## A Web service

Web services form the building blocks for creating distributed applications in that they can be published to and accessed over the Internet and corporate intranets. They rely on a set of open Internet standards that allow developers to implement distributed applications – using different tools provided by many different vendors – to create corporate applications that join together possible existing.

But more importantly, Web services can also be combined and/or configured by these distributed applications, behind the scenes to perform virtually any kind of (business-related) task or transaction.

Web services can discover and communicate with other Web services and trigger them to fulfill or outsource part of a higher-level transaction by using a common vocabulary (business terminology) and a published directory of their capabilities according to a referrence architecture called the service-oriented architecture  The modularity and flexibility of Web services make them ideal for e-business application integration [Papazoglou 2006]. For example, the inventory Web service can be accessed together with other related Web services by a business partner's warehouse management application or can be part of a new distributed application that is developed from scratch and implements an extended value chain supply planning solution.

So **web service can be defined as a platform-independent, loosely coupled, self-contained, programmable Web-enabled application that can be described, published, discovered, coordinated, and configured using XML artifacts (open standards) for the purpose of developing distributed interoperable applications**. Web services possess the ability to engage other services in a common computation in order to: complete a concrete task, conduct a business transaction, or solve a complex problem. In addition, Web services expose their features programmatically over the Internet (or intranet) using standard Internet languages (based on XML) and standard protocols, and can be implemented via a self-describing interface based on open Internet standards.

**Why is a service description needed?**

To develop service-based applications and business processes, which comprise service assemblies, Web services need to be described in a consistent manner. In this way Web services can be published by service providers, discovered by service clients and developers, and assembled in a manageable hierarchy of composite services that are orchestrated to deliver value-added service solutions and composite application assemblies. However, in order to accomplish this, consumers must determine the precise XML interface of a Web service along with other miscellaneous message details a priori. In the Web services world, XML Schema can partially fill this need as it allows developers to describe the structure of XML messages understood by Web services. However, XML Schema alone cannot describe important additional details involved in communicating with a Web service such as service functional and non-functional characteristics.

**Service description is a key to making the SOA loosely coupled and reducing the amount of required common understanding, custom programming, and integration between the service provider and the service requestor's applications. Service description is a machine-understandable specification describing the structure, operational characteristics, and non-functional properties of a Web service. It also specifies the wire format and transport protocol that the Web service uses to expose this functionality.** It can also describe the payload data using a type system. The service description combined with the underlying SOAP infrastructure sufficiently isolates all technical details, e.g., machine- and implementation-language-specific elements, from the service requestor's application and the service provider's Web service.

**SOAP BASICS AND SOAP MESSAGING**

To address the problem of overcoming proprietary systems running on heterogeneous infrastructures, Web services rely on SOAP, an XML-based communication protocol for exchanging messages between computers regardless of their operating systems, programming environment, or object model framework. SOAP was originally an acronym for Simple Object Access Protocol (now it is just a name). SOAP is the de facto standard messaging protocol used by Web services. SOAP's primary application is inter-application communication. SOAP codifies the use of XML as an encoding scheme for request and response parameters using HTTP as a means for transport. In particular, a SOAP method is simply an HTTP request and response that complies with the SOAP encoding rules. A SOAP endpoint is simply an HTTP-based URL that identifies a target for method invocation.

**SOAP provides a wire protocol in that it specifies how service-related messages are structured when exchanged across the Internet**. SOAP can be defined as a lightweight wire protocol for exchanging structured and typed information back and forth between disparate systems in a distributed environment, such as the Internet or even a LAN (Local Area Network), enabling remote method invocation. The term lightweight wire protocol means that SOAP possesses only two fundamental properties. It can send and receive HTTP (or other) transport protocol packets, and process XML messages

One distinction that many people find confusing is the difference between the wire protocol (format) and the transport protocol. Whereas the wire protocol specifies the form or shape of the data to be exchanged between disparate applications, and eventually systems, the transport protocol is the method by which that data is transferred from system to system. The transport protocol is responsible for taking its payload from its point of origin to its destination.

**Structure of SOAP Message:**

A SOAP element contains blocks of information relevant to how the message is to be processed. This provides a way to pass information in SOAP messages that is not part of the application payload. Such "control" information includes, for example, passing directives or contextual information related to the processing of the message, e.g., routing and delivery settings, authentication or authorization assertions, and transaction contexts. This allows a SOAP message to be extended in an application-specific manner.
A pictorial representation of the SOAP message structure is shown in Figure 4.5

