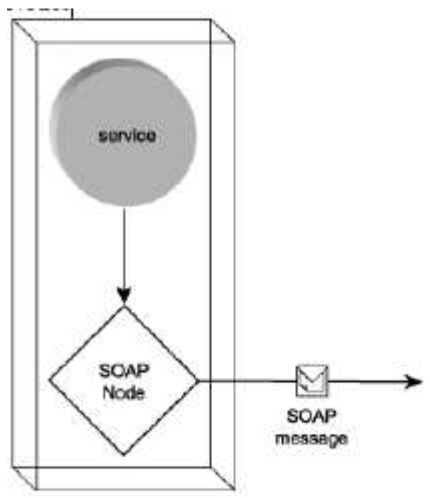
Attachments

To facilitate requirements for the delivery of data not so easily formatted into an XML document, the use of *SOAP attachment* technologies exist. Each provides a different encoding mechanism used to bundle data in its native format with a SOAP message. SOAP attachments are commonly employed to transport binary files, such as images.

Faults

Finally, SOAP messages offer the ability to add exception handling logic by providing an optional *fault* section that can reside within the body area. The typical use for this section is to store a simple message used to deliver error condition information when an exception occurs

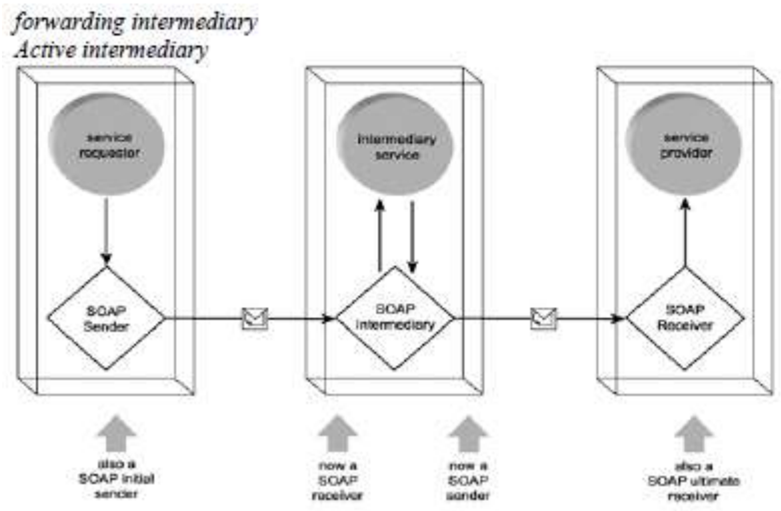
Nodes



- *SOAP sender*a SOAP node that transmits a message
- *SOAP receiver*a SOAP node that receives a message
- *SOAP intermediary*a SOAP node that receives and transmits a message, and optionally processes the message prior to transmission
- *initial SOAP sender*the first SOAP node to transmit a message
- *ultimate SOAP receiver*the last SOAP node to receive a message

