1 Reliable Messaging

Reliable Messaging defines an interoperable protocol such that the two Messaging Service Handlers (MSH) ~~operated by a *From Party* and a *To Party*~~ can “reliably” exchange messages that are sent using “reliable messaging” semantics.

“Reliably” means that the *From Party* can be highly certain that the message sent will be delivered to the *To Party*. If there is a problem in sending a message then the sender resends the message until either the message is delivered, or the sender gives up. If the message cannot be delivered, for example because there has been a catastrophic failure of the *To Party’s* system, then the *From Party* is informed.

A *From Party* is informed by a *To Party* that a message has been delivered by the *To Party* returning an *Acknowledgement Message*. <DB>Added this sentence here since we need to describe an *acknowledgement message* early in the chapter. Currently it is defined after it is used.</DB>

1.1.1 Persistent Storage and System Failure

A MSH that supports Reliable Messaging MUST keep messages that are sent or received reliably in *persistent storage*. In this context *persistent storage* is a method of storing data that does not lose information after a system failure or interruption.

This specification recognizes that different degrees of resilience may be realized depending on the technology that is used to persist the data. However, as a minimum, persistent storage that has the resilience characteristics of a hard disk (or equivalent) SHOULD be used. It is strongly RECOMMENDED though that implementers of this specification use technology that is resilient to the failure of any single hardware or software component.

Even after a system interruption or failure, a MSH MUST ensure that messages in persistent storage are processed in the same way as if the system failure or interruption had not occurred. How this is done is an implementation decision.

In order to support the filtering of duplicate messages, a Receiving MSH SHOULD, save the ***MessageId*** in *persistent storage*. It is also RECOMMENDED that the following be kept in *Persistent Storage*:

- the complete message, at least until the information in the message has been passed to the application or other process that needs to process it
- the time the message was received, so that the information can be used to generate the response to a Message Status Request (see section **Error! Reference source not found.**)

1.1.2 Methods of Implementing Reliable Messaging

Support for Reliable Messaging can be implemented in one of the following two ways:

- using the ebXML Reliable Messaging protocol, or
- using ebXML Header and Message structures together with commercial software products that are designed to provide reliable delivery of messages using alternative protocols.*<DB>Change elsewhere</DB>*

Each of these are described ~~below~~later.

1.2 Reliable Messaging Parameters

This section describes the parameters required to control reliable messaging. This parameter information is contained in the following:

- the *ebXML Message Header*, or

- the *CPA* that governs the processing of a message.

The table below indicates where these parameters may be set.

Parameter	**CPA**	**Header**
deliverySemantics	Yes	Yes
syncReplyMode	Yes	Yes
timeToLive	Yes	Yes
reliableMessagingMethod	No	Yes
ackRequested	No	Yes
timeout	Yes	No
retries	Yes	No
retryInterval	Yes	No
persistDuration	Yes	No

In this table, the following interpretation of the columns should be used:

1) if the **CPA** column contains a **Yes** then it indicates that the value that is present in the CPA determines the processing semantics

2) if the **CPA** column contains a **No** then it indicates that the parameter value is never specified in the **CPA**

3) <DB>I think we have four alternative interpretations here I prefer option a)<DB>:

   a) if the **Header** column contains a **Yes** then it indicates that the parameter value MAY be specified in the *ebXML Header* document. If it is present, then it overrides the value in the CPA

   b) if the Header column contains a Yes and the value of the header element differs from the equivalent in the CPA use the value in the header and report an error with ***severity*** of ***Warning*** and an ***errorCode*** of ***Inconsistent***

   c) if the Header column contains a Yes and the value of the header element differs from the equivalent in the CPA use the value in the CPA and report an error with ***severity*** of ***Warning*** and an ***errorCode*** of ***Inconsistent***

   d) if the Header column contains a Yes then the value of the header element MUST be set to the same value as in the CPA. If it differs, then report an error with ***severity*** of ***Error*** and an ***errorCode*** of ***Inconsistent<DB>***

1.2.1 Delivery Semantics

The ***deliverySemantics*** parameter may be present as either an element within the ***ebXMLHeader*** element or as a parameter within the CPA. See section **Error! Reference source not found.** for more information.

1.2.2 Sync Reply Mode

The ***syncReplyMode*** parameter may be present as either an element within the ***ebXMLHeader*** element or as a parameter within the CPA. See section **Error! Reference source not found.** for more information.

1.2.3 Time To Live

The ***TimeToLive*** element may be present within the ***ebXMLHeader*** element see section **Error! Reference source not found.** for more information.

1.2.4 Reliable Messaging Method

The ***ReliableMessagingMethod*** parameter indicates the requested method for Reliable Messaging that will be used when sending a Message. Valid values are:

- ***ebXML*** in this case the ebXML Reliable Messaging Protocol as defined in section 1) is followed, or
- ***Transport***, in this case a commercial software product is used for reliable delivery of the message, see section1.4.

1.2.5 Ack Requested

The ***AckRequested*** parameter is used by the Sending MSH to request that the Receiving MSH that receives the *Message* returns an *acknowledgment message* with an ***Acknowledgment*** element with a ***type*** of ***Acknowledgment***..

Valid values for ***IntermediateAckRequested*** are:

- ***Unsigned*** - requests that an unsigned Acknowledgement is requested
- ***Signed*** - requests that a signed Acknowledgement is requested, or
- ***None*** - indicates that no Acknowledgement is requested.

The default value is ***None***.

1.2.6 Timeout Parameter

The ***timeout*** parameter is an integer value that specifies the minimum time in seconds <DB>Perhaps this should be an XML Schema TimeDuration?. </DB> that the Sending MSH MUST wait for an *Acknowledgment Message* before first resending a message to the Receiving MSH.

1.2.7 Retries Parameter

The ***retries*** Parameter is an integer value that specifies the maximum number of times a Sending MSH SHOULD attempt to redeliver an unacknowledged or undelivered *message* using the same Communications Protocol.

1.2.8 RetryInterval Parameter

The ***retryInterval*** parameter is an integer value specifying, in seconds, <DB>Perhaps this should be an XML Schema TimeDuration?. </DB> the minimum time the Sending MSH MUST wait between retries, if an *Acknowledgment Message* is not received.

1.2.9 PersistDuration

The ***persistDuration*** parameter s the minimum length of time, expressed as a [XMLSchema] timeDuration, that data from a *Message* that is sent reliably, is kept in *Persistent Storage* by a MSH that receives that *Message*.

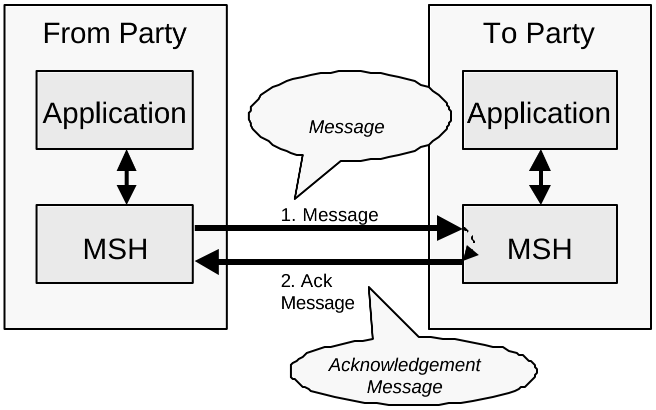
A MSH SHOULD NOT resend a message with the same ***MessageId*** to a receiving MSH if the elapsed time indicated by ***persistDuration*** has passed since the message was first sent as the receiving MSH will probably not treat it as a duplicate.

If a message cannot be sent successfully before ***persistDuration*** has passed, then the MSH should report a delivery failure (see section 1.5).

~~1.2~~1.3 ebXML Reliable Messaging Protocol

The ebXML Reliable Messaging Protocol described in this section MUST be followed if the ***deliverySemantics*** parameter/element is set to ***OnceAndOnlyOnce*** and the ***ReliableMessagingMethod*** parameter/element is set to ***ebXML*** (the default).

The ebXML Reliable Messaging Protocol is illustrated by the figure below.



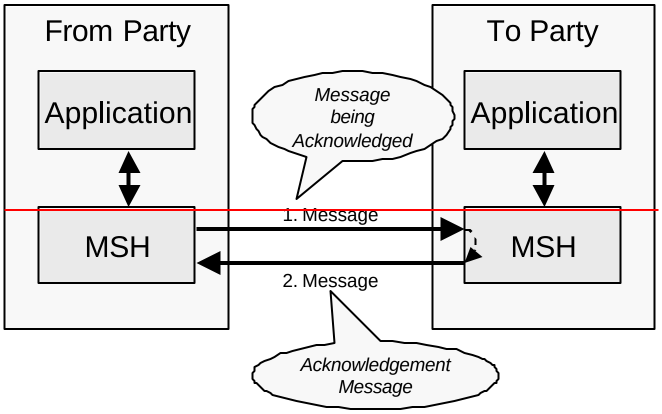


Figure 1~~10~~-1 Indicating that a message has been received

~~The diagram above illustrates two terms that are used in the remainder of this section:~~

~~? *message being acknowledged*. This is the Message that needs to be sent reliably and therefore needs to be acknowledged~~

~~? *acknowledgment message*. This is the message that acknowledges that the message being acknowledged has been received.~~

The receipt of the *acknowledgment message* indicates that the *message being acknowledged* has been ~~sent reliably~~successfully received and either processed or persisted by the receiving MSH to which the *message* was sent.

