

Chapter 4

Message Brokers

The messaging systems described in the previous chapter can be seen as an integration infrastructure based on message queues that applications can use to communicate with each other asynchronously. The use of such infrastructure has several advantages. One is that all applications use the same interface (the messaging system API) to communicate with each other, rather than having to integrate each application with the custom API of other applications. The second advantage is that the use of message queues decouples applications from each other by allowing each application to handle requests at its own pace, without blocking other applications. And the third advantage is that applications can rely on the mechanisms of the messaging system that provide guaranteed delivery, efficient routing, storage, etc. without having to implement those mechanisms themselves.

However, from the point of view of integration, the use of a messaging system alone does not suffice, since it leaves up to the applications the decision of how they will interact with each other. If application *A* is configured to send messages to application *B*, and *B* is configured to send messages to application *C*, then if the need arises to change the sequence of interaction between these applications, it will be necessary to reconfigure the behavior of each application. In such scenario, the integration logic is distributed across applications, and changing this logic requires not only reconfiguring several applications but also checking that the new configuration works as expected. In a large infrastructure comprising many heterogeneous applications, such effort may be cumbersome and also error-prone.

Message brokers are different from messaging systems in that they themselves control how applications will interact with each other. Rather than letting each application decide the destination for a message, it is the message broker who will decide which application the message should be sent to. In this scenario, the sending application simply produces and delivers the message to the message broker which, in turn, will route the message according to its own rules. These rules can be as simple as specifying the subscribers for each kind of message, hence the connection between message brokers and the publish–subscribe paradigm. However, the simple fact that these rules are stored in a single place changes the nature of the integration solution and provides much more flexibility when the integration logic needs to

be reconfigured. In this scenario, it becomes much easier to modify the interaction between applications, as this can be done at the message broker, by rewiring them in order to implement a different business process.

Already in Sect. 3.1 we discussed the concept of content-based routing, where a messaging system is configured to route messages to different destinations according to the actual message content. This can be based not only on the type of message, but even for messages of the same type it may be the case that some have an attribute with a certain value and should be forwarded to one queue, while others have a different value for the same attribute and should be routed to another queue. A more sophisticated routing mechanism was presented in Fig. 3.5, where a process manager controls the sequence of interactions and the flow of data between applications. This provides much more flexibility in implementing the desired behavior and at the same time provides a single point of control. The downside is that, under heavy load, the process manager may become a performance bottleneck, since all traffic must pass through that routing component.

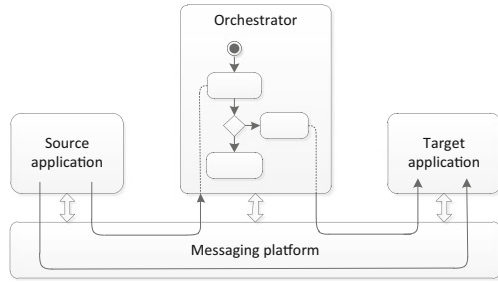
In the messaging technologies described in Sect. 3.5 (JMS) and Sect. 3.6 (MSMQ) we have not seen such mechanisms as content-based routers or process managers since they are usually reserved for message brokers, i.e., systems which not only provide the messaging infrastructure, as messaging systems do, but also provide capabilities for configuring the whole interaction between applications. In this context, both JMS and MSMQ can be regarded as belonging to the category of messaging systems, whereas a system such as the one presented in Chap. 2 (BizTalk) belongs to the category of message brokers. There are other integration platforms with capabilities similar to those of BizTalk Server, and all of them provide process-based routing on top of a messaging infrastructure.

4.1 Message-Level vs. Orchestration-Level Integration

A message broker is a composite system that comprises a messaging platform and an orchestrator, as illustrated in Fig. 4.1. Using a message broker, it is possible to integrate applications at two different levels: at the level of the messaging platform, or at the level of the orchestrator. At the level of the messaging platform, messages are routed between applications according to the publish–subscribe rules configured in the platform. If an application is configured as being a subscriber for a certain kind of message (or for a message that obeys certain criteria), then the message will be forwarded to that application. The publish–subscribe rules are stored in the messaging platform itself; applications connected to that platform have no control over that configuration and it is possible to change the interaction between applications just by changing the configuration in the messaging platform.