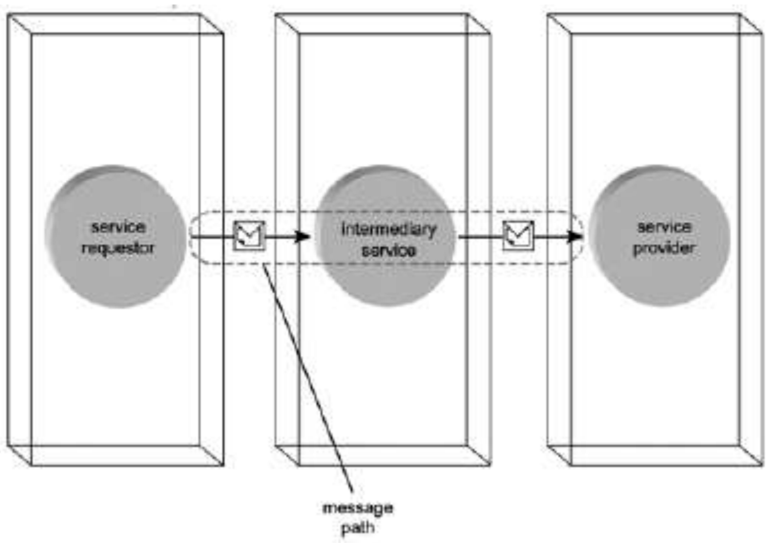
SOAP intermediaries

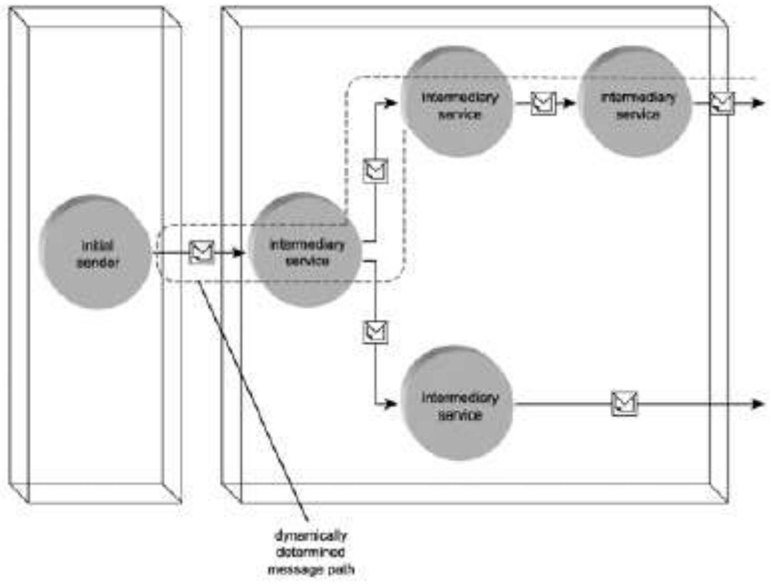
The same way service intermediaries transition through service provider and service requestor roles, *SOAP intermediary* nodes move through SOAP receiver and SOAP sender types when processing a message



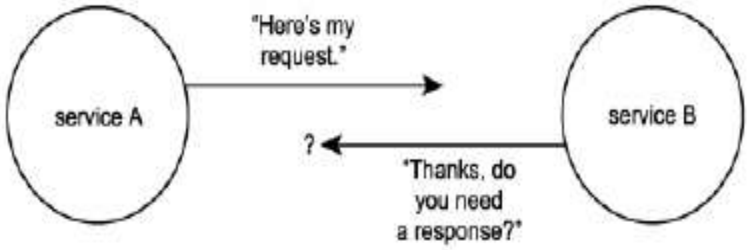
Message paths

A *message path* refers to the route taken by a message from when it is first sent until it arrives at its ultimate destination. Therefore, a message path consists of at least one initial sender, one ultimate receiver, and zero or more intermediaries





Message exchange patterns



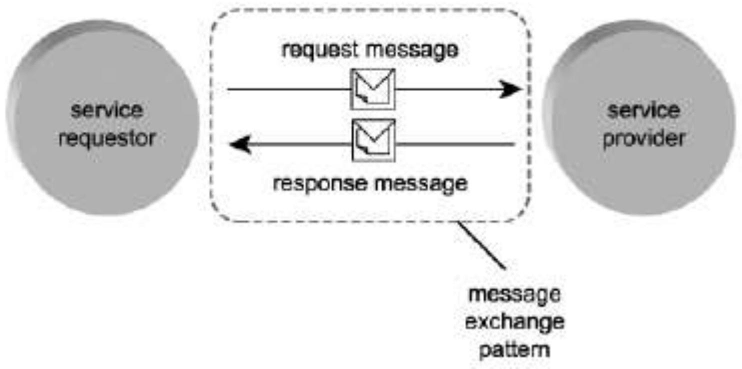
Message exchange patterns (MEPs) represent a set of templates that provide a group of already mapped out sequences for the exchange of messages. The most common example is a request and response pattern. Here the MEP states that upon successful delivery of a message from one service to another, the receiving service responds with a message back to the initial requestor.

Primitive MEPs Request-response

This is the most popular MEP in use among distributed application environments and the one pattern that defines synchronous communication (although this pattern also can be applied asynchronously).