An *acknowledgment message* MUST contain a ***MessageData*** element with a ***RefToMessageId*** that contains the same value as the ***MessageId*** element in the *message being acknowledged*.

~~A Message can be sent reliably either over:~~

~~? a Single-hop i.e. the sending of a message directly from the *From Party's* MSH to the *To Party's* MSH without passing through any intermediate MSHs.~~

~~? Multi-hops i.e. the sending of a message indirectly from the *From Party's* MSH to the *To Party's* MSH via one or more intermediate MSHs.~~

~~Single-hop Reliable Messaging is described first followed by Multi-hop Reliable Messaging. Note that Multi-hop Reliable Messaging is an extension of Single-hop reliable Messaging.~~

~~1.2.1Single-hop Reliable Messaging~~

~~This section describes the REQUIRED behavior of a Message Service Handler (MSH) that is sending and/or receiving messages that support the ebXML Reliable Messaging Protocol.~~

~~1.2.1.1~~1.3.1 Sending Message Behavior

If a MSH is given data by an application that needs to be sent reliably then the MSH MUST do the following:

1) Create a message from components received from the application that includes:
    a) ***deliverySemantics*** set to ***OnceAndOnlyOnce***, and
    b) a ***RoutingHeader*** element that identifies the sender and the receiver URIs
~~1)~~2) Save the message in *persistent storage* (see section 1.1.1~~10.1.1~~)
~~2)~~3) Send the message ~~(the *message being acknowledged*)~~ to the *Receiver* MSH
~~3)~~4) Wait for the *Receiver* MSH to return an *acknowledgment message* and, if it does not, then resend the *identical* message as described in section 1.3.2.2~~10.2.1.3~~

~~It is RECOMMENDED that messages that are sent reliably include ***deliveryReceiptRequested*** set to ***Signed*** or ***UnSigned***.~~

~~If the message does not need to be sent reliably, then ***deliverySemantics*** MUST be set to ***BestEffort*** (the default).~~

~~1.2.1.2~~1.3.2 Receiving Message Behavior

If ***deliverySemantics*** on the received message is set to ***OnceAndOnlyOnce*** then do the following:

~~2)~~1) Check to see if the message is a duplicate (e.g. there is a message in *persistent storage* that was received earlier that contains the same value for the ***MessageId***)
~~3)~~2) If the message is not a duplicate then do the following:
    a) Save the ***MessageId*** of the received message in *persistent storage*. As an implementation decision, the whole message MAY be stored if there are other reasons for doing so~~.<DB>Need to re-look at how duplicates are detected if sequence numbers are used. </DB>~~
    b) If the received message contains a ***RefToMessageId*** element then do the following:
        i) Look for a message in *persistent storage* that has a ***MessageId*** that is the same as the value of ***RefToMessageId*** on the received Message
        ii) If a message is found in *persistent storage* then mark the persisted message as delivered

c) Generate an *Acknowledgement Message* in response (see section 1.3.2.1). <DB>This is a simpler version of the text in version 0.93 and relies more on interpretation of other parts of the spec.</DB>

~~c)If ***deliveryReceiptRequested*** is set to ***Signed*** or ***UnSigned*** then create an ***Acknowledgment*** element with ***type*** set to ***DeliveryReceipt*** that identifies the *received message*~~

~~d)If ***syncReplyMode*** is set to ***True*** then pass the data in the received message to the application or other process that needs to process it and wait for the application to produce a response.~~

~~e)If ***deliveryReceiptRequested*** is set to ***Signed*** or ***UnSigned***, or ***syncReplyMode*** is set to ***True*** then do the following:~~

~~i)Create a ***RoutingHeader*** element that identifies the sender and the receiver URIs~~

~~ii)Set the ***RefToMessageId*** to the value of the ***MessageId*** in the received message~~

~~iii)Create a *message* from the response generated by the application (if any), the ***Acknowledgment*** element (if any) and the ***RoutingHeader*** that includes ***deliverySemantics*** set to ***OnceAndOnlyOnce***~~

~~iv)Save the message in *persistent storage* for later resending~~

~~v)Send the message back to the Sending MSH~~

~~f)If ***syncReplyMode*** is set to ***False*** then pass the data in the received message to the application or other process that needs to process it. Note that, depending on the application, this can result in the application generating another message to be sent (see previous section).~~

~~4)~~3)If the message is a duplicate, then do the following:

a) Look in persistent storage for a response to the received message (i.e. it contains a ***RefToMessageId*** that matches the ***MessageId*** of the received message) that was *most recently sent* to the MSH that sent the received message (i.e. it has a ***RoutingHeader*** element with the greatest value of the ***Timestamp***.)<DB>Note it is not yet agreed whether the most recent message should be sent. Whatever message is sent, we need to define rules for it.</DB>

b) If no message was found in *persistent storage* then ignore the received message as either no message was generated in response to the message, or the processing of the earlier message is not yet complete

c) If a message was found in *persistent storage* then resend the persisted message back to the MSH that sent the received message.

1.3.2.1 Generating an Acknowledgement Message

An *Acknowledgement Message* MUST be generated whenever a message is received with:

- ***deliverySemantics*** set to ***OnceAndOnlyOnce*** and
- *reliableMessagingMethod* set to ***ebXML*** (the default).

As a minimum, it MUST contain a ***MessageData*** element with a ***RefToMessageId*** that contains the same value as the ***MessageId*** element in the *message being acknowledged*.

If ***ackRequested*** in the ***RoutingHeader*** of the received message is set to ***Signed*** or ***Unsigned*** then the acknowledgement message MUST also contain an ***Acknowledgement*** element.

Depending on the value of the ***syncReplyMode*** parameter, the *Acknowledgement Message* can also be sent at the same time as the response to the processing of the received message. In this case, the values for the *Header* elements of the *Acknowledgement Message* are set by the designer of the Service (see section **Error! Reference source not found.**).

If an ***Acknowledgment*** element is being sent on its own, then the value of the ***Header*** elements MUST be set as follows:

1) The ***Service*** element MUST be set to: http://www.ebxml.org/namespaces/messageService/MessageAcknowledgment
2) The ***Action*** element MUST be set to ***Acknowledgment***.
3) The ***From*** element MUST be set to the ***ReceiverURI*** from the last ***RoutingHeader*** in the *message* that has just been received
4) The ***To*** element MUST be set to the ***SenderURI*** from the last ***RoutingHeader*** in the *message* that has just been received
5) The ***RefToMessageId*** element MUST be set to the ***MessageId*** of the *message* that has just been received
6) The ***deliverySemantics*** MUST be set to BestEffort

~~1.2.1.3~~1.3.2.2 *Resending Lost Messages and Duplicate Filtering*

This section describes the behavior that is required by the sender and receiver of a message in order to handle when messages are lost. A message is "lost" when a sending MSH does not receive a response to a message. For example, it is possible that a *message* ~~*being acknowledged*~~ was lost, for example:

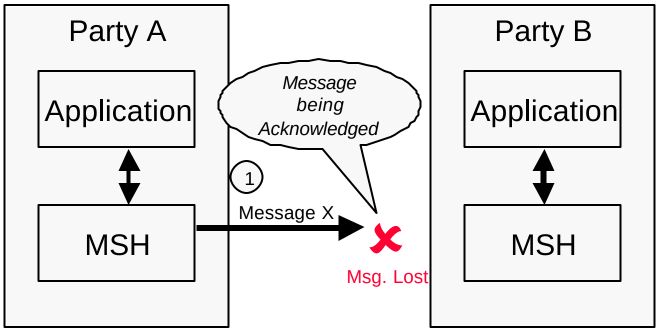


Figure 1~~10~~-2 Lost "Message Being Acknowledged"

It is also possible that the *Acknowledgment Message* was lost, for example ...

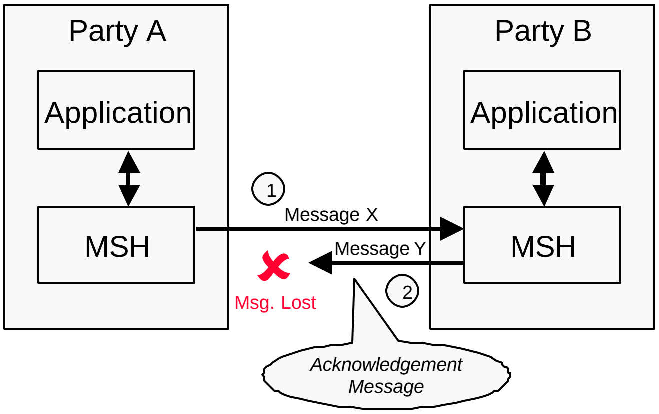


Figure 1~~10~~-3 Lost Acknowledgment Message

The rules that apply are as follows:

~~5)~~1) The Sending MSH MUST resend the original message if an *Acknowledgment Message* has not been received from the Receiving MSH and either of the following are true:

a) The message has not yet been resent and at least the time specified in the ***timeout*** parameter has passed since the first message was sent, or

b) The message has been resent, and the following are both true:

  i) At least the time specified in the ***retryInterval*** has passed since the last time the message was resent, and

  ii) The message has been resent less than the number of times specified in the ***retries*** Parameter

~~4)~~2)If the Sending MSH does not receive an *Acknowledgment Message* after the maximum number of retries, the Sending MSH SHOULD notify the application and/or system administrator function.

