Web Service Transactions

Objectives

- Understand the mechanics of a two-phase commit.
- Provide details about the WS-Coordination protocol and its place in a Web service transaction.
- Diagram the message flow of an Atomic Transaction, as described in the WS-Transaction protocol.
- Describe how a Business Activity transaction occurs within the context of the WS-Transaction protocol.

The files associated with this chapter are located in the following folders:
  ○ \{inetpub\wwwroot\}\WebServices\Transactions

NOTE This chapter has no lab.
Transaction Concepts

As long as applications are based on the single-user, single-table model, the need for something as complex as transactions is limited. But in the real world, the need to have applications access multiple databases and provide access to multiple simultaneous users is a requirement. And with this demand, the need for transactions becomes an imperative.

The purpose of a transaction is to ensure that two or more database updates occur simultaneously. The underlying premise is that many database operations are actually interrelated. For example, when an order is shipped, the status on the order is changed to “shipped” at the same time that the inventory quantity of the item being shipped is decreased. If one part of this update happens without the other, then the database would be put into an inconsistent state (also known as “wrong” to those of us who are not academics).

Any time that a discussion of transactions comes up, the term ACID will probably be used. ACID is a cute acronym that is used to remember the attributes that a transaction is expected to support.

- **Atomic:** All updates in the transaction must succeed or fail as a group. If any update fails, then the remaining updates must be rolled back. When a transaction is rolled back, all of the updated records are put back into the same state they were in before the transaction started.

- **Consistent:** The database must not be left in an unstable state. No update may be only partially completed.

- **Isolated:** No transaction is affected by any other transaction. Specifically, no application can see the results of a transaction until the transaction has been committed. For this reason, the updates associated with a transaction are usually kept in memory until the commit (the notification that the transaction can be made permanent) is received.

- **Durable:** The results of a transaction are permanent and can only be modified through another transaction. This attribute addresses the idea that a hardware failure cannot affect the outcome of a transaction.

Notice that at no point in the previous discussion was mention made that only one database can be affected. In fact a transaction can, and in many cases does, span database types and platforms. Given the enterprise space that Web services aspire to, it should not be surprising that a standard is emerging to allow Web services to participate in a transaction. The protocols involved are called WS-Coordination and WS-Transaction. These protocols provide support for two different coordination types: Atomic Transactions (AT) and Business Activity (BA).
Atomic Transactions

An Atomic Transaction coordinates activities that have a short duration and are executed within a small set of domains. They are called Atomic Transactions not because of their nuclear properties, but because they are committed or rolled back as a group. This portion of the specification also defines the protocols necessary to integrate WS-Transaction into existing transaction processing systems.

Business Activity

A business activity coordinates activities for a different set of transactions. The transactions supported by the BA are longer in duration and more complicated in processing logic than Atomic Transactions. Instead of just an all or nothing proposition, the BA allows logic to be applied when an exception caused by business rules is raised. Instead of keeping data resources locked for the duration of the transaction, updates are applied immediately and permanently. This means that the additional logic included in the BA must support the conceptual rolling back of a broken transaction.
Two-Phase Commit

Throughout this discussion of transactions, it is important to understand the concepts behind a two-phase commit (2PC). It is not only one of the commit types supported by the above protocols, it is also the most commonly used mechanism to support distributed transactions. That is, transactions that affect more than one database or operating environment.

The basic premise of a 2PC is that one coordinating process is responsible for the entire transaction. At the time that a commit is requested, the transaction coordinator notifies each of the participants in the transaction. The participants then vote to either commit or abort the transaction (see Figure 1).

![Figure 1. Phase 1 of the two-phase commit.](image)

Once the transaction coordinator has either received votes from all of the participants or a pre-determined period of time has elapsed, the envelope is opened. If any participant votes against the commit or no vote is received from a participant within a specified time, the entire transaction is rolled back. Otherwise a commit request is sent to each of the participants (see Figure 2).
Two-Phase Commit

By using this mechanism, the transaction coordinator can utilize the built-in functionality of each database to construct a cross-platform, atomic update. Are there places where this mechanism can fall down? In a simple incarnation the answer is yes. It’s possible that after voting for a commit and before the global commit is received, that one of the participants could fail. But more sophisticated coordinators make use of acknowledgements to ensure that updates meet the ACID test even in the case of hardware or network failures.
WS-Coordination

As could be deduced from the previous section, implementing transactions among Web services requires that someone or something be in charge. That “thing” becomes responsible for monitoring the status of the updates that are being made by the other participants and deciding, based on input from the participants, what the fate of the transaction is. The purpose of the WS-Coordination protocol is to define the messages that are exchanged between the participants and the coordinator.