The same publish–subscribe mechanism can be used to integrate applications via the orchestrator. Here, the orchestrator is automatically configured as a subscriber for the messages that the orchestration is waiting to receive. Also, when the orchestration specifies that a certain message should be sent to an application,

Fig. 4.1 Message-level and orchestration-level routing in a message broker



that application is automatically configured as a subscriber for that message. So the orchestrator works together with the messaging platform by configuring the publish–subscribe rules that enable the orchestration to work as expected. In other words, the publish–subscribe rules are configured automatically in the messaging platform in order to support the behavior defined in the orchestration.

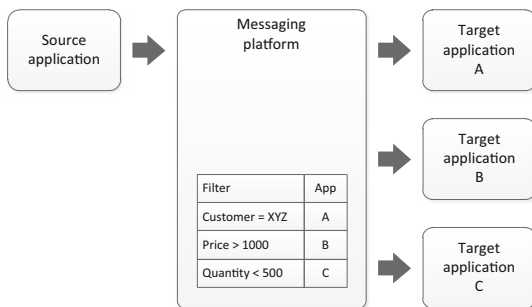
As a result of these mechanisms, applications can be integrated in two different ways. One way is to use the messaging platform alone and manually configure a set of publish–subscribe rules in the messaging platform. These rules are *static* and, if needed, they must be changed by a system administrator. Another way is to develop an orchestration, and the publish–subscribe rules will be configured automatically in the messaging platform when the orchestration is *deployed*. Deployment means installing and configuring the orchestration in the message broker, both in the orchestrator and in the messaging platform, so that everything is prepared for the orchestration to run. In this case, the publish–subscribe rules are *dynamic* and they will be reconfigured automatically if the orchestration is changed and redeployed.

4.2 Publish–Subscribe with Message Filters

Perhaps the simplest way to implement a publish–subscribe system is through the use of message filters. The concept of message filter has been briefly described in Sect. 3.1 as a special kind of router. Basically, a message filter is a component that can be associated with a receiving application; the application will receive a message only if the message complies with the condition specified in the filter. If each application connected to the messaging platform has its own filter, then message routing between applications becomes as simple as iterating through all filters and checking whether the message produced by a sending application complies with the conditions in any of those filters. The filters which yield a positive result will let the message go through to the corresponding receiving application.

Figure 4.2 illustrates this concept. A message that enters the messaging platform will be inspected in order to determine which filter conditions hold true. In the example of Fig. 4.2, a message will be forwarded to application *A* if the property *Customer* has the value *XYZ*; it will be forwarded to application *B* if *Price* is

Fig. 4.2 Use of filters to route messages in a message broker



over 1,000; and it will be forwarded to application *C* if Quantity is less than 500. These conditions are independent and, in general, they do not have to be mutually exclusive, so the same message may be forwarded to n -out-of- m applications, where $n \leq m$. In particular, it may happen that none of the conditions hold true (i.e., $n = 0$); in this case, the message is not delivered to any destination. Although such behavior may be the intended one for a given message, the message broker is likely to generate an error or warning if such situation occurs. For example, this is the reason why BizTalk generates the error “no subscribers found.”

The use of message filters requires the message broker to inspect an incoming message in order to retrieve the property values that are needed to evaluate the filter conditions. In the example of Fig. 4.2, Customer, Price, and Quantity are properties to be retrieved from the message body. Clearly, the message broker must know how to find these properties in the message. Usually, the message schema will be available to the broker (in case, for example, transformations are needed) so it would not be difficult to inspect the message content. However, for performance reasons, it is a good idea to facilitate the access to the required properties, so that the message broker spends as little time as possible in the processing of each message. This leads to the concept of *promoted properties*, as explained next.

4.3 Promoted Properties

When the routing of messages is to be decided based on the actual message content, as in the example of Fig. 4.2, it becomes necessary to provide the message broker with all the properties required for evaluating the filter conditions. Rather than requiring the message broker to go through the whole message content in order to find those properties, it is more convenient to bring those properties to the forefront of the message, where they can be accessed more easily. This can be done, for example, by writing those properties in the message header, so that the message broker can read them without having to open and go through the message body.

