The Addison-Wesley Signature Series

Enterprise Integration Patterns

Designing, Building, and Deploying Messaging Solutions

Gregor Hohpe
Bobby Woolf

With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon

Forewords by John Crupi and Martin Fowler

List of Patterns

Example: *Web Services Request-Response*

Web services standards, as of SOAP 1.1 [SOAP 1.1], do not provide very good support for asynchronous messaging, but SOAP 1.2 starts to plan for it. SOAP 1.2 incorporates the *Request-Response Message Exchange* pattern [SOAP 1.2 Part 2], a basic part of asynchronous SOAP messaging. However, the request-response pattern does not mandate support for "multiple ongoing requests," so it does not define a standard *Correlation Identifier* field, not even an optional one.

As a practical matter, service requestors often do require multiple outstanding requests. "Web Services Architecture Usage Scenarios" [WSAUS] discusses several different asynchronous Web services scenarios. Four of them—Request-Response, Remote Procedure Call (where the transport protocol does not support [synchronous] request-response directly), Multiple Asynchronous Responses, and Asynchronous Messaging—use `message-id` and `response-to` fields in the SOAP header to correlate a response to its request. This is the request-response example:

SOAP Request Message Containing a Message Identifier

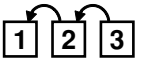
```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:MessageId>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
      ........
  </env:Body>
</env:Envelope>
```