CoordinationContext

Within the world of WS Coordination, a context is a construct that is used to associate two (or more) disparate processes together. By embedding context information in a particular message, the requester tells the Web service to operate as part of a larger structure, such as a transaction. To play with the big boys in the technology schoolyard, the Web service needs to be able to cede control of when the completion of the function occurs to the coordinator of the larger structure. To put it another way, the Web service needs to implement both an “attempt” and “commit” version of each function. The “attempt” version performs the uncommitted updates while the “commit” version makes the update permanent.

The CoordinationContext type starts out being derived from the Context type. From this base, two attributes are inherited. The Identifier attribute is a URI that uniquely identifies a particular context. This identifier is used to determine which messages are part of which transactions. The Expires attribute describes when the context is terminated in the absence of any prior requests. The ability to “timeout” a context is an optional piece of functionality that is not implemented in every coordinator. So before designing to use timeouts, be sure that the coordinator supports them.

On top of these two inherited attributes, the CoordinationContext adds two additional elements. The CoordinationType identifies the protocol being used in this specific coordination process. At the moment, the coordination type can be either Atomic Transaction or Business Activity. In the section of this chapter covering each of these protocols, an example of the URI for each can be found. The RegistrationService attribute provides the URI to be contacted in order to register for participation in a transaction.

The WS-Coordination specification allows the CoordinationContext to be extended to support custom functionality. A common extension, especially when used in conjunction with ADO transactions, would be to include an IsolationLevel attribute that corresponds to the similar field in an ADO Connection object.
Below is an example of a CoordinationContext element, complete with the custom extension of the IsolationLevel.

```xml
<wscoor:CoordinationContext
  xmlns:ext="http://appdev.com/WebServices">
  <wsu:Expires>
    2002-10-01T16:35:00.000-05:00
  </wsu:Expires>
  <wsu:Identifier>
    http://appdev.com/WebServices/tx1234
  </wsu:Identifier>
  <wscoor:CoordinationType>
    http://schemas.xmlsoap.org/ws/2002/08/wstx
  </wscoor:CoordinationType>
  <wscoor:RegistrationService>
    <wsu:Address>
      http://appdev.com/WebServices/WSCRegistration
    </wsu:Address>
  </wscoor:RegistrationService>
  <ext:IsolationLevel>
    RepeatableRead
  </ext:IsolationLevel>
</wscoor:CoordinationContext>
```

The wsu namespace contains the definition for the Context type from which the CoordinationContext is derived. For this reason, it is used with the Identifier and Expires tags in the preceding example. The Coordination Type for this example is an Atomic Transaction.

## The Coordinator

Naturally, at the heart of the protocol is a coordinator. In WS-Coordination, there are three functions that need to be performed by the coordinator. They are as follows:
An Activation Service

The first functionality used in the coordinator is the Activation service. Through this service, the initiating application can create the CoordinationContext that will be used throughout the transaction. For each subsequent message that is to participate in the coordination, the CoordinationContext must be passed in the Header section of the SOAP message.

A Registration Service

Through the Registration service exposed by the Coordinator another process can express interest in a particular transaction. When invoking the Register method on the coordinator, the participant includes information such as the level of interest in the transaction and the address where they (the registering process, that is) can be notified.

In the WS-Coordination specification, how the level of interest is defined is purposefully vague. Still, it is possible to impose a little reality for the purpose of clarity. For example, a process might want to actually take part in the transaction, be notified upon its completion, or be notified immediately before the vote is taken. All of these are possible, but the details of how to do so are hidden in the WS-Transaction specification.