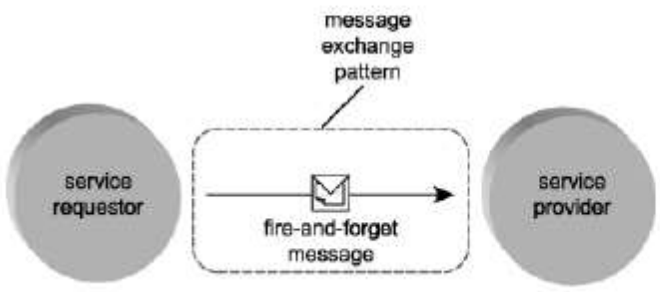
The *request-response* MEP establishes a simple exchange in which a message is first transmitted from a source (service requestor) to a destination (service provider). Upon receiving the message, the destination (service provider) then responds with a message back to the source (service requestor).

The request-response MEP.



Fire-and-forget

This simple asynchronous pattern is based on the unidirectional transmission of messages from a source to one or more destinations.



A number of variations of the *fire-and-forget* MEP exist, including:

- The *single-destination* pattern, where a source sends a message to one destination only.

- The *multi-cast* pattern, where a source sends messages to a predefined set of destinations.
- The *broadcast* pattern, which is similar to the multi-cast pattern, except that the message is sent out to a broader range of recipient destinations.

Complex MEPs

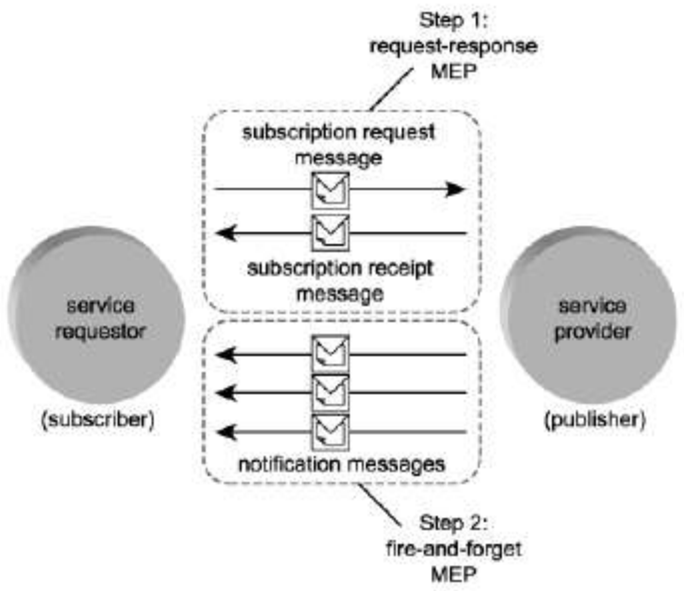
The publish-and-subscribe pattern introduces new roles for the services involved with the message exchange. They now become publishers and subscribers, and each may be involved in the transmission and receipt of messages. This asynchronous MEP accommodates a requirement for a publisher to make its messages available to a number of subscribers interested in receiving them. The steps involved are generally similar to the following:

Step 1.

The subscriber sends a message to notify the publisher that it wants to receive messages on a particular topic.

Step 2.

Upon the availability of the requested information, the publisher broadcasts messages on the particular topic to all of that topic's subscribers.

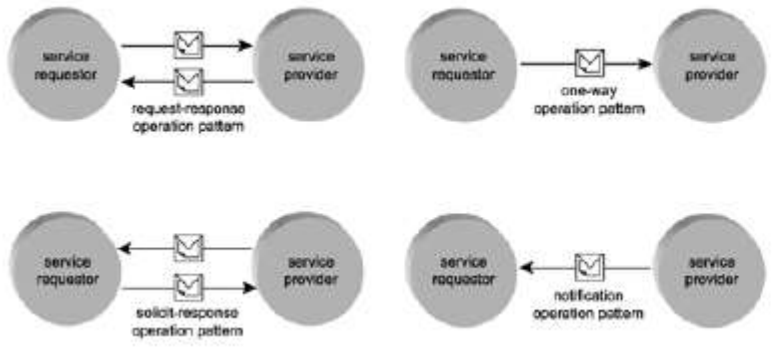


MEPs and SOAP

On its own, the SOAP standard provides a messaging framework designed to support single-direction message transfer. The extensible nature of SOAP allows countless messaging characteristics and behaviors (MEP-related and otherwise) to be implemented via SOAP header blocks. The SOAP language also provides an optional parameter that can be set to identify the

MEP associated with a message. (Note that SOAP MEPs also take SOAP message compliance into account.) **. MEPs and WSDL** Release 1.1 of the WSDL specification provides support for four message exchange patterns that roughly correspond to the MEPs we described in the previous section. These patterns are applied to service operations from the perspective of a service provider or endpoint. In WSDL 1.1 terms, they are represented as follows:

- *Request-response operation* Upon receiving a message, the service must respond with a standard message or a fault message.
- *Solicit-response operation* Upon submitting a message to a service requestor, the service expects a standard response message or a fault message.
- *One-way operation* The service expects a single message and is not obligated to respond.
- *Notification operation* The service sends a message and expects no response.



release 2.0 of the WSDL specification extends MEP support to eight patterns (and also changes the terminology) as follows.