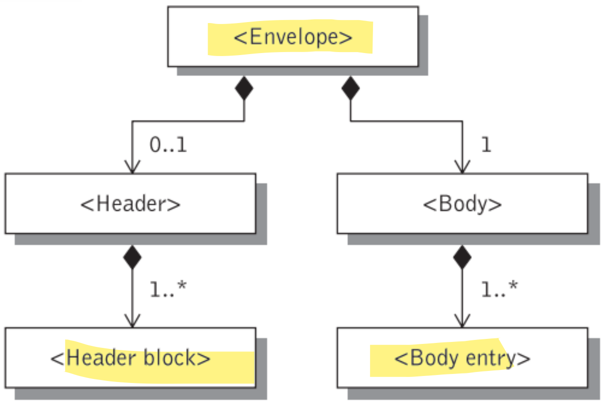


**Figure 4.5** The SOAP message containment structure described in UML

**SOAP Envelope:** Listing 4.1 shows the structure of a SOAP message.

This listing shows that in SOAP the *<Envelope>* element is the root element, which may contain an optional *<header>* section and a mandatory *<body>* section. If a *<Header>* element is used, it must be the immediate child of the *<Envelope>* element, and precede the *<body>* element. The <body> element is shown to delimit the application-specific data. A SOAP message may have an XML declaration, which states the version of XML used and the encoding format, as shown in the snippet in Listing 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
   xmlns:env="http://www.w3.org/2003/05/soap-envelope">

   <env:Header> <!-- optional -->
    <!-- header blocks go here . . . -->
   </env:Header>