~~5)~~3)If the Sending MSH detects a communications protocol error that is unrecoverable at the transport protocol level then the Sending MSH SHOULD first attempt to resend the message using the same transport protocol until the number of ***retries*** has been reached, and then again, using ~~a~~ different communications protocols, if the CPA allows this. If these are not successful, then notify the From Party of the failure to deliver as described in section 1.5~~10.5~~.

1.3.2.3 Duplicate Message Handling

In this context:

- an *identical message* is a *message* that contains, apart from perhaps an additional ***RoutingHeader*** element, the same *ebXML Header* and *ebXML Payload* as the earlier *message* that was sent.
- a *duplicate message* is a *message* that contains the same ***MessageId*** as an earlier message that was received.
- the *most recent message* is the message with the latest ***Timestamp*** in the ***MessageData*** element that has the same ***RefToMessageId*** as the duplicate message that has just been received.*<DB>Chris Ferris, disagrees with resending the latest message. DB & CF need to go through this. This is carried over from the last version of the spec. </DB>*

Note that the Communication Protocol Envelope MAY be different. This means that the same message MAY be sent using different communication protocols and the reliable messaging behavior described in this section will still apply. The ability to use alternative communication protocols is specified in the CPA and is an OPTIONAL implementation specific feature.

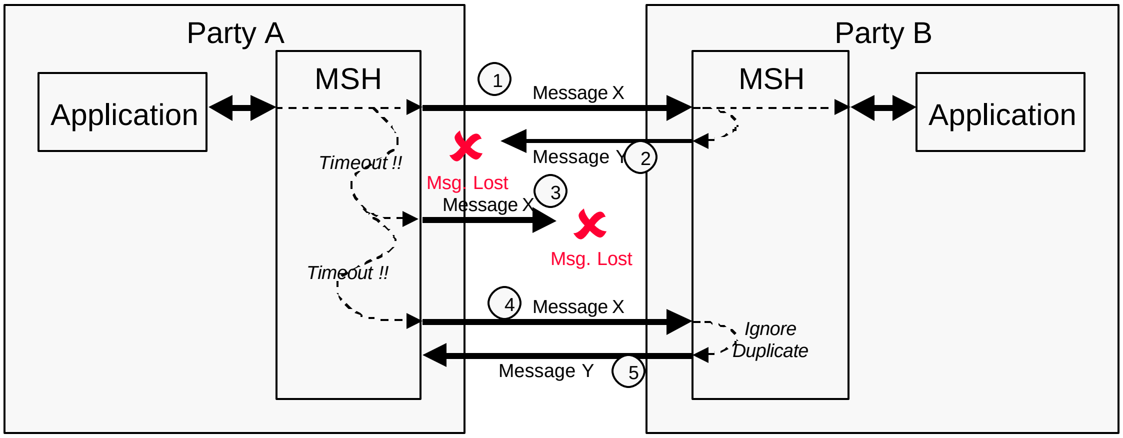


Figure 1~~10~~-4 Resending Lost Messages

The diagram above shows the behavior that MUST be by the sending and receiving MSH that are sent with ***deliverySemantics*** of ***OnceAndOnlyOnce***. ~~followed by the sender of the *message being acknowledged* (e.g. Message X) and the *acknowledgment message* (e.g. Message Y).~~ Specifically:

~~6)~~1)The sender of the *message* ~~*being acknowledged*~~ (e.g. Party A) MUST re-send the *identical message* ~~to the *To Party* MSH (e.g. Party B)~~ if no *Acknowledgment Message* is received

~~7)~~2)The recipient of the *message being acknowledged* (e.g. Party B), when it receives a *duplicate message*, MUST re-send to the sender of the *message* ~~*being acknowledged*~~ (e.g. Party A), a *message* identical to the *most recent message* that was sent to the recipient (i.e. Party A)

~~8)~~3)The recipient of the *message* ~~*being acknowledged*~~ (e.g. Party ~~A~~B) MUST ~~ignore *duplicate messages* and not~~NOT forward them a second time to the application, ~~the next MSH *<DB>next MSH is multi-hop, should not be here. </DB>*~~or other process that ultimately needs to ~~receive~~ process received messages~~them~~.

~~*<DB>The above also includes recipient behavior which is not part of sending behavior. Should be in a separate section. </DB>*~~

~~In this context:~~

~~? an *identical message* is a *message* that contains, apart from perhaps an additional ***RoutingHeader*** element, the same *ebXML Header* and *ebXML Payload* as the earlier *message* that was sent.~~

~~? a *duplicate message* is a *message* that contains the same ***MessageId*** as an earlier message that was received.~~

~~? the *most recent message* is the message with the latest ***Timestamp*** in the ***MessageData*** element that has the same ***RefToMessageId*** as the duplicate message that has just been received.*<DB>Chris Ferris, disagrees with resending the latest message. DB & CF need to go through this. </DB>*~~

~~Note that the Communication Protocol Envelope MAY be different. This means that the same message MAY be sent using different communication protocols and the reliable messaging behavior described in this section will still apply. The ability to use alternative communication protocols is specified in the CPA.~~

~~1.2.2~~1.3.3 Multi-hop Reliable Messaging

<DB>I've just concluded that we can probably do away with the complete Munlti-hop reliable messaging section if we consider the intermediary receiving MSH as acting as a proxy for the To Party MSH. This works since:

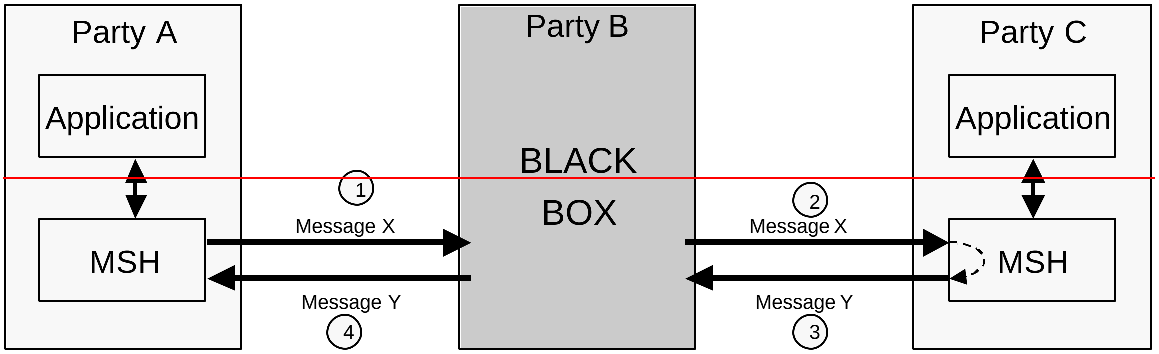
- The Acknowledgement message contains a ***From*** element that identifies the organization that generated the Acknowledgement element if it is not the To Party.
- The Routing Header can provide an audit trail (or not) if you allow multiple entries. After all, if some of the hops are not ebXML, then you cannot generate an audit trail for them

The big advantage is that it makes the behavior of the From Party the same whether or not multi-hop is being used. The text below illustrates how this could work.</DB>

Multi-hop reliable Messaging involves the sending of a message reliably from the *From Party* to the *To Party* via an intermediary that acts as a "black box". This means that the sender of a message does not need to know the address or protocols used to deliver the message to the final destination.

Multi-hop Reliable Messaging can occur either with or ~~:~~

~~?~~ without Intermediate ~~Intermediate~~ Acknowledgments~~, or~~



~~? with Intermediate Acknowledgments~~.

An Intermediary knows that Multi-hop Reliable Messaging with Intermediate Acknowledgments applies if the received message contains ***ackRequested*** set to ***Signed*** or ***UnSigned***.

1.3.3.1 *Multi-hop Reliable Messaging without Intermediate Acknowledgments*

This is illustrated by the diagram below.