SOAP Response Message Containing Correlation to Original Request

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-567b-2891-b623-9dke28yod7m9</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
      ........
  </env:Body>
</env:Envelope>
```

Like the JMS and .NET examples, in this SOAP example, the request message contains a unique message identifier, and the response message contains a response (e.g., a correlation ID) field whose value is the message identifier of the request message.

Message Sequence



My application needs to send a huge amount of data to another process, more than may fit in a single message. Or, my application has made a request whose reply contains too much data for a single message.

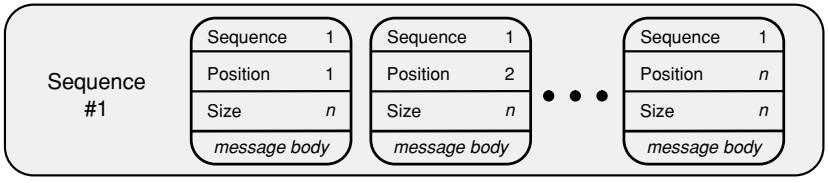
How can messaging transmit an arbitrarily large amount of data?

It's nice to think that messages can be arbitrarily large, but there are practical limits to how much data a single message can hold. Some messaging implementations place an absolute limit on how big a message can be. Other implementations allow messages to get quite big, but large messages nevertheless hurt performance. Even if the messaging implementation allows large messages, the message producer or consumer may place a limit on the amount of data it can process at once. For example, many COBOL-based and mainframe-based systems will consume or produce data only in 32 Kb chunks.

So, how do you get around this? One approach is to limit your application so it never needs to transfer more data than the messaging layer can store in a single message. This is an arbitrary limit, though, which can prevent your application from producing the desired functionality. If the large amount of data is the result of a request, the caller could issue multiple requests, one for each result chunk, but that increases network traffic and assumes the caller even knows how many result chunks will be needed. The receiver could listen for data chunks until there are no more (but how does it know there aren't any more?) and then try to figure out how to reassemble the chunks into the original, large piece of data, but that would be error-prone.

Inspiration comes from the way a mail order company sometimes ships an order in multiple boxes. If there are three boxes, the shipper marks them as "1 of 3," "2 of 3," and "3 of 3," so the receiver knows which ones he has received and whether he has received all of them. The trick is to apply the same technique to messaging.

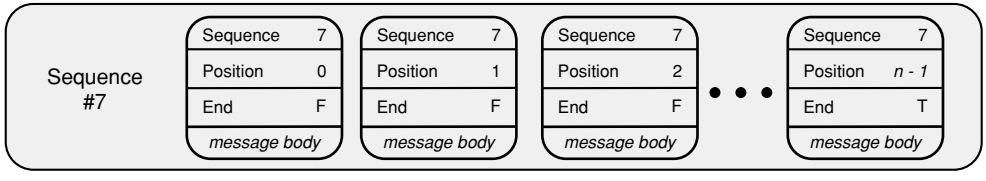
Whenever a large set of data needs to be broken into message-size chunks, send the data as a *Message Sequence* and mark each message with sequence identification fields.



The three *Message Sequence* identification fields are as follows.

1. **Sequence identifier**—Distinguishes this cluster of messages from others.
2. **Position identifier**—Uniquely identifies and sequentially orders each message in a sequence.
3. **Size** or **End indicator**—Specifies the number of messages in the cluster or marks the last message in the cluster (whose position identifier then specifies the size of the cluster).

The sequences are typically designed so that each message in a sequence indicates the total size of the sequence—that is, the number of messages in that sequence. As an alternative, you can design the sequences so that each message indicates whether it is the final message in that sequence.



Message Sequence with End Indicator

Let's say a set of data needs to be sent as a cluster of three messages. The sequence identifier of the three-message cluster will be some unique ID. The position identifier for each message will be different: either 1, 2, or 3 (assuming

that numbering starts from 1, not 0). If the sender knows the total number of messages from the start, the sequence size for each message is 3. If the sender does not know the total number of messages until it runs out of data to send (e.g., the sender is streaming the data), each message except the last will have a "sequence end" flag that is false. When the sender is ready to send the final message in the sequence, it will set that message's sequence end flag as true. Either way, the position identifiers and sequence size/end indicator will give the receiver enough information to reassemble the parts back into the whole, even if the parts are not received in sequential order.

If the receiver expects a *Message Sequence*, then every message sent to it should be sent as part of a sequence, even if it is only a sequence of one. Otherwise, when a single-part message is sent without the sequence identification fields, the receiver may become confused by the missing fields and may conclude that the message is invalid (see *Invalid Message Channel* [115]).

If a receiver gets some of the messages in a sequence but doesn't get all of them, it should reroute the ones it did receive to the *Invalid Message Channel* (60).

An application may wish to use a *Transactional Client* (484) for sending and receiving sequences. The sender can send all of the messages in a sequence using a single transaction. This way, none of the messages will be delivered until all of them have been sent. Likewise, a receiver may wish to use a single transaction to receive the messages so that it does not truly consume any of the messages until it receives all of them. If any of the messages in the sequence are missing, the receiver can choose to roll back the transaction so that the messages can be consumed later. In many messaging system implementations, if a sequence of messages is sent in one transaction, the messages will be received in the order they are sent, which simplifies the receiver's job of putting the data back together.

When the *Message Sequence* is the reply message in a *Request-Reply* (154), the sequence identifier and the *Correlation Identifier* (163) are usually the same thing. They would be separate if the application sending the request expected multiple responses to the same request, and one or more of the responses could be in multiple parts. When only one response is expected, then uniquely identifying the response and its sequence is permissible but redundant.

Message Sequence tends not to be compatible with *Competing Consumers* (502) or *Message Dispatcher* (508). If different consumers/performers receive different messages in a sequence, none of the receivers will be able to reassemble the original data without exchanging message contents with each other. Thus, a message sequence should be transmitted via a *Message Channel* with a single consumer.

An alternative to *Message Sequence* is to use a *Claim Check* (346). Rather than transmitting a large document between two applications, if the applications both have access to a common database or file system, store the document and just transmit a key to the document in a single message.

Using *Message Sequence* is similar to using a *Splitter* (259) to break up a large message into a sequence of messages and using an *Aggregator* (268) to reassemble the message sequence back into a single message. *Splitter* (259) and *Aggregator* (268) enable the original and final messages to be very large, whereas *Message Sequence* enables the *Message Endpoint*s (95) to split the data before any messages are sent and to aggregate the data after the messages are received.

Example: *Large Document Transfer*

Imagine that a sender needs to send a receiver an extremely large document, so large that it will not fit within a single message or is impractical to send all at once. In this case, the document should be broken into parts, and each part can be sent as a message. Each message needs to indicate its position in the sequence and how many messages there are in all. For example, the maximum size of an MSMQ message is 4 MB. [Dickman] discusses how to send a multipart message sequence in MSMQ.

Example: *Multi-Item Query*

Consider a query that requests a list of all books by a certain author. Because this could be a very large list, the messaging design might choose to return each match as a separate message. Then, each message needs to indicate the query this reply is for, the message's position in the sequence, and how many messages to expect.

Example: *Distributed Query*

Consider a query that is performed in parts by multiple receivers. If the parts have some order to them, this will need to be indicated in the reply messages so that the complete reply can be assembled properly. Each receiver will need to know its position in the overall order and will need to indicate that position is the reply's message sequence.