   <env:Body>
    <!-- payload or Fault element goes here . . . -->
   </env:Body>
</env:Envelope>
```

**Listing 4.1** Structure of SOAP message

A SOAP envelope can specify a set of encoding rules serializing the application defined XML data over the network. Both the provider and requestor must agree on the encoding rules (typically by downloading the same XML schema that defines them). To allow two or more communicating parties that agree on a specific encoding style for XML messages to express that agreement in the SOAP messages, they can use the global encoding Style attribute. This attribute mandates a certain encoding style (typically defined by the schema) for the element on which it appears and all child elements thereof. Encoding style is simply a set of rules that describes how data can be expressed (serialized to XML) by the parties to enable interoperability between their systems.

The SOAP specification allows the envelope tag to contain any number of additional, custom attributes.

**SOAP Header blocks**

A primary characteristic of the SOAP communications framework used by SOAs is an emphasis on creating messages that are as intelligence-heavy and self-sufficient as possible. This results in SOAP messages achieving a level of independence that increases the robustness and extensibility of this messaging framework qualities that are extremely important when relying on communication within the loosely coupled environment that Web services require.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.

Examples of the types of features a message can be outfitted with using header blocks include:

• processing instructions that may be executed by service intermediaries or the ultimate receiver

• routing or workflow information associated with the message

• security measures implemented in the message

• reliability rules related to the delivery of the message

• context and transaction management information

• correlation information (typically an identifier used to associate a request message with a response message)

• 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

**.Nodes**

• Although Web services exist as self-contained units of processing logic, they are reliant upon a physical communications infrastructure to process and manage the exchange of SOAP messages. Every major platform has its own implementation of a SOAP communications server, and as a result each vendor has labeled its own variation of this piece of software differently. In abstract, the programs that services use to transmit and receive SOAP messages are referred to as SOAP nodes.

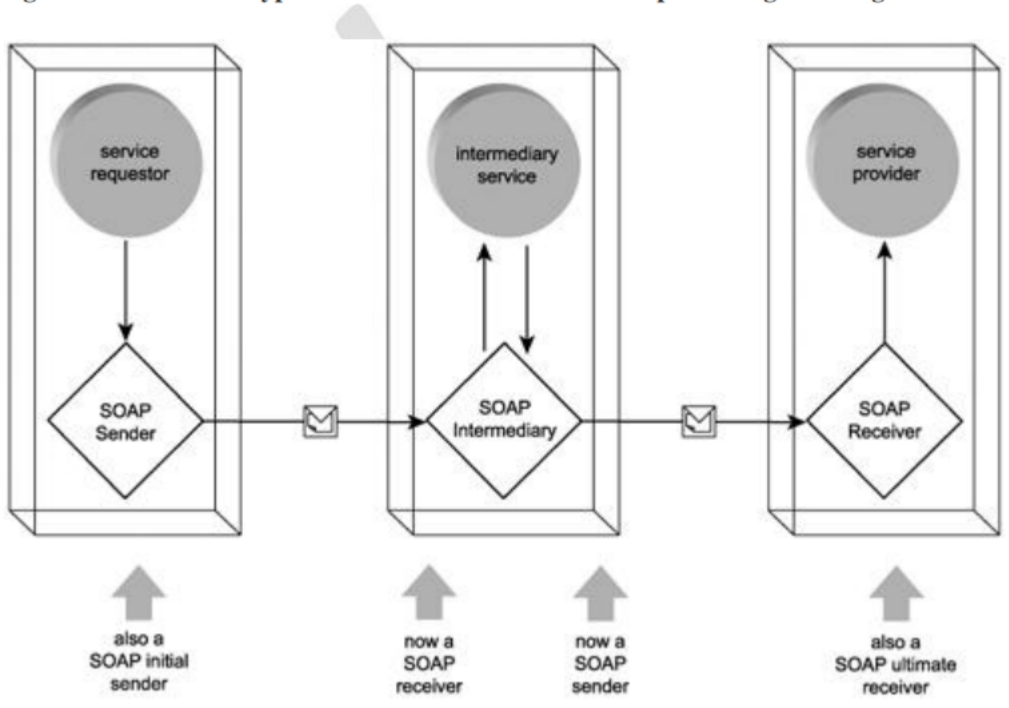
As with the services that use them, the underlying SOAP nodes are given labels that identify their type, depending on what form of processing they are involved with in a given message.Some examples of labels are:

- *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 messaging scenario.

**Figure 5.25. Different types of SOAP nodes involved with processing a message**



SOAP nodes acting as intermediaries can be classified as forwarding or active. When a SOAP node acts as a forwarding intermediary, it is responsible for relaying the contents of a message to a subsequent SOAP node. In doing so, the intermediary will often process and alter header block information relating to the forwarding logic it is executing. For example, it will remove a header block it has processed, as well as any header blocks that cannot be relayed any further.Active intermediary nodes are distinguished by the type of processing they perform above and beyond forwarding-related functions. An active intermediary is not required to limit its processing logic to the rules and instructions provided in the header blocks of a message it receives. It can alter existing header blocks, insert new ones, and execute a variety of supporting actions.

**ADVANTAGES AND DISADVANTAGES OF SOAP:** Read Topic 4.7

**Message Exchange Patterns:** Message exchange pattern (MEP) is a template that establishes a pattern for the exchange of messages between SOAP nodes.

There are two basic message exchange patterns described in the SOAP specification:

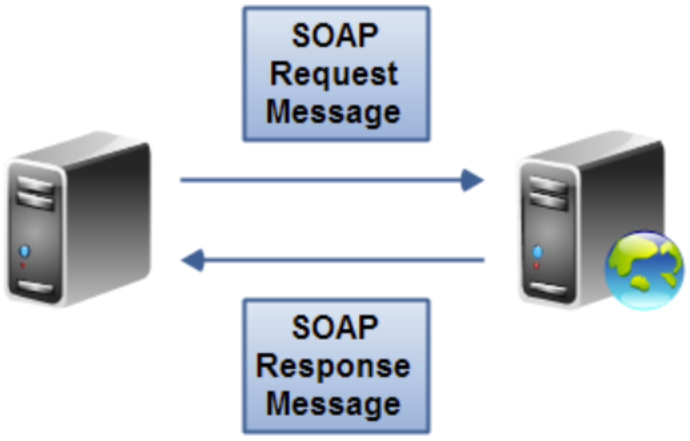
1. Request - Response
2. Response

These two message exchange patterns are described in the following sections.