- The *in-out pattern*, comparable to the request-response MEP (and equivalent to the WSDL 1.1 request-response operation).
- The *out-in pattern*, which is the reverse of the previous patternwhere the service provider initiates the exchange by transmitting the request. (Equivalent to the WSDL 1.1 solicit-response operation.)
- The *in-only pattern*, which essentially supports the standard fire-and-forget MEP. (Equivalent to the WSDL 1.1 one-way operation.)
- The *out-only pattern*, which is the reverse of the in-only pattern. It is used primarily in support of event notification. (Equivalent to the WSDL 1.1 notification operation.)
- The *robust in-only pattern*, a variation of the in-only pattern that provides the option of launching a fault response message as a result of a transmission or processing error.
- The *robust out-only pattern*, which, like the out-only pattern, has an outbound message initiating the transmission. The difference here is that a fault message can be issued in response to the receipt of this message.
- The *in-optional-out pattern*, which is similar to the in-out patternwith one exception. This variation introduces a rule stating that the delivery of a response message is optional and

should therefore not be expected by the service requestor that originated the communication. This pattern also supports the generation of a fault message.

- The *out-optional-in pattern* is the reverse of the in-optional-out pattern, where the incoming message is optional. Fault message generation is again supported.

MEPs and SOA

MEPs are highly generic and abstract in nature. Individually, they simply relate to an interaction between two services. Their relevance to SOA is equal to their relevance to the abstract Web services framework. They are therefore a fundamental and essential part of any Web services-based environment, including SOA.

Service discovery

Service discovery is the process of locating Web service providers, and retrieving Web services descriptions that have been previously published.

• Interrogating services involve querying the service registry for Web services matching the needs of a service requestor.

– A query consists of search criteria such as:

- the type of the desired service, preferred price and maximum number of returned results, and is executed against service information published by service provider.

- Discovering Web services is a process that is also dependent on the architecture of the service registry.

• After the discovery process is complete, the service developer or client application should know the exact location of a Web service (URI), its capabilities, and how to interface with it.

Types of service discovery

Static

• The service implementation details are bound at design time and a service retrieval is performed on a service registry.

• The results of the retrieval operation are examined usually by a human designer and the service description returned by the retrieval operation is incorporated into the application logic.

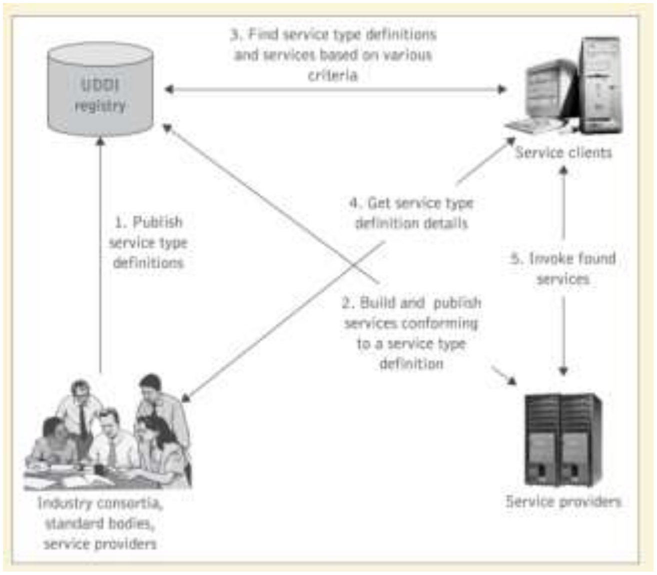
Dynamic

- The service implementation details are left unbound at design time so that they can be determined at run-time.
- The Web service requestor has to specify preferences to enable the application to infer/reason which Web service(s) the requester is most likely to want to invoke.
- Based on application logic quality of service considerations such as best price, performance, security certificates, and so on, the application chooses the most appropriate service, binds to it, and invokes it.