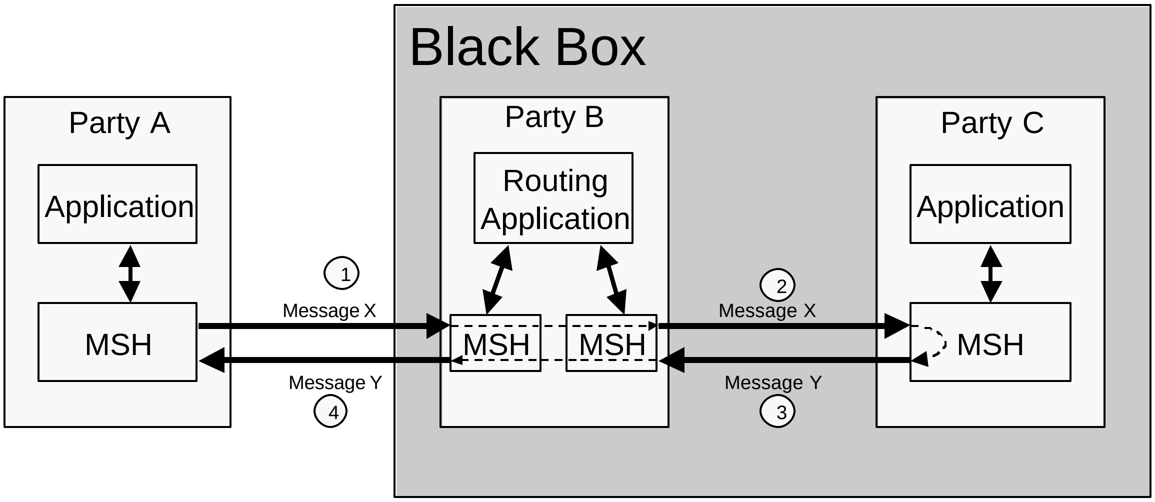
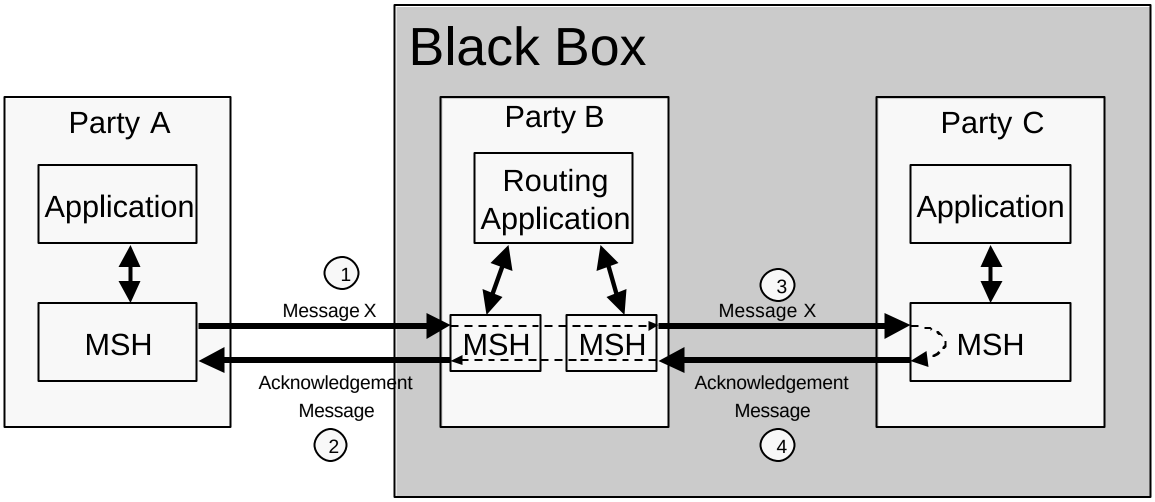


Figure 1-5 Multi-hop Reliable Messaging without Intermediate Acknowledgments

In this case, the intermediary (Party B) is acting as a proxy for the To Party (Party C).

1.3.3.2 Multi-hop Reliable Messaging with Intermediate Acknowledgments

This is illustrated by the diagram below.



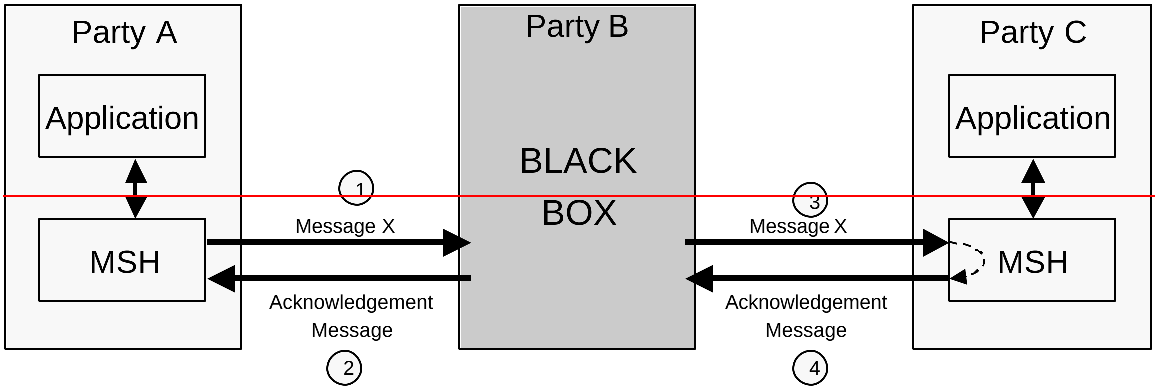


Figure 1-6 Multi-hop Reliable Messaging with Intermediate Acknowledgments

In this case, the Intermediary (Party B) accepts responsibility for delivering the message to its final destination by sending an *Acknowledgement Message* back to the sender of the original message. As far as sending and receiving of messages, the Intermediary behaves the same as a *To Party* with respect to the sending and receiving of messages.

If the Intermediary cannot, for some reason, deliver the message successfully to *To Party* (Party C), then it sends a Deli very Failure message to the *From Party* (Party A) – see section 1.5.

~~One reason for using Multi-hop Reliable Messaging with Intermediate Acknowledgments is when the *From Party* that is sending a message is confident that the total time taken for ...~~

~~? the *message being acknowledged* to be sent to the *To Party*, and~~

~~? the *acknowledgment message* to be returned~~

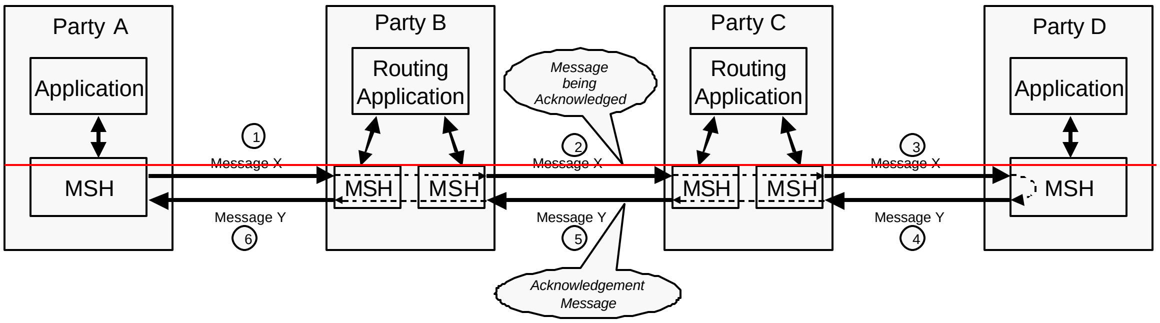
~~... is likely to result in the *From Party* resending the *message being acknowledged. <DB>Chris thinks this is superfluous, David thinks it useful as it explains why you should do multi-hop and helps an implementer decide when to use it. This requires further discussion. </DB>*~~

~~Each of these is described below.~~

~~*1.2.2.1Multi-hop Reliable Messaging without Intermediate Acknowledgments*~~

~~Multi-hop Reliable Messaging without Intermediate Acknowledgment is identified by the ***IntermediateAckRequested*** of the *Routing Header* for the hop being set to ***False*** (the default).~~

~~The overall message flow is illustrated by the diagram below.~~



~~**Figure 10-5 Multi-hop Reliable Messaging without Intermediate Acknowledgments**~~

~~This is essentially the same as Single-hop Reliable Messaging except that the Message passes through multiple intermediate parties. This means that:~~

~~? the *From Party* (e.g. Party A) and the *To Party* (e.g. Party D) are the only parties that adopt the Reliable Messaging behavior described in this section~~

~~? the intermediate parties (e.g. Parties B and C), just forward the messages they receive, they do not undertake any Reliable Messaging behavior.~~

~~This is described in more detail below:~~

~~6)The *From Party* and the *To Party* adopt the sending message and receiving message behavior described in sections 10.2.1.1 and 10.2.1.2 except that the *From Party* MSH (e.g. Party A) sends to an Intermediate Party (e.g. Party B) a message (the *message being acknowledged)* e.g. Message X in transmission 1, that contains~~

~~a)a ***QualityOfServiceInfo*** element with ***deliverySemantics*** set to ***OnceAndOnlyOnce***~~

~~b)a ***RoutingHeader*** element that contains the ***SenderURI*** of the sender (e.g. the URL for Party A's MSH) and the ***ReceiverURI*** of the next recipient of the message (e.g. the URL of Party B's MSH)~~

~~9)Once the Intermediate Party (e.g. Party B or Party C) receives the message, they determine its next destination (in the example above this could be done by the Routing Application) and forward the message (e.g. Transmission 2 of Message X) to the next Party (e.g. either Party C or Party D). Before sending the message they do the following:~~

~~a)transfer elements in the ebXML Header and Payload unchanged from the inbound message to the outbound message except that, they~~

~~b)add a ***RoutingHeader*** element to the ***RoutingHeaderList*** that contains the ***SenderURI*** of the next party to receive the message (e.g. the URL for Party C's or Party D's MSH) and the ***ReceiverURI*** (e.g. the URL for Party B's or Party C's MSH)~~

~~10)If the Sending MSH (either at the From Party or at an Intermediate Party) does not receive an *Acknowledgment Message* after the maximum number of retries, the Sending MSH SHOULD notify the following of the delivery failure:~~

~~The application and/or system administrator function if the Sending MSH is the *From Party* MSH, or~~

~~The Sending MSH of the *From Party*, if the Sending MSH is operated by an Intermediate Party (see section 10.5)~~

~~11)The previous step then repeats until eventually the message (e.g. Message X) reaches its final destination at the *To Party* (e.g. Party D)~~