**Request - Response**

In a request - response message exchange the SOAP client sends a SOAP request message to the service. The service responds with a SOAP response message.

A request - response message exchange patterns is necessary when the SOAP client needs to send data to the SOAP service, for the service to carry out its job. For instance, if the client request that data be stored by
the service, the client needs to send the data to be stored to the service.
**Here is an illustration of a request - response message exchange:**



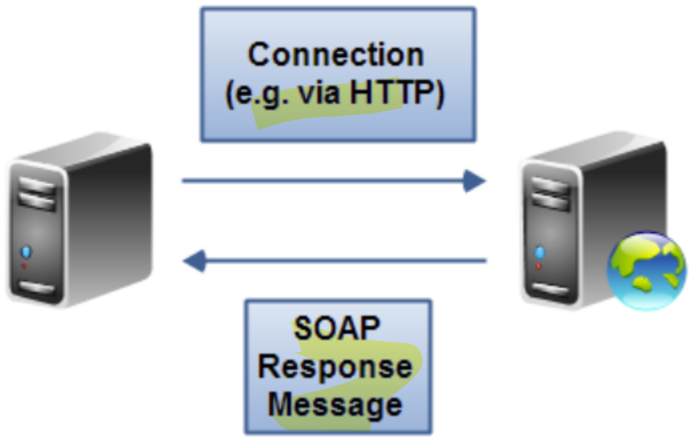
**A SOAP request - response message exchange.**

**Response**

In a response message exchange the SOAP client connects to the service, but does not send a SOAP message through. It just sends e.g. a simple HTTP request. The service responds with with a SOAP message.

This message exchange pattern is similar to how a browser communicates with a web server. The browser sends an HTTP request telling what page it wants to access, but the HTTP request does not carry any additional data. The web server responds with an HTTP response carrying the requested page.

**Here is an illustration of the response message exchange pattern:**



**A SOAP response message exchange.**

**Web Service Coordination (WS Coordination):** provides a framework for coordinating the actions of distributed applications via context sharing. WS-Coordination leverages a separate protocol aimed solely at outcome determination and processing.

**WS-Coordination**

It is a Web Services specification developed by BEA Systems, IBM, and Microsoft and accepted by OASIS Web Services Transaction TC in its 1.2 version. It describes an extensible framework for providing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support a number of applications, including those that need to reach consistent agreement on the outcome of distributed transactions.

The framework defined in this specification enables an application service to create a context needed to propagate an activity to other services and to register for coordination protocols. The framework enables existing transaction processing, workflow, and other systems for coordination to hide their proprietary protocols and to operate in a heterogeneous environment.

The Coordination service (or coordinator) is an aggregation of the following services:

· **Activation service**: Defines a CreateCoordinationContext operation that allows a CoordinationContext to be created. The exact semantics are defined in the specification that defines the coordination type. The Coordination service MAY support the Activation service.

· **Registration service**: Defines a Register operation that allows a Web service to register to participate in a coordination protocol. The Coordination service MUST support the Registration service.

· A set of **coordination protocol** services for each supported coordination type. These are defined in the specification that defines the coordination type.

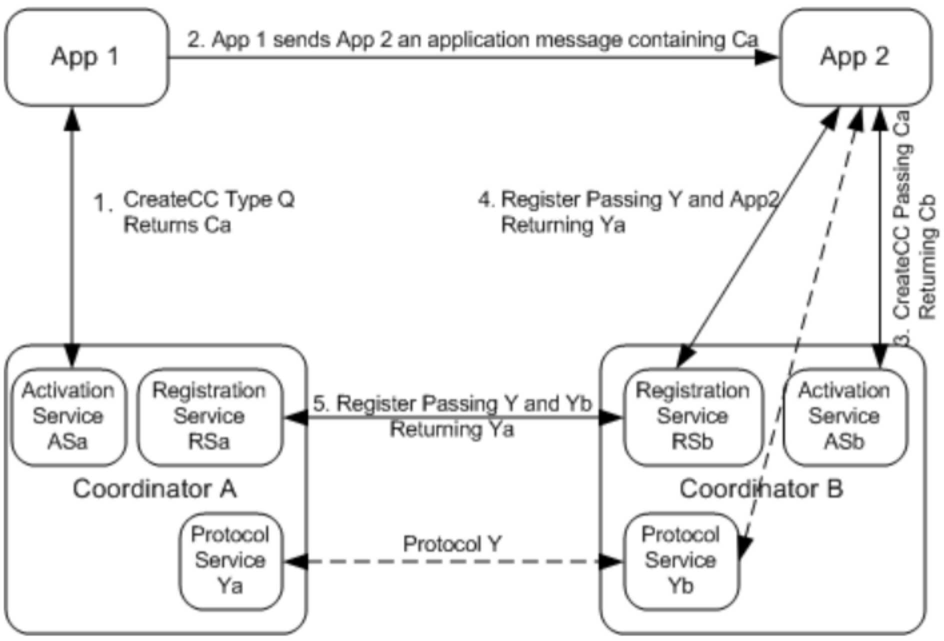
Figure 2 illustrates an example of how two application services (App1 and App2) with their own coordinators (CoordinatorA and CoordinatorB) interact as the activity propagates between them. The protocol Y and services Ya and Yb are specific to a coordination type, which are not defined in this specification.

1. App1 sends a CreateCoordinationContext for coordination type Q, getting back a Context Ca that contains the activity identifier A1, the coordination type Q and an Endpoint Reference to CoordinatorA's Registration service RSa.
2. App1 then sends an application message to App2 containing the Context Ca.
3. App2 prefers to use CoordinatorB instead of CoordinatorA, so it uses CreateCoordinationContext with Ca as an input to interpose CoordinatorB. CoordinatorB creates its own CoordinationContext Cb that

contains the same activity identifier and coordination type as Ca but with its own Registration service RSb.

4. App2 determines the coordination protocols supported by the coordination type Q and then Registers for a coordination protocol Y at CoordinatorB, exchanging Endpoint References for App2 and the protocol service Yb. This forms a logical connection between these Endpoint References that the protocol Y can use.