To understand this concept, an analogy can be established with a traditional mail system. Instead of sending a letter to the destination specified in the envelope, suppose that the destination is to be determined based on some information contained in the letter itself. In this scenario, the postman would have to open the letter and read its content in order to find that info. Rather than allowing this to happen, it is more convenient to write that information outside in the envelope, so that the letter can be delivered without being opened.

The same principle applies to the messaging platform in a message broker. For routing purposes, it is more convenient to make the required properties easily accessible, rather than requiring the message broker to read and retrieve them from the message body. Making properties (that are inside the message body) accessible from the outside is referred to as *property promotion*.

Property promotion is a mechanism that should be used with care. In general, it is possible to promote all properties in the message body. However, properties should not be promoted because they *can* be promoted, but because they *must* be promoted. The way in which property promotion is implemented depends on the particular message broker being used but, in its simplest form, promoting a property may imply writing additional message headers. These headers inform the message broker that the message carries properties that are required for content-based routing. The presence of such headers also instructs the message broker to read and use those property values. If the message contains promoted properties that are not being used for message routing, then the message broker undergoes unnecessary work and delay while processing the message.

Therefore, system developers and integrators should have a good understanding of promoted properties and what they are used for, in order to avoid making suboptimal decisions which may have an impact on the performance of the message broker and of the integration solution overall.

In BizTalk, the use of promoted properties is implemented by means of a special-purpose *property schema*, as illustrated in Fig. 4.3. A property schema defines which properties will be used for message routing. The conditions specified in a message filter must refer to properties defined in a property schema. So, in a sense, a property schema defines the vocabulary to be used by message filters. A different matter is how to obtain the values for these promoted properties; this is done by establishing a relationship between the elements in a message schema and the properties defined in a property schema.

For example, suppose that an incoming message (e.g., a purchase request) has an element Qty used to denote the quantity of the product to be purchased. Also, suppose that this information is used to route the purchase request in the following way: if the quantity exceeds 500 units the request is denied and is routed back to the original sender; otherwise, the request is approved and forwarded to an ERP system. (This is similar to the example that was used in Sect. 2.4 to present a simple orchestration; here we resort to the same scenario to illustrate the use of content-based routing at the message level.) In this scenario, it would be necessary to create a property schema and to define a promoted property, e.g., Quantity. Then, in a second step, it would be necessary to establish a relationship between the Qty element in

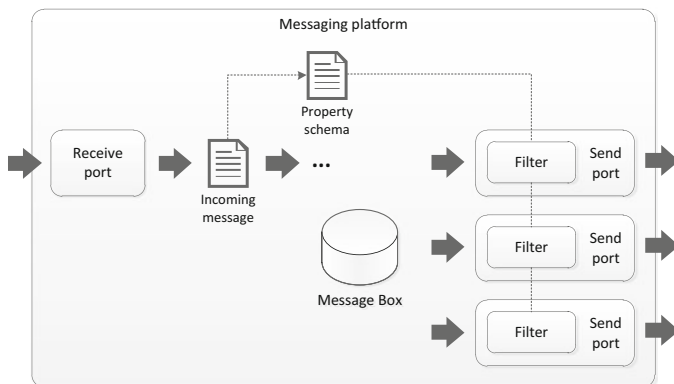


Fig. 4.3 Use of property schemas in BizTalk

the purchase request schema and the Quantity promoted property. This is equivalent to *promoting* the Qty element, and such promotion has two effects:

- The first is that the value for the Qty element will be brought to the forefront of the message, so that the message broker can read it without having to read and parse the message body.
- The second effect is that, as a new purchase request enters the messaging platform, the value of the Qty element will be used to set the value of the Quantity property defined in the property schema.

In general, the name of the promoted property (Quantity in this example) does not have to match the name of the element (Qty) which provides the value for that property. However, the names used to specify the filter conditions (e.g., `Quantity ≤ 500`) must be the names of properties defined in the property schema.

After reading the promoted properties in the message and setting the value for the corresponding properties in the property schema, these properties can now be used to evaluate the filter conditions. Figure 4.3 illustrates the use of property schemas in connection with filters at each send port. As explained in Sect. 2.3, a send port has an adapter, a pipeline, and an optional transformation map. In addition, a send port may have another optional component, which is the message filter. The filter contains a set of logical expressions, each yielding a result of true or false. These expressions are based on the values of properties defined in a property schema, and they can be combined with logical AND and OR operators. The filter condition may therefore comprise several expressions on different properties.