Universal Description, Discovery and Integration (UDDI)

The universal description, discovery, and integration is a registry standard for Web service description and discovery together with a registry facility that supports the WS publishing and discovery processes.

- UDDI enables a business to:
  - describe its business and its services;
  - discover other businesses that offer desired services;
  - integrate (interoperate) with these other businesses.
- Conceptually, a UDDI business registration consists of three inter-related components:
  - "white pages" (address, contact, and other key points of contact);
  - "yellow pages" classification info. based on standard industry taxonomies;
  - "green pages", the technical capabilities and information about services.


3. Find service type definitions and services based on various criteria
UDDI registry
Service clients
1. Publish service type definitions
4. Get service type definition details
5. Invoke found services
2. Build and publish services conforming to a service type definition
Industry consortia, standard bodies, service providers
Service providers

UDDI – main characteristics

- UDDI provides a mechanism to categorize businesses and services using taxonomies.
  - Service providers can use a taxonomy to indicate that a service implements a specific domain standard, or that it provides services to a specific geographic area
  - UDDI uses standard taxonomies so that information can be discovered on the basis of categorization.

• UDDI business registration: an XML document used to describe a business entity and its Web services.

• UDDI is not bound to any technology. In other words,

– An entry in the UDDI registry can contain any type of resource, independently of whether the resource is XML based or not, e.g., the UDDI registry could contain information about an enterprise's electronic document interchange (EDI) system or even a service that, uses the fax machine as its primary communication channel.

– While UDDI itself uses XML to represent the data it stores, it allows for other kinds of technology to be registered.

UDDI – Data Structures

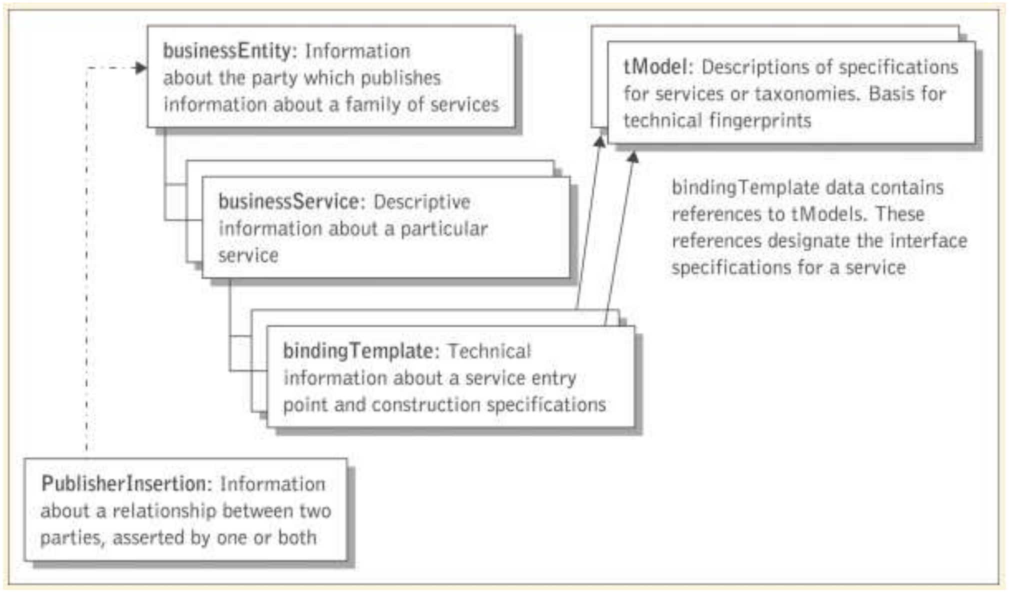
UDDI also defines a data structure standard for representing company and service description information. The UDDI XML schema defines four core types of information.These are