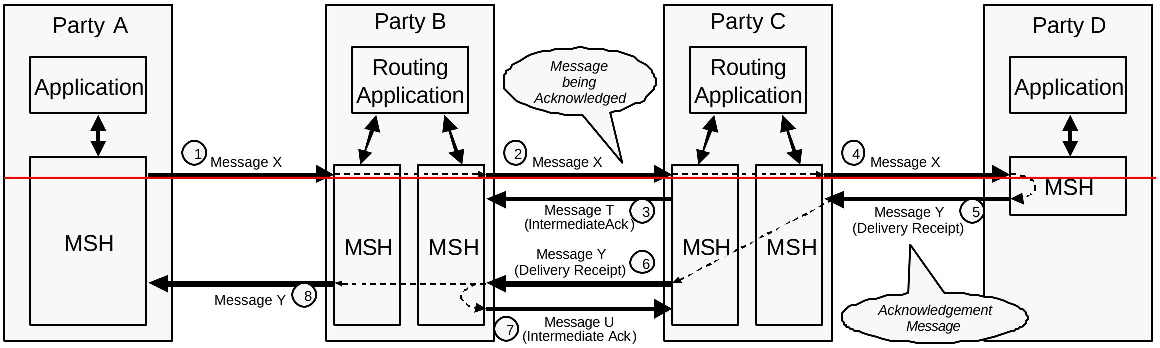
~~12)Once the *To Party* receives the message (i.e. the *message being acknowledged)* they return an *acknowledgment message* to the *From Party* through the Intermediate Parties.)~~

~~13)Steps 2 and 3 above then repeat until the *acknowledgment message* reaches the *To Party* (e.g. Party A)~~

~~*1.2.2.2Multi-hop Reliable Messaging with Intermediate Acknowledgments*~~

~~Multi-hop Reliable Messaging with Intermediate Acknowledgments is similar to Multi-hop Reliable Messaging without Intermediate Acknowledgment except that any of the Parties that are transmitting a Message can request that the recipient return an *Intermediate Acknowledgment.*~~

~~This is illustrated by the diagram below.~~



~~**Figure 10-6 Multi-hop Reliable Messaging with Intermediate Acknowledgments**~~

~~The main difference between Multi-Hop Reliable Messaging with Intermediate Acknowledgments and the without is:~~

~~? any party may request an intermediate acknowledgment~~

~~? any party that either sends or receives a message that requests an intermediate acknowledgment must adopt the reliable messaging behavior even if the ***QualityOfServiceInfo*** element indicates otherwise.~~

~~The rules that apply to Multi-hop Reliable Messaging with Intermediate Acknowledgment are as follows:~~

~~1)Any Party that is sending a message can request that the recipient send an *Acknowledgment Message* that is an *Intermediate Acknowledgment* by setting the ***IntermediateAckRequested*** of the ***RoutingHeader*** for the hop to ***Signed*** or ***Unsigned***. (e.g. Transmission 2 of Message X, or Transmission 6 of Message Y)~~

~~2)If a MSH that is not the *To Party* receives a message that requires an Intermediate Acknowledgment (e.g. Transmission 2 of Message X, or Transmission 6 of Message Y) then:~~

~~a)If the MSH can identify itself as the ***ReceiverURI*** in the ***RoutingHeader*** for the hop, and an *Intermediate Acknowledgment* is requested, then the MSH must return an *Acknowledgment Message* (e.g. Transmission 3 of Message T, or Transmission 7 of Message U) with:~~

~~i)The ***Service*** and ***Action*** elements set as in defined in section 10.4~~

~~ii)The ***From*** element contains the ***ReceiverURI*** from the last ***RoutingHeader*** in the message that has just been received~~

~~iii)The ***To*** element contains the ***SenderURI*** from the last ***RoutingHeader*** in the message that has just been received~~

~~iv)a ***RefToMessageId*** element that contains the ***MessageId*** of the message being acknowledged~~

~~v)a ***QualityOfServiceInfo*** element with ***deliverySemantics*** set to ***OnceAndOnlyOnce***~~

~~vi)an ***Acknowledgment*** element with type set to ***IntermediateAck***~~

~~vii)a ***RoutingHeader*** element that contains the ***SenderURI*** of the sender (e.g. the URL for Party C's or Party B's MSH) and the ***ReceiverURI*** of the next recipient of the message (e.g. the URL of Party B's or Party C's MSH)~~

~~3)If a MSH that is the *To Party* receives a message and it requires an Intermediate Acknowledgment (see step 2) then, unless the *To Party* is returning an *Acknowledgment Message* that is a *Delivery Receipt*, return an *Acknowledgment Message* as described in step 2c above.~~

~~1.3~~1.4 ebXML Reliable Messaging using Commercial Software Products~~Queuing Transports~~

This section describes the differences that apply if commercial software products ~~a Queuing Transport is~~ are used to implement Reliable Messaging.

Use of the ebXML Reliable Messaging Protocol is identified by the ***ReliableMessagingMethod*** parameter being set to ~~***Transport***~~ ***Tra*** for transmission~~ (either a Single-hop or a Multi-hop)~~

If Reliable Messaging using a commercial software product ~~Queuing Transport~~ is being used then the following rules apply:

~~1)An Intermediate Ack SHOULD not be requested. If an Intermediate Ack is requested, then it is ignored.~~

~~2)No message acknowledgments with an ***Acknowledgment*** element with a ***type*** of ***IntermediateAck*** should be sent, even if requested~~

~~3)~~1)Implementations should use the facilities of the commercial software product ~~Queuing Transport~~ to determine if the message was delivered

~~4)~~2)If the software product being used reports that a message cannot be delivered then~~If an intermediate MSH cannot forward a message to the next Party then the~~ the From Party should be notified using the procedure described in section 1.5~~10.5~~.

~~5)An acknowledgment message with an ***Acknowledgment*** element with a type attribute set to ***deliveryReceipt*** can be sent if requested to inform the sender of the message being acknowledged that the message was delivered.~~

~~1.4Service and Action Element Values~~

~~An ***Acknowledgment*** element can be included in an ***ebXMLHeader*** that is part of a *message* that is being sent as a result of processing of an earlier message. In this case the values for the ***Service*** and ***Action*** elements are set by the designer of the Service (see section 8.4.4).~~

~~An ***Acknowledgment*** element also can be included in an ***ebXMLHeader*** that does not include any results from the processing of an earlier message. In this case, the values of the ***Service*** and ***Action*** elements MUST be set as follows:~~

~~? The ***Service*** element MUST be set to:~~

~~***http://www.ebxml.org/namespaces/messageService/MessageAcknowledgment***~~

~~? The ***Action*** element MUST be set to **the value of the *type* attribute in the *Acknowledgment* element.**~~

~~Note that ***deliveryReceiptRequested*** must be set to ***None*** on a message that is only an acknowledgment.~~

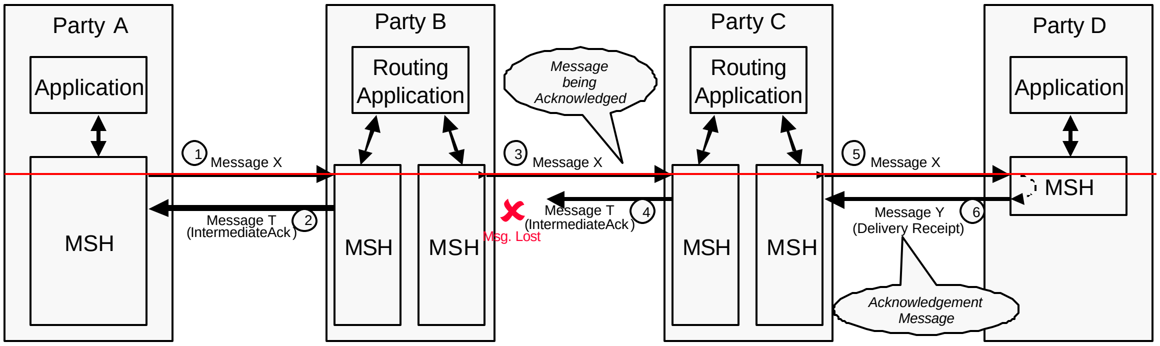
1.5 Failed Message Delivery

In the event that a MSH or other process that is involved, in some capacity in the delivery of a *message* that is sent with ***deliverySemantics*** set to *OnceAndOnlyOnce* has determined that the *message* cannot be delivered to the application or other process that has been designated to process the message, then that MSH or process SHOULD send a delivery failure notification *message* to the *From Party* that sent the *message*. The delivery failure notification message contains:

~~It is possible, that a *Message* cannot be delivered to its ultimate destination. This can be either:~~

~~? when the *To Party* MSH cannot deliver the message to the Application or other process that needs it, or~~

~~? when using Intermediate Acknowledgments and an Intermediate system determines that a message may have been lost. This is illustrated by the diagram below.~~