A Coordination Service

The Coordination service as implemented by the Coordinator does what has been expected of WS-Coordination all along. Manage the status and completion states of a transaction. The details of how this actually takes place depends on the coordination protocol that is used. What’s a coordination protocol you say? Glad you asked. Again, vagueness runs rampant in the specification, but the short answer is that a coordination protocol is a set of rules that define how transactions are processed. Two possible coordination protocols (coincidentally called Atomic Transaction and Business Activity) are specified within a WS-Transaction.
As was just mentioned, from the WS-Coordination protocol comes the need to define specific coordination protocols that support different types of coordination. The activities that are required of the coordinator are purposefully vague. The reason for the vagueness is that there is no requirement that WS-Coordination be used exclusively for transactions. However, especially while the standard is in its infancy, it is easier to think of these two protocols as being intertwined. Once an understanding of the concepts is achieved, extending them to other situations will be easier.

Within WS-Transaction is the definition of two coordination protocols that can be used with WS-Coordination. They are called Atomic Transaction and Business Activity. And each one warrants closer examination of the messages that get passed back and forth, as they address different classes of business situations.

**Atomic Transaction**

The defining characteristic of an Atomic Transaction is the all-or-nothing nature of any updates. This part of the specification defines the workings of a transaction in the traditional, database-oriented sense (in case that wasn’t clear from the name). In this situation, the coordinator must have a great deal of control over the behavior of the participants.

When a participant registers with the coordinator, one of five participant types can be specified. These categories identify the type and timing of the messages that will be received from the coordinator. The participant types are:

- **Completion**: Any participant that registers for Completion can tell the coordinator that the entire transaction can be committed or rolled back. Typically, there is only one participant with this registration, that being the initiating application. However, it is possible to have more than one participant be able to start the transaction commit/rollback processing. A status code indicating the final outcome is returned to the participant that starts the process.

- **CompletionWithAck**: This participant type is the same as Completion, except that the coordinator must wait for the participant that initiated the commit/rollback to acknowledge receipt of the outcome status.

- **PhaseZero**: Any participant that registers for Phase Zero will receive a notification immediately before the 2PC protocol is initiated. This would normally be used by a process that caches its data and needs an opportunity to flush any changes before the 2PC process takes place.
• **2PC:** This participant type is used for any process that is involved in the updates that are part of the transaction. All 2PC participants receive notification when the transaction is to be committed and submit their vote to continue. They then receive a second notification indicating a final commit or rollback instruction.

**NOTE** As a point of interest, the protocol is smart enough to reduce processing where necessary. If there is only one 2PC participant, then the entire commit/rollback decision (as would normally be determined by the coordinator) is left up to that participant.

• **OutcomeNotification:** This participant type is used by a process that wants to be notified of the final outcome of the transaction. It could be used by an ancillary process that needs to unlock resources that had been held as part of the transaction. As an additional complication, it is possible for one process to register for multiple participation types. In fact, there is nothing (besides rational thought, that is) that prevents one process from registering for all of the participant types.

### Sample Message Flow

To gain a clearer understanding of the processing involved in an Atomic Transaction, an example is in order. Assume that a travel agent receives an itinerary from a client. The client needs an airline and hotel reservation. To meet this need, the agent communicates with the reservation system for each company. If either of these companies can’t satisfy the request, the entire itinerary must be cancelled. Of course, for this example, all of the information exchanges will be electronic between various Web services.

### Agent Starts the Transaction

Figure 3 shows the messages that are passed in the first stage of the transaction.
Figure 3. The initial process flow for an Atomic Transaction.

The Agent application starts off the transaction by requesting a CoordinationContext from the Coordinator. This is done by sending a CreateCoordinationContext message to the Coordinator (Step 1). The SOAP messages in this example (shown below) have been simplified for brevity. Also, assume that the wscoor namespace refers to "http://schemas.xmlsoap.org/ws/2002/08/wscoor" and the wsu namespace refers to "http://schemas.xmlsoap.org/ws/2002/07/utility".

```
<envelope>
  <body>
    <wscoor:CreateCoordinationContext>
      <wscoor:ActivationService>
        <wsu:Address>
          http://appdev.com/WebServices/WSCActivation
        </wsu:Address>
      </wscoor:ActivationService>
      <wscoor:RequesterReference>
        <wsu:Address>
          http://appdev.com/wscoor/TA
        </wsu:Address>
      </wscoor:RequesterReference>
      <wscoor:CoordinationType>
        http://schemas.xmlsoap.org/ws/2002/08/wstx
      </wscoor:CoordinationType>
    </wscoor:CreateCoordinationContext>
  </body>
</envelope>
```
The ActivationService element represents the URL of the Activation service for the Coordinator. This address is where the SOAP message will be posted. The RequesterReference element contains a URI from which the message is being sent. Finally, the CoordinationType contains a URI indicating the type of transaction coordination required. The URI used above is for an Atomic Transaction (naturally).