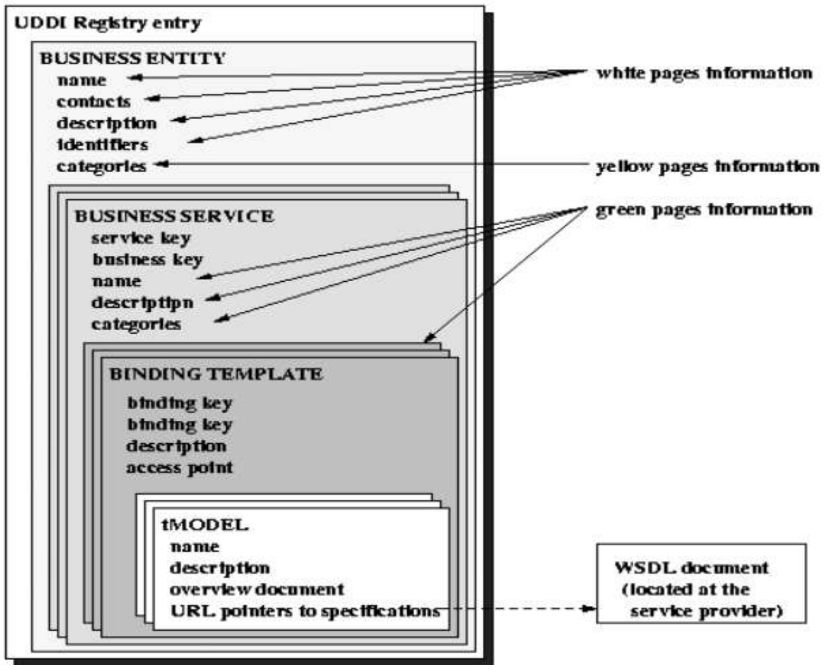
– businessEntity: a description of the organization that provides the service.

– businessService: a list of all the Web services offered by the business entity.

– bindingTemplate: describes the technical aspects of the service being offered.

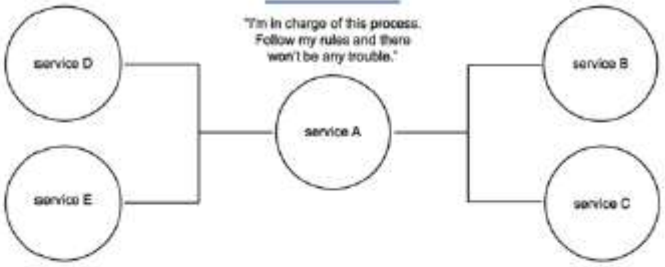
– tModel: (“technical model”)is a generic element that can be used to store technical information on how to use the service, conditions for use, guarantees, etc.


businessEntity: Information about the party which publishes information about a family of services
tModel: Descriptions of specifications for services or taxonomies. Basis for technical fingerprints
businessService: Descriptive information about a particular service
bindingTemplate data contains references to tModels. These references designate the interface specifications for a service
bindingTemplate: Technical information about a service entry point and construction specifications
PublisherInsertion: Information about a relationship between two parties, asserted by one or both



Orchestration

- A centrally controlled set of workflow logic facilitates interoperability between two or more different applications.
- A common implementation of orchestration is the hub-and-spoke model that allows multiple external participants to interface with a central orchestration engine.
- With orchestration, different processes can be connected that originally automated the processes individually.
- The use of orchestration can significantly reduce the complexity of solution environments.
- Workflow logic is abstracted and more easily maintained than when embedded within individual solution components.
- This allows for business process logic to be expressed via services, orchestration can represent and express business logic in a standardized, services-based venue.
- When building service-oriented solutions, this provides housing and controlling the logic representing the process being automated.
- It provides potential integration endpoints into processes.
- The orchestrations themselves exist as services. Therefore, building upon orchestration logic standardizes process representation across an organization, while addressing the goal of enterprise federation and promoting service-orientation.


"I'm in charge of this process. Follow my rules and there won't be any trouble."
service D
service E
service A
service B
service C

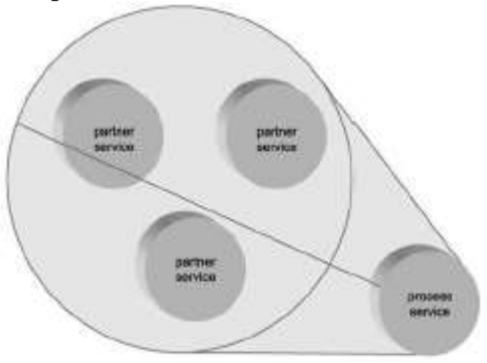
Business protocols and process definition

- The workflow logic that comprises an orchestration can consist of numerous business rules, conditions, and events. Collectively, these parts of an orchestration establish a business protocol that defines how participants can interoperate to achieve the completion of a business task.
- The details of the workflow logic encapsulated and expressed by an orchestration are contained within a process definition.