5. This registration causes CoordinatorB to decide to immediately forward the registration onto CoordinatorA's Registration service RSa, exchanging Endpoint References for Yb and the protocol service Ya. This forms a logical connection between these Endpoint References that the protocol Y can use.

**Figure 2: Two applications with their own coordinators**


App 1
2. App 1 sends App 2 an application message containing Ca
App 2
1. CreateCC Type Q Returns Ca
4. Register Passing Y and App2 Returning Ya
3. CreateCC Passing Ca Returning Cb
Activation Service ASa
Registration Service RSa
Coordinator A
Protocol Service Ya
5. Register Passing Y and Yb Returning Ya
Protocol Y
Registration Service RSb
Activation Service ASb
Coordinator B
Protocol Service Yb

**Activation Service:**creates a new activity and returns its coordination context.

An application sends:

CreateCoordinationContext

The activation service returns:

CreateCoordinationContextResponse

### CreateCoordinationContext

This request is used to create a coordination context that supports a coordination type (i.e., a service that provides a set of coordination protocols). This command is required when using a network-accessible Activation service in heterogeneous environments that span vendor implementations. To fully understand the semantics of this operation it is necessary to read the specification where the coordination type is defined (e.g. WS-AtomicTransaction).

### CreateCoordinationContextResponse

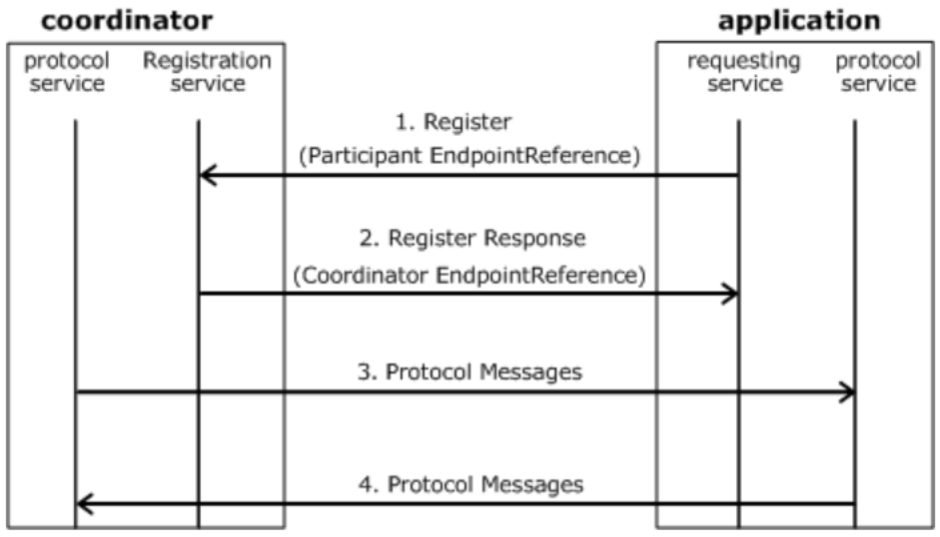
This returns the CoordinationContext that was created.

Once an application has a coordination context from its chosen coordinator, it can register for the activity. The interface provided to an application registering for an activity and for an interposed coordinator registering for an activity is the same.

The requester sends:Register

The coordinator's registration service responds with:Registration Response

Figure 3: The usage of Endpoint References during registration



In Figure 3, the coordinator provides the Registration Endpoint Reference in the CoordinationContext during the CreateCoordinationContext operation. The requesting service receives the Registration service Endpoint Reference in the CoordinationContext in an application message.

1.) The Register message targets this Endpoint Reference and includes the participant protocol service Endpoint Reference as a parameter.

2.) The RegisterResponse includes the coordinator's protocol service Endpoint Reference.

3. & 4.) At this point, both sides have the Endpoint References of the other's protocol service, so the protocol messages can target the other side.

## Register Message

The Register request is used to do the following:

- Participant selection and registration in a particular Coordination protocol under the current coordination type supported by the Coordination Service.
- Exchange Endpoint References. Each side of the coordination protocol (participant and coordinator) supplies an Endpoint Reference.

## RegistrationResponse Message

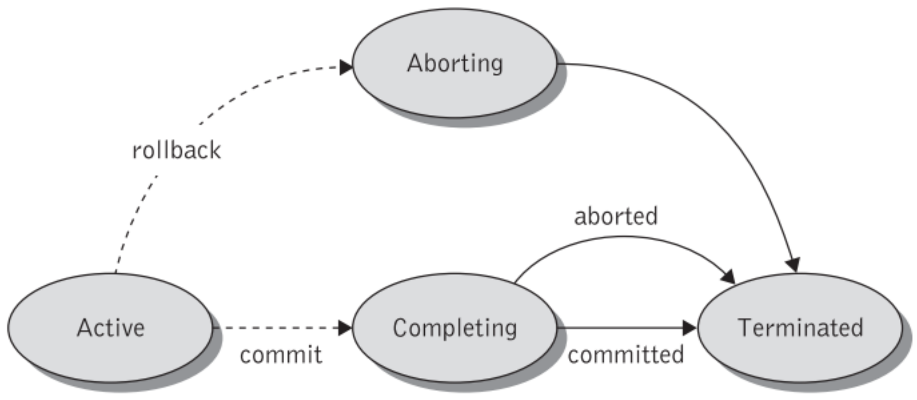
The response to the registration message contains the coordinator's Endpoint Reference.

The two transaction (coordination protocol) types that can be executed within the WS Coordination framework are: **Atomic transaction and Business Activity.**

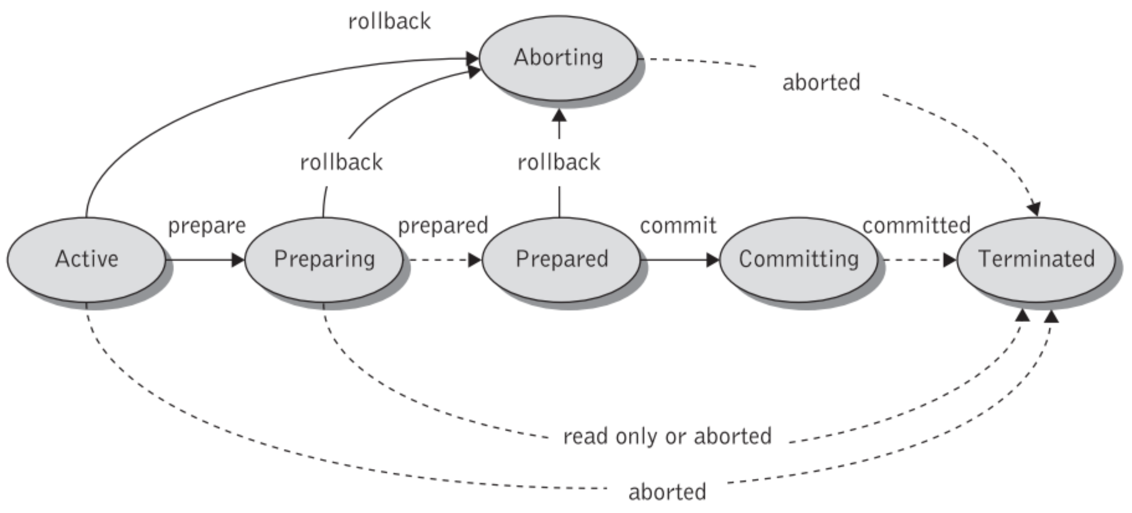
**Atomic transactions** are suggested for transactions that are short-lived atomic units of work within a trust domain, while **business activities** are suggested for transactions that are long-lived units of work comprising activities of potentially different trust domains. These are explained brelow:

**WS-ATOMIC TRANSACTIONS:** WS-AtomicTransaction [Cabrera 2005b] is usually confined to individual organizations where a client needs to consolidate operations across various internal applications. There are ***three kinds of transaction coordination protocols for atomic transactions*** as given below:

***1.Completion protocol***: This protocol is used by the application that controls the atomic transaction. When an application starts an atomic transaction, it establishes a coordinator that supports the WS-AtomicTransaction protocol. The application registers for this protocol and instructs the coordinator to commit or to abort the transaction after all the necessary application work has been done. The sequence of actions during this protocol is illustrated in Figure below where solid arcs signify coordinator-generated actions and dashed arcs signify participant-generated actions. The state transition diagram assumes that a coordinator will receive either a commit or rollback message from a participant. In this way, the application can detect whether the desired outcome is successful or not.