The concept is similar to that presented in Fig. 4.2, but while in Fig. 4.2 the filter condition for each target application had a single expression, in practice the conditions may be composed of several expressions connected by logical operators. Another difference between Figs. 4.2 and 4.3 is in the place where these filter conditions are actually stored. In Fig. 4.2 these appear to be stored together, somewhere inside the messaging platform, while Fig. 4.3 shows that, in BizTalk, the filter conditions are actually stored in the filter within each send port.

4.4 Orchestration-Level Integration

In enterprise systems integration, the behavior of an integration solution can often be implemented in different ways. In the previous section, a solution that checked the quantity of a purchase request and decided where to send that request was implemented at the messaging level based on message routing mechanisms. The same behavior can be implemented at the orchestration level, with much more flexibility. At the orchestration level the solution is not confined to the publish–subscribe paradigm and, in principle, it is possible to implement any kind of interaction between applications. In this context, the message routing mechanisms of the messaging platform are essential in order to ensure that the orchestration runs as expected. The key feature of an orchestrator is that it allows specifying the interaction between applications at a different level, on top of the messaging platform.

When using an orchestrator, the messaging platform is configured in such a way that the orchestrator becomes a subscriber for all messages that the deployed orchestrations are waiting to receive. On the other hand, the orchestrator also becomes the producer of messages that are to be delivered to target applications. In this context, the orchestrator publishes and subscribes messages just like a regular application. The difference is the close connections that exist between the orchestrator and the messaging platform, especially in terms of receive and send ports: the ports defined in an orchestration can be *bound* to specific ports that exist in the messaging platform, so that receiving or sending a message at the orchestration level has the same effect as routing messages at the messaging level.

Figure 4.4 illustrates the use of an orchestration to implement the behavior of the same scenario as before, where a purchase request is received and the decision of what to do with it depends on some property of that request, in this case the quantity specified in the Qty element. To implement this scenario, the orchestration has the following steps:

1. It receives the purchase request through a receive port.
2. It checks whether the Qty element in the purchase request is greater than 500, and then:
 - a. If so, it transforms the request into a denied request message and sends it through a send port back to the original requester (left branch).
 - b. If not, then it just forwards the message to the ERP system through another send port (right branch).
3. In either case, the orchestration has nothing left to do afterwards and terminates.

It is worth noting that the transformation on the left branch can be attained by a transformation map, so the use of a transformation map in the orchestration effectively dispenses the use of a transformation map in the send port. This mere detail is quite significant, as it shows that the use of an orchestration brings visibility to components of the integration solution that would otherwise be embedded in the

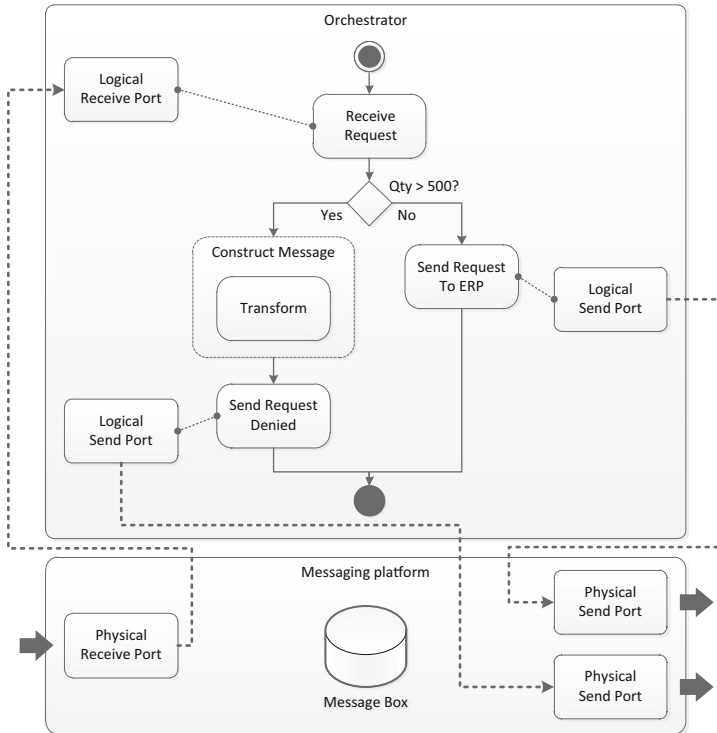


Fig. 4.4 Integration through an orchestrator on top of a messaging platform

messaging platform. Also, in the orchestration the send port can be changed while keeping the same transformation, whereas at the messaging platform changing to another send port would require configuring the transformation map in the new send port. These are just some examples of the advantages in using an orchestration rather than relying solely on the mechanisms of the messaging platform.