Process services and partner services

- Identified and described within a process definition are the allowable process participants. First, the process itself is represented as a service, resulting in a process service

- Other services allowed to interact with the process service are identified as partner services or partner links. Depending on the workflow logic, the process service can be invoked by an external partner service, or it can invoke other partner services



A process service coordinating and exposing functionality from three partner services

• Basic activities and structured activities
  - WS-BPEL breaks down workflow logic into a series of predefined primitive activities. Basic activities (receive, invoke, reply, throw, wait) represent fundamental workflow actions which can be assembled using the logic supplied by structured activities (sequence, switch, while, flow, pick).

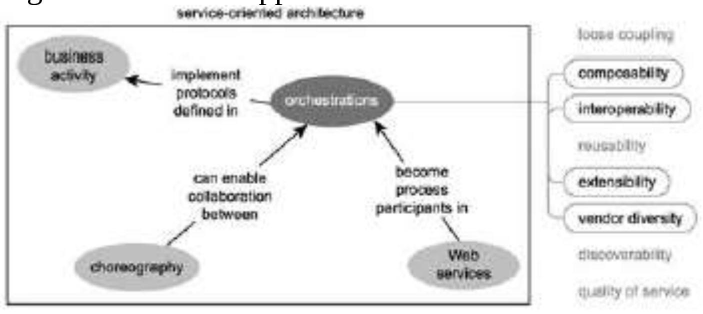
• Sequences, flows, and links
  - Basic and structured activities can be organized so that the order in which they execute is predefined. A sequence aligns groups of related activities into a list that determines a sequential execution order. Sequences are especially useful when one piece of application logic is dependent on the outcome of another.
  - Pieces of application logic can execute concurrently within a flow, meaning that there is not necessarily a requirement for one set of activities to wait before another finishes. However, the flow itself does not finish until all encapsulated activities have completed processing. This ensures a form of synchronization among application logic residing in individual flows.
    - Links are used to establish formal dependencies between activities that are part of flows. Before an activity fully can complete, it must ensure that any requirements established in outgoing links first are met. Similarly, before any linked activity can begin, requirements contained within any incoming links first must be satisfied. Rules provided by links are also referred to as synchronization dependencies.

• Orchestrations and activities
  - An activity is a generic term that can be applied to any logical unit of work completed by a service-oriented solution. The scope of a single orchestration, therefore, can be classified as a complex, and most likely, long-running activity.

• Orchestration and coordination
  - Orchestration, as represented by WS-BPEL, can fully utilize the WS-Coordination context management framework by incorporating the WS-BusinessActivity coordination type. This specification defines coordination protocols designed to support complex, long-running activities.

• Orchestration and SOA

- Orchestration provides an automation model where process logic is centralized yet still extensible and compos able . Orchestrations themselves establish a common point of integration for other applications.

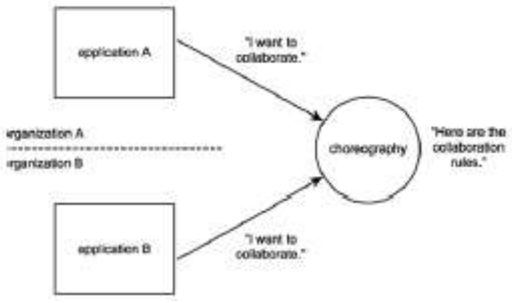


Orchestration relating to other parts of SOA

Choreography

The Web Services Choreography Description Language (WS-CDL) is a specification that attempts to organize information exchange between multiple organizations (or even multiple applications within organizations), with an emphasis on public collaboration

- choreography describes the global protocol governing how individual participants interact with one another. Each participant has its own process, but choreography is a master process that acts as a kind of traffic cop.
- It merely a message exchange protocol. If the participants follow the protocol, the live exchange will run as smoothly as if there were a central broker.
- choreography is more like a set of traffic rules. To mix metaphors, choreography teaches the participant processes how to dance as a group.