***2.Durable 2PC:*** This protocol is based on the classic two-phase commit with presumed abort technique, where the default behavior in the absence of a successful outcome is to roll back all actions in an activity. The durable 2PC is used to coordinate a group of participants that all need to reach the same decision, either a commit or an abort. After receiving a ***commit*** notification in the completion protocol, the root coordinator begins the ***prepare*** phase (phase 1) for durable 2PC participants. All participants registered for this protocol must respond with ***prepared or aborted*** notification. During the second (commit) phase, in case the coordinator has received a ***prepared*** response from the participants, it indicates a successful outcome by sending a ***commit*** notification to all registered participants. All participants then return a committed acknowledgement. Alternatively, if the coordinator has received an ***aborted*** response from one or more of the participants, it indicates a failed outcome by sending a ***rollback*** notification to all remaining participants. The participants then return ***aborted*** to acknowledge the result. It is illustrated in diagram below:



[illegible] Durable two-phase commit state transitions

**3.Volatile 2PC:**
Accessing durable data storage for the duration of a transaction,  leads to hard locking of local resources and performance bottlenecks. Alternatively, operating on cached copies of data can significantly improve performance. However, volatile data on which a transaction has worked will eventually have to be written back to back-end enterprise information systems such as databases, or ERP systems – where the state being managed by the applications ultimately resides – prior to this transaction committing. Supporting this requires additional coordination infrastructure. For example, to enable flushing of their updated cached state to the back-end servers, participants need to be notified before 2PC begins. In the WS-AtomicTransaction specification, this is achieved by volatile 2PC.

**Business Activities:**
Business activities govern long-running, complex service activities. Hours, days, or even weeks can pass before a business activity is able to complete. During this period, the activity can perform numerous tasks that involve many participants.

What distinguishes a business activity from a regular complex activity is that its participants are required to follow specific rules defined by protocols. Business activities primarily differ from the protocol-based atomic transactions in how they deal with exceptions and in the nature of the constraints introduced by the protocol rules. For instance, business activity protocols do not offer rollback capabilities. Given the potential for business activities to be long-running, it would not be realistic to expect ACID type transaction functionality. Instead, business activities provide an optional compensation process that, much like a "plan B," can be invoked when exception conditions are encountered

WS-BusinessActivity is a coordination type designed to leverage the WS-Coordination context management framework. It provides two very similar protocols, each of which dictates how a participant may behave within the overall business activity.

• ***The Business Agreement With Participant Completion*** protocol, which allows a participant to determine when it has completed its part in the business activity

• **The Business Agreement With Coordinator Completion** protocol, which requires that a participant rely on the business activity coordinator to notify it that it has no further processing responsibilities. Business activity participants interact with the standard WS-Coordination coordinator composition to register for a protocol

**Business Activity Coordinator:**

When its protocols are used, the WS-Coordination controller service assumes a role specific to the coordination. This coordinator has varying degrees of control in the overall activity, based on the coordination protocols used by the participants.