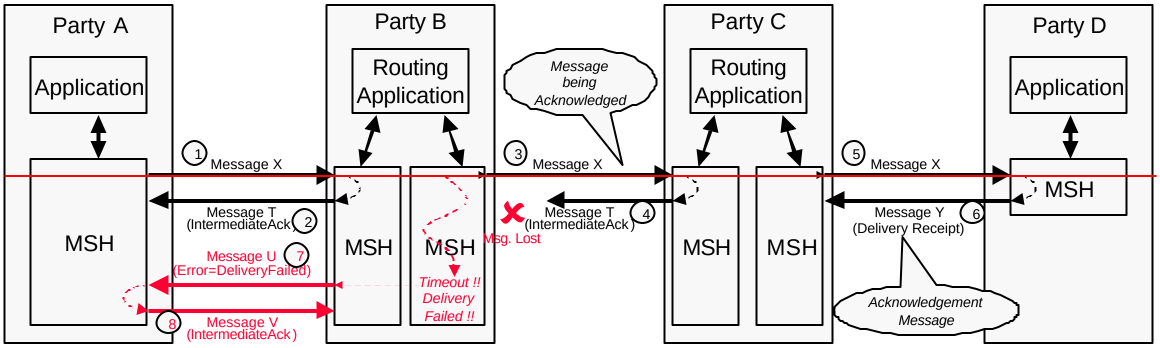
~~**Figure 10-7 Failed Message Delivery using Intermediate Acknowledgments**~~

~~In this example, Party B does not know if Party C (or Party D) has received the message since, even after resending, it has not received the *acknowledgment message* (Message T).~~

~~In both these circumstances the MSH that detects the problem MUST send a message to the *From Party* that sent the *message being acknowledged* (via the Intermediate Party if required). The message contains:~~

- a ***From Party*** that identifies the Party that detected the problem
- a ***To Party*** that identifies the ***From Party*** that created the message that could not be delivered
- a ***Service*** element and ***Action*** element set as described in **Error! Reference source not found.**~~11.5~~
- a ***QualityOfServiceInfo*** element with ***deliverySemantics*** set to the same value as the ***deliverySemantics*** on the message that could not be delivered
- an ***Error*** element with a severity of:
  - ***Error*** if the Party that detected the problem could not even transmit the message (e.g. ~~Transmission 3 was impossible~~the communications transport was not available)
  - ***Warning*** if the message ~~(e.g. Message X in Transmission 3)~~ was transmitted, but no *acknowledgment* *message* was received. This means that the message probably was not delivered although there is a small probability that it was
- an ***ErrorCode*** of ***DeliveryFailure***

~~This is illustrated by the diagram below by the text and arrows in red.~~


Party A
Application
MSH
Party B
Routing Application
MSH
MSH
Party C
Routing Application
MSH
MSH
Party D
Application
MSH
Message being Acknowledged
① Message X
② Message T (IntermediateAck)
③ Message X
④ Message T (IntermediateAck)
Msg. Lost
⑤ Message X
⑥ Message Y (Delivery Receipt)
Acknowledgement Message
⑦ Message U (Error=DeliveryFailed)
⑧ Message V (IntermediateAck)
Timeout !! Delivery Failed !!

~~**Figure 10-8 Reporting Failed Message Delivery**~~

~~Note that the message that contains an ***Error*** element with an ***ErrorCode*** of ***DeliveryFailure*** (e.g. Message U in Transmission 7) might be sent reliably. It is possible the *acknowledgment message* for this message (e.g. Message V in Transmission 8) is not received. In this case, the Party that detects the failed delivery (e.g. Party B) SHOULD inform the Party (e.g. Party A) that sent the *message being acknowledged* (e.g. Message X in Transmission 1) of the failure. How this is done is outside the scope of this specification.~~

~~1.6Reliable Messaging Parameters~~

~~This section describes the parameters required to control reliable messaging. This parameter information may be contained:~~

~~? in the ebXML Message header, or~~

~~? in the CPA associated with the message.~~

~~If the information is in both the ebXML message header and the CPA, the information in the header over-rides the CPA.~~

~~1.6.1Who sets Message Service Parameters~~

~~The values to be used in parameters can be specified by the following parties:~~

~~? the *From Party*~~
~~? the *To Party*~~
~~? the sending Message Service Handler (MSH)~~
~~? the receiving Message Service Handler~~

~~Parameters set by the *From Party* or the *To Party*, apply to the delivery of a message as a whole. Parameters set by the sending or receiving MSH apply to a single-hop.~~

~~Note that the *From Party* is the sending MSH and the *To Party* is the receiving MSH for the first/last MSH that handles the message.~~

~~The table below indicates where these parameters may be set.~~

~~In this table, the following interpretation of the columns should be used:~~

~~7)the **Specified By** columns indicates the Party that sets the value in the Collaboration Party Protocol, Message Header, or Routing Header~~

~~14)if the **CPA/CPP** column contains a **Yes** then it indicates that the party in the **Specified By** column specifies the value that is present in the CPP~~

~~15)if the **CPA/CPP** column contains a **No** then it indicates that the parameter value is never specified in the **CPP**~~

~~16)if the **Message Header** or **Routing Header** columns contain a **Yes** then it indicates that the parameter value may be specified in the **Header** element or **Routing Header** and over-rides any value in the CPA. It the value is not specified in the **Header element** or **Routing Header** then the value in the **CPA** must be used.~~

~~17)if the **Message Header/Routing Header** columns contain a **No** then it indicates that the value in the **CPA** is always used~~

~~18)if the **Message Header/Routing Header** columns contain a **N/A** then it indicates that the value may be specified in another header~~

~~These parameters are described below.~~

~~1.6.2From Party Parameters~~

~~This section describes the parameters that are set by the *From Party*~~

~~*1.6.2.1Delivery Semantics*~~

~~The ***deliverySemantics*** parameter may be present as either an element within the ***ebXMLHeader*** element or as a parameter within the CPA. See section 8.4.7.1 for more information.~~

~~*1.6.2.2Delivery Receipt Requested*~~

~~The ***deliveryReceiptRequested*** parameter may be present as either an element within the ***ebXMLHeader*** element or as a parameter within the CPA. See section 8.4.7.2 for more information.~~

~~*1.6.2.3Sync Reply Mode*~~

~~The ***syncReplyMode*** parameter may be present as either an element within the ***ebXMLHeader*** element or as a parameter within the CPA. See section 8.4.7.3 for more information.~~

~~*1.6.2.4Time To Live*~~

~~The ***TimeToLive*** element may be presented within the ***ebXMLHeader*** element see section 8.4.6.4 for more information.~~

~~1.6.3To Party Parameters~~

~~This section describes the parameters that are set by the *To Party*~~

~~*1.6.3.1Delivery Receipt Provided*~~

~~The ***DeliveryReceiptProvided*** parameter indicates whether a *To Party* can provide an *acknowledgment message* with a ***type*** attribute of ***deliveryReceipt*** in response to a message. Valid values are:~~

~~? ***Signed*** - indicates that only a signed Delivery Receipt can be provided~~
~~? ***Unsigned*** - indicates only an unsigned Delivery Receipt can be provided,~~
~~? ***Both*** - indicates that either a signed or an unsigned Delivery Receipt can be provided, or~~
~~? ***None*** - indicates that the *To Party* does not create Delivery Receipts~~

~~If a MSH receives a Message where ***deliveryReceiptRequested*** is in not compatible with the value of ***DeliveryReceiptProvided*** then the MSH MUST return an *Error Message* to the *From Party* MSH, reporting that the ***DeliveryReceiptProvided*** is not supported. This must contain an ***errorCode*** set to ***NotSupported*** and a ***severity*** of Error.~~

~~1.6.4Sending MSH Parameters~~

~~This section describes the parameters that are set by the *Party* that operates the Sending MSH.~~

~~*1.6.4.1Reliable Messaging Method*~~

~~The ***ReliableMessagingMethod*** parameter indicates the requested method for Reliable Messaging that will be used when sending a Message. Valid values are:~~

~~? ***ebXML*** in this case the ebXML Reliable Messaging Protocol as defined in section 10.2 is followed, or~~

~~? ***Transport***, in this case a Queuing Transport Protocol is used for reliable delivery of the message, see section10.3.~~

~~*1.6.4.2Intermediate Ack Requested*~~

~~The ***IntermediateAckRequested*** parameter is used by the Sending MSH to request that the Receiving MSH that receives the *Message* returns an *acknowledgment message* with an ***Acknowledgment*** element with a ***type*** of ***IntemediateAcknowledgment***..~~

~~Valid values for ***IntermediateAckRequested*** are:~~

~~? ***Unsigned*** - requests that an unsigned Delivery Receipt is requested~~

~~? ***Signed*** - requests that a signed Delivery Receipt is requested, or~~

~~? ***None*** - indicates that no Delivery Receipt is requested.~~

~~The default value is ***None***.~~

~~*1.6.4.3Timeout Parameter*~~

~~The ***timeout*** parameter is an integer value that specifies the time in seconds that the Sending MSH MUST wait for an *Acknowledgment Message* before first resending a message to the Receiving MSH.~~

~~*1.6.4.4Retries Parameter*~~

~~The ***retries*** Parameter is an integer value that specifies the maximum number of times the *message being acknowledged* must be resent to the Receiving MSH using the same Communications Protocol by the Sending MSH.~~

~~*1.6.4.5RetryInterval Parameter*~~

~~The ***retryInterval*** parameter is an integer value specifying, in seconds, the time the Sending MSH MUST wait between retries, if an *Acknowledgment Message* is not received.~~

~~*1.6.4.6Deciding when to resend a message*~~

~~The Sending MSH MUST resend the original message if an *Acknowledgment Message* has not been received from the Receiving MSH and either:~~

~~? the message has not yet been resent and at least the time specified in the ***timeout*** parameter has passed since the first message was sent, or~~