Collaboration

- An important characteristic of choreographies is that they are intended for public message exchanges.

- The goal is to establish a kind of organized collaboration between services representing different service entities, only no one entity (organization) necessarily controls the collaboration logic.
- Choreographies provide the potential for establishing universal interoperability patterns for common inter-organization business tasks.

• Roles and participants

- Within any given choreography, a Web service assumes one of a number of predefined roles. This establishes what the service does and what the service can do within the context of a particular business task.
- Roles can be bound to WSDL definitions, and those related are grouped accordingly, categorized as participants (services).

• Relationships and channels

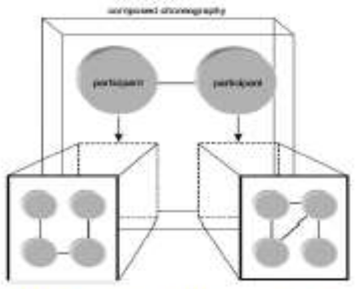
- Every action that is mapped out within a choreography can be broken down into a series of message exchanges between two services. Each exchange between two roles in a choreography is defined individually as a relationship. Every relationship consequently consists of exactly two roles.
- Channels defining the characteristics of the message exchange between two specific roles.
- Further, to facilitate more complex exchanges involving multiple participants, channel information can be passed around in a message. This allows one service to send another the information required for it to be communicated with by other services. This is a significant feature of the WS-CDL specification, as it fosters dynamic discovery and increases the number of potential participants within large-scale collaborative tasks.

• Interactions and work units

- The actual logic behind a message exchange is encapsulated within an interaction. Interactions are the fundamental building blocks of choreographies because the completion of an interaction represents actual progress within a choreography.
- Related to interactions are work units. These impose rules and constraints that must be adhered to for an interaction to successfully complete.

• Reusability, composability, and modularity

- Each choreography can be designed in a reusable manner, allowing it to be applied to different business tasks comprised of the same fundamental actions.
- Further, using an import facility, a choreography can be assembled from independent modules. These modules can represent distinct sub-tasks and can be reused by numerous different parent choreographies

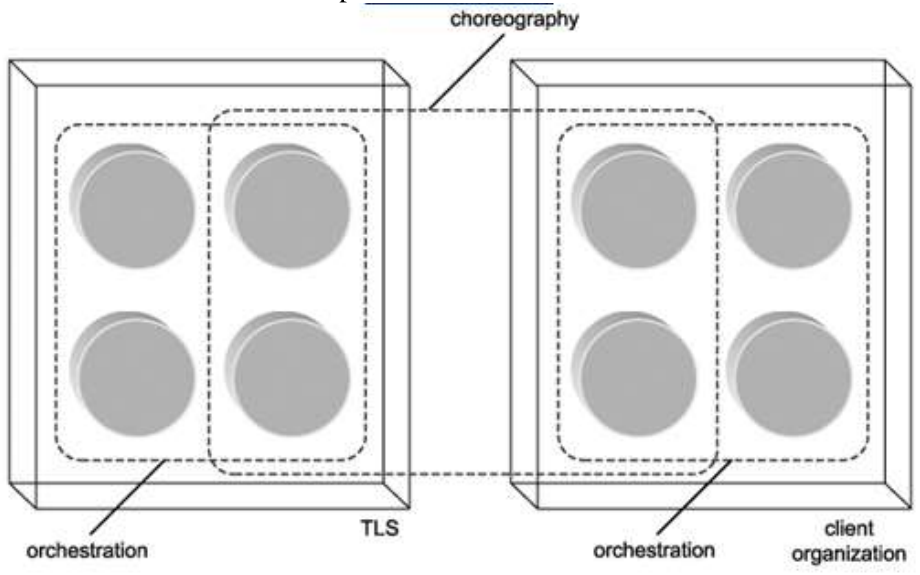


choreography composed of two choreographies

Orchestrations and choreographies

- While both represent complex message interchange patterns, there is a common distinction that separates the terms "orchestration" and "choreography."

- An orchestration expresses organization-specific business workflow. This means that an organization owns and controls the logic behind an orchestration, even if that logic involves interaction with external business partners.
  - A choreography, on the other hand, is not necessarily owned by a single entity. It acts as a community interchange pattern used for collaborative purposes by services from different provider entities


choreography
orchestration
TLS
orchestration
client organization