Figure 4.4 also illustrates the connections between ports in the orchestration and ports in the messaging platform. At the orchestration level, ports are referred to as *logical ports*, and they specify the points of entry and exit of messages in the orchestration. In the messaging platform, ports are referred to as *physical ports* and they contain all the required configurations (in terms of protocols and addresses) to communicate with the application associated with that port. After an orchestration has been deployed, but before it can actually run, it is necessary to establish a connection between each of its logical ports and a physical ports existing, or to be created, in the messaging platform. Such connection between a logical port and physical port is referred to as a *port binding*.

When a message enters a physical receive port, if that port is bound to a logical port, then the message is forwarded to the orchestration containing that logical port, as in Fig. 4.4. Likewise, when the orchestration sends a message through a logical

send port, the message is sent through the physical send port that is bound to that logical port. For physical ports that are not bound to logical ports, the messaging platform will make use of its own routing mechanisms, as described in the previous section. In particular, a physical send port that is not bound to a logical port will need to have a filter in order to subscribe to messages.

4.5 Distinguished Properties

In the example of Fig. 4.4, the orchestration receives a purchase request and decides whether to deny the request (left branch) or to forward it to the ERP system (right branch). The decision is based on the value of the Qty element in the incoming message, so the orchestration must have access to this element in order to determine how to proceed. Contrary to what happens in the messaging platform (Sect. 4.3), where the properties to be accessed for the purpose of message routing must have been promoted, at the orchestration level there is no need to promote such properties, since the orchestration has full access to the actual message content.

As explained in Sect. 3.1, and as shown in Sects. 3.5 and 3.6, a message has two main parts known as header and body. While the header is used by the messaging platform for routing purposes, the body carries the actual content that is of interest to applications. This is similar to a traditional mail scenario in which the information in the envelope is used by the post office to deliver a letter to its destination; but once the letter arrives at the destination, the receiver will throw away the envelope and focus on its actual content. The same happens with the messaging platform and the orchestrator: while the messaging platform will use the message header to route the message, the orchestrator will be working with the actual message body in order to implement the desired integration logic.

Therefore, the orchestrator can make use of any content available in a message to implement the orchestration logic. For example, if a message needs to be transformed to another schema, the orchestration can use a transformation map to create a new message with the same content but with a different structure. However, some elements in particular may play a critical role in determining how the orchestration will be executed, as is the case with the element Qty in the example of Fig. 4.4. To allow the use of expressions such as `Qty > 500` in the orchestration, those elements need to be marked in a special way, so that the orchestrator knows that Qty refers to a specific element in the purchase request message. An element that is marked for this purpose is called a *distinguished property*. Essentially, distinguished properties are message elements that can be used in expressions throughout an orchestration, to determine how the orchestration will be executed at run-time.

Distinguished properties, contrary to promoted properties, do not pose a problem from the performance point of view. The use of promoted properties requires the messaging platform to retrieve and store them for routing purposes, and this may result in a slight increase in the time required to handle each message at run-time. On

the other hand, distinguished properties have no significant impact on performance; they are used at design-time to specify how the behavior of an orchestration may change depending on the actual content of messages. Regardless of how this content affects the orchestration flow, it does not require any additional processing, since that content (the message body) is available to the orchestration in any case.

To summarize, promoted properties are used at the message level to implement message routing based on filters associated with send ports. Distinguished properties are used at the orchestration level to specify how the orchestration will behave at run-time according to the content of messages. In both cases, the message content is used to determine the behavior of the integration solution. However, at the message level one can only make use of publish–subscribe mechanisms, while at the orchestration level there is a wide range of constructs allowing to implement any form of desired behavior, as we will see in later chapters.