~~? the message has been resent, and~~

  ~~-at least the time specified in the ***retryInterval*** has passed since the last time the message was resent, and~~

  ~~-the message has been resent less than the number of times specified in the ***retries*** Parameter, and~~

~~If the Sending MSH does not receive an *Acknowledgment Message* after the maximum number of retries, the Sending MSH SHOULD notify either:~~

~~? the application and/or system administrator function if the Sending MSH is the *From Party* MSH, or~~

~~? send an message reporting the delivery failure, if the Sending MSH is operating by an Intermediate Party (see section 10.5)~~

~~1.6.5Receiving MSH Parameters~~

~~This section describes the parameters that are set by the *Party* that operates the Receiving MSH.~~

~~*1.6.5.1Reliable Messaging Methods Supported*~~

~~The ***reliableMessagingMethodsSupported*** parameter is a list of the methods that a MSH uses to support Reliable Messaging. It must be a URI. The URI for the ebXML Reliable Messaging Protocol described in section 10.2 is ***http://www.ebxml.org/namespaces/reliableMessaging***~~

~~*1.6.5.2PersistDuration*~~

~~***persistDuration*** is the minimum length of time, expressed as a [XMLSchema] timeDuration, that data from a *Message* that is sent reliably, is kept in *Persistent Storage* by a MSH that receives that *Message*.~~

~~In order to support the filtering of duplicate messages, a Receiving MSH MUST, as a minimum, save the ***MessageId*** in *persistent storage*. It is also RECOMMENDED that the following be kept in *Persistent Storage*:~~

~~? the complete message, at least until the information in the message has been passed to the application or other process that needs to process it~~

~~? the time the message was received, so that the information can be used to generate the response to a Message Status Request (see section 9.1.1)~~

~~***persistDuration*** is specified in the CPA.~~

~~A MSH SHOULD NOT resend a message with the same ***MessageId*** to a receiving MSH if the elapsed time indicated by ***persistDuration*** has passed since the message was first sent as the receiving MSH will probably not treat it as a duplicate.~~

~~If a message cannot be sent successfully before ***persistDuration*** has passed, then the MSH should report a delivery failure (see section 10.5).~~

~~Note that implementations may determine that a message is persisted for longer than the time specified in ***persistDuration***, for example in order to meet legal requirements or the needs of a business process. This information is recorded separately within the CPA.~~

~~In order to ensure that persistence is continuous as the message is passed from the receiving MSH to the process or application that is to handle it, it is RECOMMENDED that a message is not removed from *persistent storage* until the MSH knows that the data in the message has been received by the process/application.~~

~~*1.6.5.3MSH Time Accuracy*~~

2 ~~The *mshTimeAccuracy* parameter in the CPA indicates the minimum accuracy that a Receiving MSH keeps the clocks it uses when checking, for example, *TimeToLive*. It's value is in the format "mm:ss" which indicates the accuracy in minutes and seconds.~~Parameters that need to be specified in the CPA

<DB>The following (or something similar) is not part of the TRP spec but needs to be included in the CPA spec.</DB>

2.1.1.1 Delivery Receipt Requested

The ***deliveryReceiptRequested*** parameter may be present as either an element within the ***ebXMLHeader*** element or as a parameter within the CPA. See section **Error! Reference source not found.** for more information.

2.1.1.2 Delivery Receipt Provided

The ***DeliveryReceiptProvided*** parameter indicates whether a *To Party* can provide an *acknowledgment message* with a ***type*** attribute of ***deliveryReceipt*** in response to a message. Valid values are:

- ***Signed*** - indicates that only a signed Delivery Receipt can be provided
- ***Unsigned*** **-** indicates only an unsigned Delivery Receipt can be provided,
- ***Both*** - indicates that either a signed or an unsigned Delivery Receipt can be provided, or
- ***None*** **-** indicates that the *To Party* does not create Delivery Receipts

If a MSH receives a Message where ***deliveryReceiptRequested*** is in not compatible with the value of ***DeliveryReceiptProvided*** then the MSH MUST return an *Error Message* to the *From Party* MSH, reporting that the ***DeliveryReceiptProvided*** is not supported. This must contain an ***errorCode*** set to ***NotSupported*** and a ***severity*** of Error.

2.1.1.3 Reliable Messaging Methods Supported

The ***reliableMessagingMethodsSupported*** parameter is a list of the methods that a MSH uses to support Reliable Messaging. It must be a URI. The URI for the ebXML Reliable Messaging Protocol described in section 1) is ***http://www.ebxml.org/namespaces/reliableMessaging***

2.1.1.4 PersistDuration

persistDuration is the minimum length of time, expressed as a [XMLSchema] timeDuration, that data from a *Message* that is sent reliably, is kept in *Persistent Storage* by a MSH that receives that *Message*.

In order to support the filtering of duplicate messages, a Receiving MSH MUST, as a minimum, save the ***MessageId*** in *persistent storage*. It is also RECOMMENDED that the following be kept in *Persistent Storage*:

- the complete message, at least until the information in the message has been passed to the application or other process that needs to process it
- the time the message was received, so that the information can be used to generate the response to a Message Status Request (see section **Error! Reference source not found.**)

persistDuration is specified in the CPA.

A MSH SHOULD NOT resend a message with the same ***MessageId*** to a receiving MSH if the elapsed time indicated by ***persistDuration*** has passed since the message was first sent as the receiving MSH will probably not treat it as a duplicate.

If a message cannot be sent successfully before ***persistDuration*** has passed, then the MSH should report a delivery failure (see section 1.5).

Note that implementations may determine that a message is persisted for longer than the time specified in ***persistDuration***, for example in order to meet legal requirements or the needs of a business process. This information is recorded separately within the CPA.

In order to ensure that persistence is continuous as the message is passed from the receiving MSH to the process or application that is to handle it, it is RECOMMENDED that a message is not removed from *persistent storage* until the MSH knows that the data in the message has been received by the process/application.

2.1.1.5 MSH Time Accuracy

The ***mshTimeAccuracy*** parameter in the CPA indicates the minimum accuracy that a Receiving MSH keeps the clocks it uses when checking, for example, ***TimeToLive***. It's value is in the format "mm:ss" which indicates the accuracy in minutes and seconds.

3 Acknowledgement element

Changes required to the acknowledgement element

~~8.9~~3.1 Acknowledgment Element

The Acknowledgment element is an optional element that is used by one Message Service Handler to indicate that another Message Service Handler has received a message.

For clarity two terms are defined:

- *message being acknowledged*. This is the Message that is has been received by a MSH that is now being acknowledged
- *acknowledgment message*. This is the message that acknowledges that the *message being acknowledged* has been received.

The *message being acknowledged* is identified by the ***RefToMessageId*** contained in the ***MessageData*** element contained within the ***Header*** Element of the acknowledgment message containing the value of the ***MessageId*** of the message being acknowledged.

The ***Acknowledgment*** element consists of the following:

- a ***Timestamp*** element
- a ***From*** element

~~? a ***type*** attribute~~

- a ***signed*** attribute

~~8.9.1~~3.1.1 Timestamp element

No change

~~8.9.2~~3.1.2 From element

This is the same element as the ***From*** element within ***Header*** element (see section **Error! Reference source not found.**~~8.4.1~~). However, when used in the context of an Acknowledgment Element, it contains the identifier of the *Party* that is generating the *acknowledgment message*.

If the ***From*** element is omitted then the *Party* that is sending the element is identified by the ***From*** element in the ***Header*** element.

~~8.9.3~~3.1.3 type attribute

delete this section

~~8.9.4~~3.1.4 signed attribute

No change

4 Updated XML Schema

This specifies the only required change to the Schema …

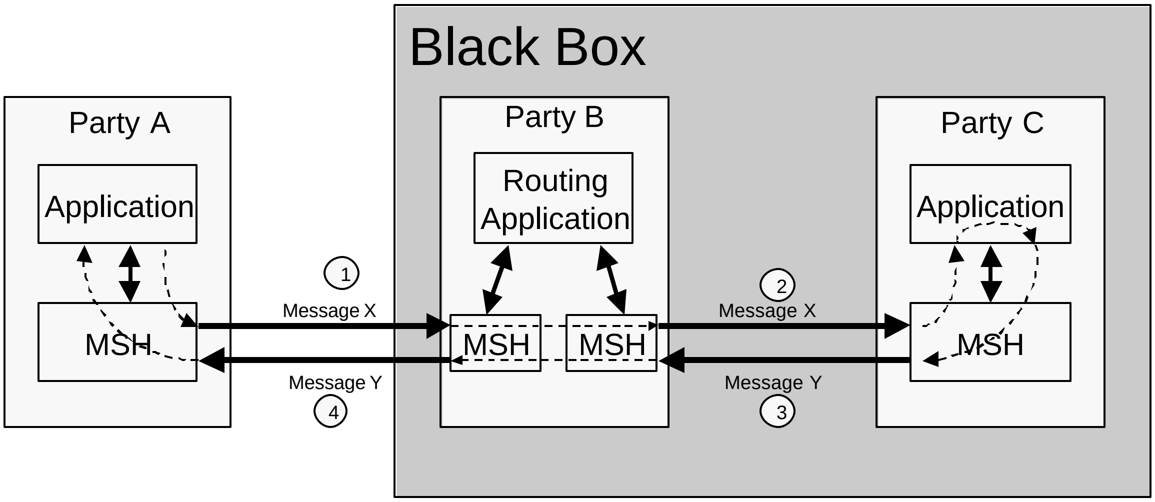
```
<!-- ACKNOWLEDGEMENT -->
   <xsd:element name="Acknowledgment">
          <xsd:complexType>
                 <xsd:sequence>
                        <xsd:element ref="Timestamp"/>
                        <xsd:element ref="From" minOccurs="0" maxOccurs="1"/>
                 </xsd:sequence>
                 <xsd:attribute name="id" type="xsd:ID"/>
                 <xsd:attribute name="type" use="default" value="DeliveryReceipt"/>
                 <xsd:simpleType>
                        <xsd:restriction base="xsd:NMTOKEN">
                               <xsd:enumeration value="DeliveryReceipt"/>
                               <xsd:enumeration value="IntermediateAck"/>
                        </xsd:restriction>
                 </xsd:simpleType>
                 <xsd:attribute name="signed" type="xsd:boolean"/>
          </xsd:complexType>
   </xsd:element>
```