The Coordinator responds by sending a message containing a CoordinationContext.

Next, the Agent registers with the Coordinator for the Completion protocol (Step 2). A Register message is sent to the Coordinator:

```xml
<envelope>
  <header>
    <!-- Coordination Context goes here -->
  </header>
  <body>
    <wscoor:Register>
      <wscoor:RegistrationService>
        <wsu:Address>
          http://appdev.com/WebServices/WSCRegistration
        </wsu:Address>
      </wscoor:RegistrationService>
      <wscoor:RequesterReference>
        <wsu:Address>
          http://appdev.com/wscoor/TA
        </wsu:Address>
      </wscoor:RequesterReference>
      <wscoor:ProtocolIdentifier>
        http://schemas.xmlsoap.org/ws/2002/08/wstx/Completion
      </wscoor:ProtocolIdentifier>
      <wscoor:ParticipantProtocolService>
        <wsu:Address>
          http://appdev.com/wscoor/TA/Completion
        </wsu:Address>
      </wscoor:ParticipantProtocolService>
    </wscoor:Register>
  </body>
</envelope>
```
Include in the header of this message is the CoordinationContext returned in the first step. The message also includes the URI to which the message is being sent (in the RegistrationService element) and the URI, which can be used by the coordinator for notifications (the ParticipantProtocolService element). The ProtocolIdentifier element contains the base name of the coordination type (as defined in the CreateCoordinationContext message) along with the protocol type. This last element of the ProtocolIdentifier corresponds to the participant types defined in the Atomic Transaction section above. In this example, the application is a participant in the Completion protocol.

The Agent application now sends a request to both the Air and Hotel reservation services (Step 3). In the header of these messages, the CoordinationContext is placed, allowing the transaction to be extended to these services as well.

For the next step in the process, only the Airline reservation system will be diagrammed. Assume that the Hotel system is conceptually identical.

In Figure 4, the reservation message sent by the Agent in Step 3 is received by the Airline Reservation Web Service. The CoordinationContext object is detected in the message header, but the Airline Web Service doesn’t want to utilize a remote coordinator for the transaction. So a CreateCoordinationContext message that includes the incoming Coordination Context is sent to the local Airline Coordinator (Step 4). The response message includes a CoordinationContext that contains the same requester reference and coordination type as the original message. Only the address used by the Web service to communicate with the coordinator has been changed. From this point forward, all of the WS-Transaction messages will be sent from the Airline Reservation Web Service to the local Airline coordinator instead of the Agent coordinator. This means that all of the messages that get passed from participant to coordinator will stay within the Airline Reservation network.
Figure 4. The airline reservation portion of an Atomic Transaction.

An immediate question is why add the extra layer of a local coordinator to this process. Two reasons immediately spring to mind. One is speed—sending messages to the local coordinator is faster and potentially more reliable. This is the primary, but by no means the only, reason.

Every participant in the transaction that updates data must register with the transaction coordinator to use the 2PC protocol. If a participant from the Airline Reservation side were to register with the Agent coordinator, it is possible that information about the internals of the reservation system could be deduced. At the very least, the registration includes details that should not leave the Airline reservation system. So the Airline Coordinator is set up as a proxy for all of the updates associated with the reservation system.

Step 5 has the Airline Reservation Web Service registering with the Airline Coordinator as a 2PC participant, as would any other process on the Airline Reservation side that needs to be included in the transaction. This registration triggers the Airline Coordinator to register with the Agent Coordinator as a 2PC participant (Step 6). In fact, for every different participant type inside the Airline Reservation system, the Airline Coordinator will register with the Agent Coordinator as that participant.

**Agent Commits the Transaction**

At some point in the future, the Agent decides to commit the transaction.