During the lifecycle of a business activity, the business activity coordinator and the activity participants transition through a series of states. The actual point of transition occurs when special notification messages are passed between these services.

For example, a participant can indicate that it has completed the processing it was required to perform as part of the activity by issuing a completed notification. This moves the participant from an active state to a completed state. The coordinator may respond with a close message to let the participant know that the business activity is being successfully completed.

However, if things don't go as planned during the course of a business activity, one of a number of options are available. Participants can enter a compensation state during which they attempt to perform some measure of exception handling. This generally invokes a separate compensation process that could involve a series of additional processing steps. A compensation is different from an atomic transaction in that it is not expected to rollback any changes performed by the participating services; its purpose is generally to execute plan B when plan A fails.

Alternatively, a cancelled state can be entered. This typically results in the termination of any further processing outside of the cancellation notifications that need to be distributed.

What also distinguishes business activities from atomic transactions is the fact that participating services are not required to remain participants for the duration of the activities.

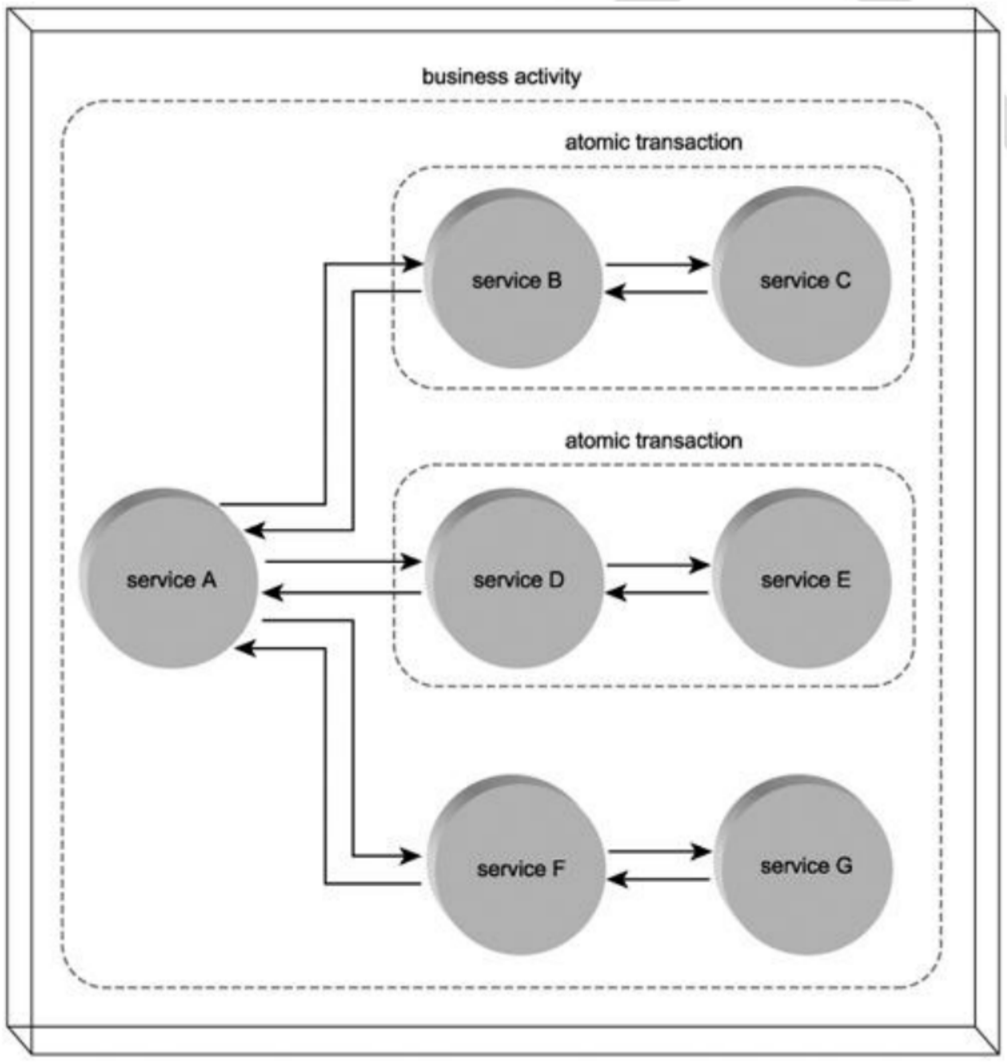
Because there is no tight control over the changes performed by services, they may leave the business activity after their individual contributions have been performed. When doing so, participants enter an exit state by issuing an exit notification message to the business activity coordinator.

These and other states are defined in a series of state tables documented as part of the WS-Business Activity specification. These tables establish the fundamental rules of the business activity protocols by determining the sequence and conditions of allowable states

**Business activities and atomic transactions:**

It is important to note that the use of a business activity does not exclude the use of atomic transactions. In fact, it is likely that a long-running business activity will encompass the execution of several atomic transactions during its lifetime

Figure 6.29. Two atomic transactions residing within the scope of a business activity.



## WS Orchestration and Choreography : Topic 9.6 (From Pg 328)

### Abstraction is the key

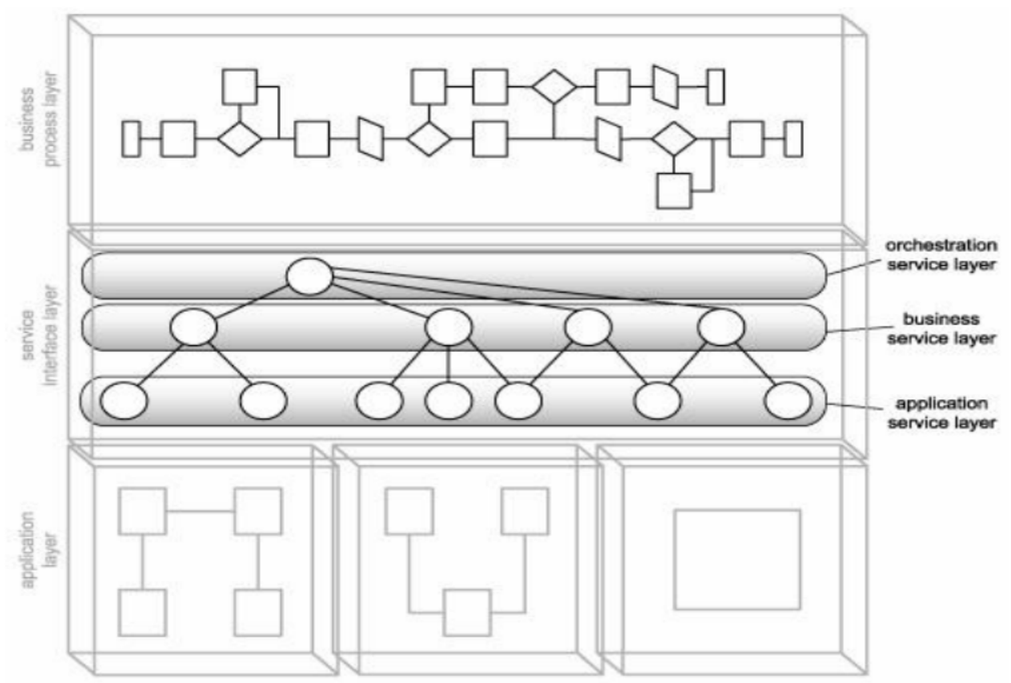
By leveraging the concept of composition, we can build specialized layers of services. Each layer can abstract a specific aspect of our solution, addressing one of the issues we identified. This alleviates us from having to build services that accommodate business, application, and agility considerations all at once.

The three layers of abstraction we identified for SOA are:

- the application service layer
- the business service layer
- the orchestration service layer

Each of these layers (also shown in Figure below) is introduced individually in the following sections.

**The three primary service layers.**



The application service layer establishes the ground level foundation that exists to express technology specific functionality. Services that reside within this layer can be referred to simply as application services. Their purpose is to provide reusable functions related to processing data within new or legacy application environments.

Application services commonly have the following characteristics:
• they expose functionality within a specific processing context
• they draw upon available resources within a given platform
• they are solution-agnostic
• they are generic and reusable
• they can be used to achieve point-to-point integration with other application services
• they are often inconsistent in terms of the interface granularity they expose
• they may consist of a mixture of custom-developed services and third-party services that have been purchased or leased

Typical examples of service models implemented as application services include the following:
• utility service
• wrapper service