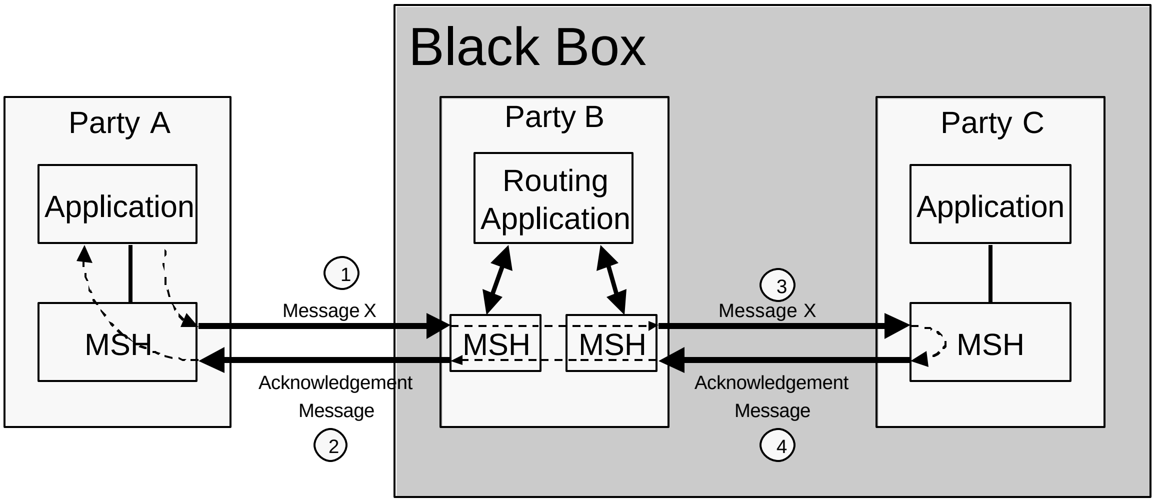
… to …

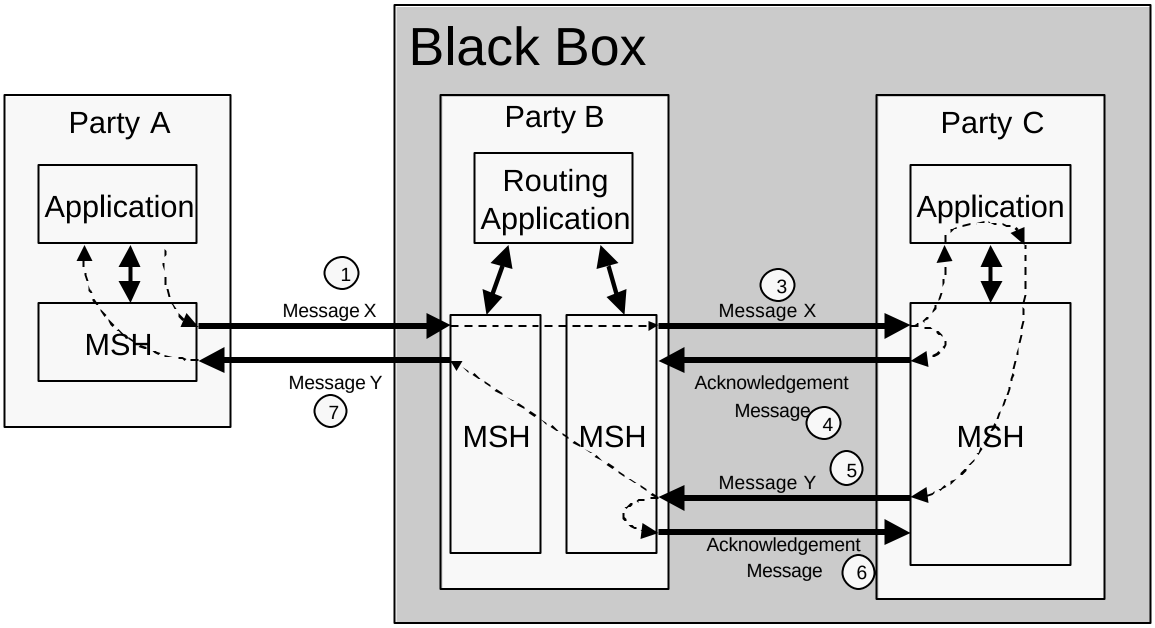
```
<!-- ACKNOWLEDGEMENT -->
   <xsd:element name="Acknowledgment">
          <xsd:complexType>
                 <xsd:sequence>
                        <xsd:element ref="Timestamp"/>
                        <xsd:element ref="From" minOccurs="0" maxOccurs="1"/>
                 </xsd:sequence>
                 <xsd:attribute name="id" type="xsd:ID"/>
                 <xsd:attribute name="signed" type="xsd:boolean"/>
          </xsd:complexType>
   </xsd:element>
```

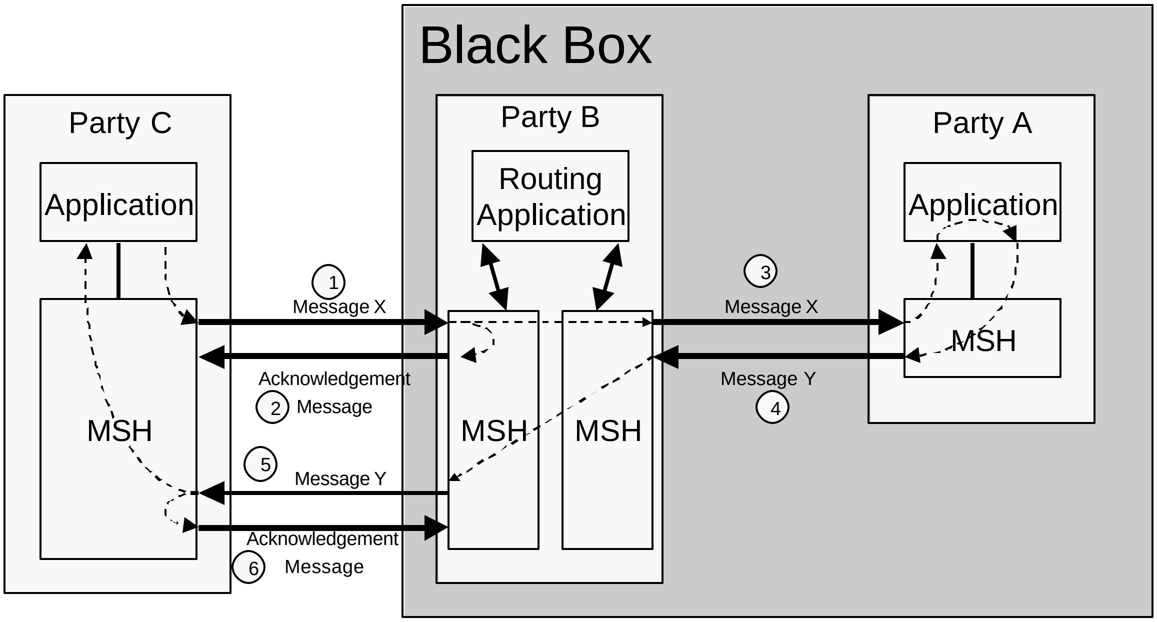
5 Non-normative examples of multi-hop

This section is not to be included in the spec but shows a number of alternative message flows that illustrate how the black box approach and multi-hop could work.


Black Box
Party A
Application
MSH
Party B
Routing Application
MSH
MSH
Party C
Application
MSH
①
Message X
Message Y
④
②
Message X
Message Y
③


Black Box
Party A
Application
MSH
Party B
Routing Application
MSH
MSH
Party C
Application
MSH
①
Message X
Acknowledgement Message
②
③
Message X
Acknowledgement Message
④


Black Box
Party A
Application
MSH
Party B
Routing Application
MSH
MSH
Party C
Application
MSH
① Message X
Message Y ⑦
③ Message X
Acknowledgement Message ④
Message Y ⑤
Acknowledgement Message ⑥


Black Box
Party C
Application
MSH
Party B
Routing Application
MSH
MSH
Party A
Application
MSH
① Message X
Acknowledgement ② Message
⑤ Message Y
Acknowledgement ⑥ Message
③ Message X
Message Y ④