While Figure 5 makes the commit process seem complicated, what with all of those numbers, in reality it is fairly straightforward. The Agent starts it off by sending a Commit message to the Coordinator (Step 1 with an example message shown in the following code).
Figure 5. The commit process flow for an Atomic Transaction.

```xml
<envelope>
  <header>
    <!-- Coordination Context goes here -->
  </header>
  <body>
    <wscoor:Commit>
      <wscoor:TargetProtocolService>
        <wsu:Address>
          http://appdev.com/WebServices/WSCCoordinator
        </wsu:Address>
      </wscoor:TargetProtocolService>
      <wscoor:SourceProtocolService>
        <wsu:Address>
          http://appdev.com/wscoor/TA/Completion
        </wsu:Address>
      </wscoor:SourceProtocolService>
    </wscoor:Commit>
  </body>
</envelope>
```

The two elements of the Commit message are the TargetProtocolService and the SourceProtocolService. The Target element contains the URI for the Coordinator, while the Source element contains the URI for the Agent. The next step depends on how the initial transaction was constructed.
If any participants had registered for the PhaseZero protocol, they would be notified of a pending Commit. But that is not the case in this example. Instead, all of the 2PC protocol registrants are sent a Prepare message. Different than a Commit, the Prepare message asks for the participants’ opinion on whether the transaction should be committed or aborted. Since the Airline Coordinator registered as a 2PC participant, it would receive the Prepare message (Step 2).

Now the Airline Coordinator is acting as the local proxy for some number of transactions associated with the reservation. After receiving the Prepare message, it retransmits the message (replacing the CoordinationContext object with one referring to itself) to all of its 2PC participants (Step 3). Once all of the responses have been received (Step 4), the Airline Coordinator makes a commit/abort decision based on the same criteria used by the Agent Coordinator (i.e., if any participant votes against the transaction, then the entire transaction is aborted). The result of its deliberations is transmitted back to the Agent Coordinator (Step 5).

The Agent Coordinator now has all of the information needed to determine if the transaction should be committed. Regardless of the outcome, a second message is sent to all of the 2PC participants directing them to commit or abort the transaction as appropriate (Step 6). The Airline Coordinator, again acting as a proxy, propagates that message to its participants (Step 7). The status for each commit/abort is returned to the Airline Coordinator (Step 8), then on to the Agent Coordinator (Step 9) and back to the initiating process (Step 10).

**Business Activity**

While Atomic Transactions work well for business tasks that are relatively tightly coupled, they are not perfect for a loosely-coupled environment where the duration of the transaction might be long. For most Atomic Transactions, a critical element for success is that any necessary resources remain locked until the transaction is either committed or aborted. But the longer the duration of the transaction, the more likely it is that other transactions will be affected. And in the Web services world, where the participants in a transaction are external to the provider and accessed across an inherently less reliable connection, keeping scarce resources on hold is not always an acceptable proposition.

**Differences Between BA and AT**

Before considering the message flow in a Business Activity, it is useful to look at the differences between the two protocols as shown in Table 1.
**Table 1. Differences between BA and AT protocols.**

<table>
<thead>
<tr>
<th>Business Activity</th>
<th>Atomic Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses lots of resources</td>
<td>Uses few resources</td>
</tr>
<tr>
<td>Can use Atomic Transactions</td>
<td>Complete in and of itself</td>
</tr>
<tr>
<td>Loss of state information is important</td>
<td>Abort and retry a cheap operation</td>
</tr>
<tr>
<td>Mistakes can be costly</td>
<td>Abort and retry a cheap operation</td>
</tr>
<tr>
<td>Partial transactions allowed</td>
<td>Results are isolated</td>
</tr>
<tr>
<td>Response time could be measured in days</td>
<td>Response time in seconds</td>
</tr>
</tbody>
</table>

Given these differences, the important elements when designing the Business Activity proposal are:

- Record all state transitions onto stable storage (i.e., tape, CD) or in a redundant storage system. This allows for auditing and rebuilding lost data in case of a catastrophic problem.

- Acknowledge all request messages. This allows problems to be detected earlier in the process. A Business Activity frequently includes interactions with other organizations. If, for example, an order gets misplaced, there are real costs associated with correcting it. So the earlier a mistake is noticed, the less real costs will be incurred.

- Requests and responses are defined as two operations, rather than two sides of the same operation. This design accommodates the possibility that steps within a Business Activity could be days in length.