4.6 Correlations

In Sect. 3.1 we explained the concept of process-based routing which relies on a process manager to coordinate the interaction between applications, as illustrated in Fig. 3.5. The purpose of this process manager is exactly the same as that of an orchestrator in a message broker, and just as a process manager is able to execute multiple instances of the same process, so an orchestrator is able to execute multiple instances of an orchestration. As an example, in Fig. 4.4 the first step in the orchestration is to receive a purchase request from a receive port. This means that the orchestration is triggered every time a purchase request arrives at that receive port. Triggering the orchestration means creating and starting a new orchestration instance. Each orchestration instance has a life of its own and is independent from other instances. In particular, there will be instances where the purchase request will be denied because the quantity exceeds 500, and other instances where the request will be accepted because the quantity is lower. Also, in general, some instances may run successfully while others may end in error, and some instances may take long to complete while others may be rather quick. It all depends on the particular data and conditions that an orchestration instance finds at run-time.

For a better understanding of the concept of orchestration instance, an analogy can be established with an order placed in an online bookshop. As the customer enters a new order, a new instance of an order processing orchestration is triggered. This orchestration may consist in, for example, fetching the books from the warehouse, packing them, and shipping them to the customer. Each order originates an instance of this orchestration that is independent from other instances being created by the orders of the same or different customers. For example, a customer may check the status of an order at any time, and each order has its own status. Also, an order may be canceled without affecting the processing of other orders. Since orders are handled as separate process instances, they can be managed independently.

When a customer wants to check the status of an order, a small but important problem arises: the customer must be able to identify the order whose status is to be retrieved. Naturally, providing the title of the ordered book does not suffice, since many customers may have ordered the same book. Perhaps providing the title together with the customer info may suffice, if the customer did not order the same book more than once. In any case, a precise, unambiguous way of identifying the customer order is needed in order to locate the correct orchestration instance in the midst of all instances for all book orders. This is not just a problem of choosing an appropriate identifier as when selecting a primary key for a relational database table; the problem of identifying a particular orchestration instance may appear several times during the execution of that instance, and it has far-reaching implications with respect to the inner workings of a message broker.

Figure 4.5 illustrates the problem by means of a simple orchestration which, at a certain point, sends a request and receives a response from an external system. To begin with, three messages enter the messaging platform through the first receive port shown in the left side. These three messages trigger three instances of the same orchestration, all of which will do exactly the same thing: they will send a shipping request to a logistics provider application, they will receive the response from that application, and they will send a confirmation to the customer. At the point shown in Fig. 4.5, the three instances have already sent the shipping request to the logistics provider and are now waiting for a response. When the first response comes in, which orchestration instance should it be sent to?

Clearly, there needs to be a mechanism for *correlating* the present response with a previously sent request. Such correlation is necessary in order to determine which orchestration instance is the correct recipient for each response. This becomes a matter of determining the subscriber for an incoming message, and therefore must be dealt with at the message level, in much the same way as explained in Sect. 4.2. In particular, it will be necessary to use promoted properties (explained in Sect. 4.3) to correlate a response with a previous request based on some property or properties. In the context of a correlation, the set of promoted properties that are used to correlate messages is referred to as the *correlation id*.

In a relational database, it is often easier to create a new column to serve as the primary key for a table, rather than selecting a subset of existing columns, which is not guaranteed to yield a unique identifier. The same scenario applies to correlation ids: often it becomes easier to create a new element in the message schemas just for the purpose of serving as a correlation id, rather than using an identifier based on the existing message elements, which are not guaranteed to be unique. In the example of the online bookshop from above, when a customer wants to check the status of an order, a suitable identifier must be provided to identify the desired process instance. For this purpose, it is common to have a unique order id, which can be used to identify the process instance without having to resort to other order data.

In any case, regardless of the actual message elements that are selected to serve as a correlation id, these elements must be promoted so that the orchestration instance can be determined at the message level.