Finally, an application service also can compose other, smaller-grained application services (such as proxy services) into a unit of coarse-grained application logic. Aggregating application services is frequently done to accommodate integration requirements. Application services that exist solely to enable integration between systems often are referred to as application integration services or simply integration services.
Integration services often are implemented as controllers.
Because they are common residents of the application service layer, now is a good time to introduce the wrapper service model. Wrapper services most often are utilized for integration purposes. They consist of services that encapsulate ("wrap") some or all parts of a legacy environment to expose legacy functionality to service requestors. The most frequent form of wrapper service is a service adapter provided by legacy vendors. This type of out-of-the-box Web service simply establishes a vendor-defined service interface that expresses an underlying API to legacy logic.

**Business Service Layer:**
They are responsible for expressing business logic through service-orientation and bring the representation of corporate business models into the Web services arena.

Application services can fall into different types of service model categories because they simply represent a group of services that express technology-specific functionality. Therefore, an application service can be a utility service, a wrapper service, or something else. However,Business services, on the other hand, are always an implementation of the business service model. The sole purpose of business services intended for a separate business service layer is to represent business logic in the purest form possible. This does not, however, prevent them from implementing other service models.For example, a business service also can be classified as a controller service and a utility service.

In fact, when application logic is abstracted into a separate application service layer, it is more than likely that business services will act as controllers to compose available application services to execute their business logic.

Business service layer abstraction leads to the creation of two further business service models.

• ***Task-centric business service:*** A service that encapsulates business logic specific to a task orbusiness process. This type of service generally is required when business process logic is not centralized as part of an orchestration layer. Task-centric business services have limited reuse potential.

• ***Entity-centric business service:*** A service that encapsulates a specific business entity (such as an invoice or timesheet). Entity-centric services are useful for creating highly reusable and business process-agnostic services that are composed by an orchestration layer or by a service layer consisting of task-centric business services (or both).

When a separate application service layer exists, these two types of business services can be positioned to compose application services to carry out their business logic

**Orchestration service layer:**
Orchestration is more valuable to us than a standard business process, as it allows us to directly link process logic to service interaction within our workflow logic. This combines business process modelling with service-oriented modeling and design. And, because orchestration languages (such as WS-BPEL)realize workflow management through a process service model, orchestration brings the business process into the service layer, positioning it as a master composition controller.

The orchestration service layer introduces a parent level of abstraction that alleviates the need for other services to manage interaction details required to ensure that service operations are executed in a specific sequence. Within the orchestration service layer, process services compose other services that provide specific sets of functions, independent of the business rules and scenario-specific logic required to execute a process instance.

All process services are also controller services by their very nature, as they are required to compose other services to execute business process logic. Process services also have the potential of becoming utility services to an extent, if a process, in its entirety, should be considered reusable. In this case, a process service that enables orchestration can itself be orchestrated.