- If multiple identical requests are received before the first response is returned, then only one request will be processed. This allows for an impatient process to retransmit requests (after a reasonable period of time, naturally) without impacting the result of the transaction.

As with an Atomic Transaction, a Business Activity has a number of protocols that are described in the specification. But before these protocols can be described, a couple of concepts need to be defined.

**Scope**

As part of a discussion on business activities, the term scope will almost certainly come up. The “official” definition is much more complicated than the conceptual one. For a Business Activity, a scope is a group of functions that get executed by one or more Web services. One of the defining features of the scope is that the outcome of the scope must be agreed to by all of the members.

For example, a purchasing application might request a price quote from five different vendor Web services. Each pricing request would be considered a separate scope, in that any number of transactions could be taking place within the vendor Web service with a single outcome (the price) as the result.
There is also a concept of hierarchy within the use of scope in a Business Activity. The purchasing application has a parent/child relationship with the pricing requests. The “parent” controls which “children” get executed and summarizes the individual results into a single value.

Also part of the parent/child relationship is exception handling. A parent is notified when one of their children raises an error. This allows the parent to determine the appropriate course of action. Continuing with the purchasing/pricing example, if one of the vendor requests times out, the parent application is notified and can decide to try again or simply disqualify that vendor from the final consideration.

Now back to the protocols.

- **BusinessAgreement**: Exists between a parent and a child within a particular scope. The child actually registers with the parent, so participation flows uphill. In this protocol, it is up to the child to know when all of the work for a Business Activity has been completed.

- **BusinessAgreementWithComplete**: Basically the same as the BusinessAgreement except that the parent tells the child when all of the work within a particular scope has been completed.

**BusinessAgreement Protocol**

Within the BusinessAgreement protocol, there are a couple of different types of children that can be created. The first is the simplest, and we’re calling it a One-Off Child (see Figure 6).

![Figure 6. A One-Off Child message flow.](image)

For all of these examples, the Parent launches the Child (or the Child is just always running) and it is up to the Child to register with the Parent to join the Business Activity. For this type of child, whatever activity is being performed is one-time only or irreversible.

The pricing requests that have already been described would be an example of a One-Off Child. The child process is responsible for querying the vendor and the processing the information before providing the data to the parent. The information returned by the child is non-volatile and no updates are performed,
so there is no reason for the child to continue. It notifies the parent with the Exited message and terminates.

But not every child process has that “use once” characteristic. A second type of child is called an On-Going Child (see Figure 7).

![Figure 7. An On-Going Child message flow.](image)

As with a One-Off Child, the Parent launches the Child followed by a Registration for the BusinessAgreement protocol. When the Child finishes its work, a Completed message is sent to the Parent. However, unlike with the One-Off Child, the On-Going Child doesn’t drop out of the Business Activity.

At some point in the future, when the Parent is finished with its work, it sends a Close message to the Child. The Child then returns a Closed message and the activity is finished. This type of child is used in a situation where the child has performed some updates. Perhaps the child has actually placed an order with an outside vendor or updated an inventory quantity. Regardless of the update, the child needs to stay active in case the parent needs it.

As an example, consider the case when the Business Activity doesn’t complete normally. In this case, the Parent sends a Compensate message to the Child. This tells the Child that all of the updates associated with this Business Activity need to be logically rolled back. Notice the use of the adverb “logically.” In some cases, it will not be possible to put the state of the database back to exactly the way it was before the transaction started.
Interaction with outside services (such as placing an order) might make that impossible. Instead, the compensation process includes whatever updates and notifications might be necessary to nullify the previous work.

**BusinessAgreementWithComplete Protocol**

The BusinessAgreementWithComplete protocol follows the same message flow as the BusinessAgreement protocol with one minor exception. With BusinessAgreement, the Child process “knew” when all of its work for a transaction was finished. This is not true for BusinessAgreementWithComplete. So instead of the Child sending the Parent an unexpected Completed message, the Parent issues a Complete message to the Child when its task is done. The remainder of the flow is unchanged.

**Security Considerations**

Although some of these same security issues relate to Atomic Transactions as well, the Business Activity, due to the disconnected and extended processing structure, is particularly vulnerable to a number of security problems.

First, remember that messages can be both modified and forged. As such, it makes good sense to utilize a standard that allows for message encryption and signing. The WS-Security protocol is a good fit here, but it is not the only solution by far. The SOAP specification allows for customized encryption techniques to be used. The important concept is to use some encryption, regardless of the choice of implementation.

A rarely considered attack point for Business Activity environments in particular is something known as a **replay attack**. A valid message is intercepted and then retransmitted to the Web service multiple times. To be honest, this is the same behavior that would be exhibited by a BA participant that was impatiently waiting for a response. That is what makes the attack difficult to detect. So instead of detection, the solution is to mitigate the problem, ensuring that each incoming request receives a time stamp. Then, when a Web service sees multiple messages differing only in their time stamp, all but the first message can be safely ignored.
Windows Transactions

All of this talk about transactions and you haven’t learned how Web services work within Windows transactions. Unfortunately, the answer is not very well. At least as part of an existing transaction, that is.

The participation level of a Web service in a transaction is indicated by the TransactionOption attribute of the WebMethod tag.

```vbnet
WebMethod(TransactionOption:=TransactionOption.Required) _
Public Sub UpdateData(Id As String, FieldList As _
    Collection, FieldValues As Collection)
    . . .
End Function
```

While the TransactionOption attribute actually has five possible values, there are really only two that matter. The problem is that a Web service can only be the root object in a transaction. As such, it cannot participate in an existing transaction. So setting the TransactionObject to Required results in a new transaction being created every time. This is the same functionality that is achieved with the RequiredNew option. On the flip side, the other choices (Disabled, NotSupported, and Supported) do not generate a new transaction when called.

So why is a Web service so ornery when it comes to Windows Transactions? The answer lies in what has already been mentioned about transactions. In order to have a transaction, there needs to be a context. For .NET transactions, that context is carried around automatically, so there is no work required by the developer. But sending a message to a Web service entails passing information to a system that could be halfway around the world. A quick mental review of everything discussed about Web services leads to the conclusion that .NET transaction context is not part of the specification. Is there anything in the above source code to indicate that the transaction context is being sent? Or that it would be read and understood by the receiver? Or that the receiving platform is Windows 2000? The answer to all of these questions is no.

So even if the Web service is running on the server sitting at the developer’s feet, there is nothing that can be done to bring that Web service method into an existing COM+ transaction. Only create new ones and move on from there.
Summary

- All transactions must support the ACID set of functionality.
- A two-phase commit is used to implement transactions that span databases and platforms.
- The WS-Coordination protocol defines the messages that are exchanged between participants and a coordinator in a transaction.
- The CoordinationContext structure is used to associated SOAP messages with a single transaction.
- The WS-Transaction protocol defines two coordination protocols that can be used in conjunction with WS-Coordination to implement a distributed transaction.
- An Atomic Transaction is a coordination protocol that is used for transactions of short duration.
- A Business Activity is a coordination protocol that is used for transactions that are longer in duration and includes business logic to perform logical roll backs.
- A Web service must be the root object in order to participate in a Windows-based transaction.
Questions

1. Name the four attributes that a transaction must support.

2. Name the structure that is used to link SOAP messages into a single transaction.

3. What three functions are supported by the coordinator in the WS-Coordination specification?

4. List the five participation types as defined in an Atomic Transaction.

5. Which participant type is responsible for initiating the commit in an Atomic Transaction?

6. In a Business Activity, which term describes a set of transactions that result in a single outcome?

7. What is the main difference between a One-Off Child and an On-Going Child in a Business Activity?
Answers

1. Name the four attributes that a transaction must support.
   Atomic, Consistent, Isolated, and Durable

2. Name the structure that is used to link SOAP messages into a single transaction.
   CoordinationContext

3. What three functions are supported by the coordinator in the WS-Coordination specification?
   Activation, Registration, Coordination

4. List the five participation types as defined in an Atomic Transaction.
   Completion, CompletionWithAck, PhaseZero, 2PC, OutcomeNotification

5. Which participant type is responsible for initiating the commit in an Atomic Transaction?
   Completion

6. In a Business Activity, which term describes a set of transactions that result in a single outcome?
   Scope

7. What is the main difference between a One-Off Child and an On-Going Child in a Business Activity?
   A One-Off child terminates automatically at the completion of its work, while the On-Going Child needs to be notified by the coordinator.