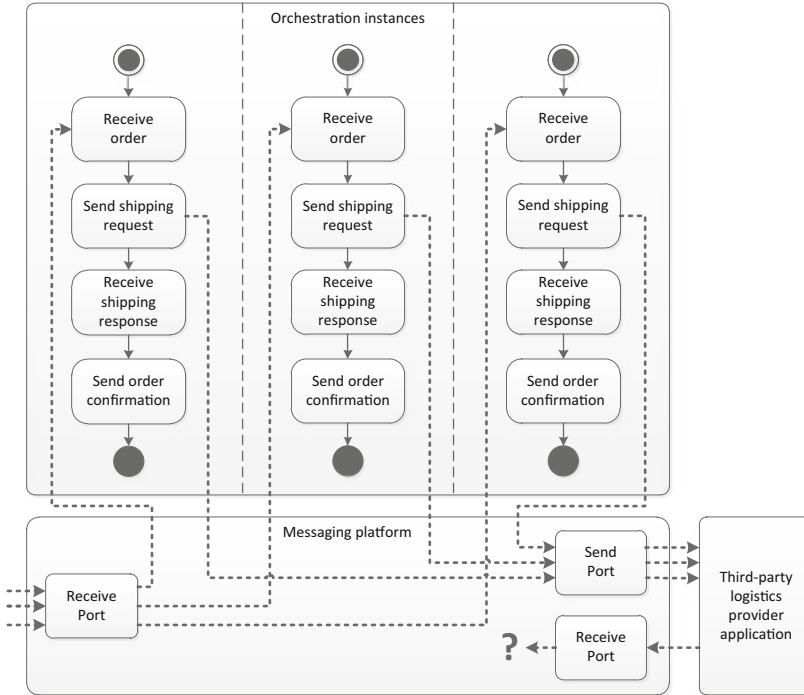


Fig. 4.5 Need for correlation in an orchestration with a request-response interaction

For the moment, let us assume that each orchestration instance shown in Fig. 4.5 has a unique order id. For simplicity, it can be assumed that this order id is provided in the original order message that triggered the whole orchestration. When an orchestration instance creates the shipping request, it includes the order id as a promoted property to serve as correlation id. The orchestration then sends the shipping request through the messaging platform, which reads the promoted properties that are being used as correlation id. In the future, any incoming message that arrives at the messaging platform having the same values for the same promoted properties will be forward to the same orchestration instance.

The scenario in Fig. 4.5 can therefore be implemented in the following way:

1. In the initial order message that triggers the whole orchestration, provide a unique order id value that can be used in correlations within the orchestration.
2. In the schema for the shipping request to be sent to the logistics provider, include an additional element to store the order id. Also in the schema for the shipping response, include an additional element to store the order id.
3. Create a property schema to define the promoted property that will be used as correlation id. We assume that this promoted property will be called `OrderId`.

4. Promote the order id elements of step 2. In both the shipping request and the shipping response, establish a relationship to the OrderId property defined in the property schema created in step 3.
5. In the orchestration, create a new correlation based on the OrderId property. This correlation will be used at two points in the orchestration: when the shipping request is being sent, and when the shipping response is being received.

In the BizTalk platform, step 5 is a bit more intricate because it requires several sub-steps. In particular, BizTalk makes a distinction between the concepts of *correlation type* and *correlation set*. Basically, a correlation type is an artifact that specifies the properties on which the correlation will be based. In this example, there is a single property called OrderId. So, by defining a correlation type with OrderId, it becomes possible to create correlations based on that property. To create an actual correlation that can be used in the orchestration, it is necessary to define a correlation set. Therefore, a correlation set can be seen as an instance of a previously defined correlation type. In general, it is possible to create several correlation sets from the same correlation type; this would mean having several correlations across the orchestration, all of which are based on the OrderId property.

After creating the correlation type and at least one correlation set, which represents the actual correlation, the next sub-step is to indicate where the correlation will be used in the orchestration. The point where the correlation is first used is the point where the correlation set will be *initialized*. In this example, the correlation set will be initialized when the shipping request is being sent. Subsequent actions within the same correlation are said to *follow* the same correlation set. In this example, the correlation set will be followed when the shipping response is being received. No other actions in this orchestration initialize or follow correlations.

Although the correlation is being initialized when the shipping request is sent, it would be possible to initialize it even earlier, at the point where the initial order is received. This is because the order id is assumed to be available at that point, so it would be possible to initialize the correlation set immediately. On the other hand, it would also be possible to extend the use of the same correlation to later actions, such as to the point when the order confirmation is sent to the customer. In other words, all the actions in the orchestration of Fig. 4.5 could be performed within a correlation, although that is not necessary. The point at which the correlation is strictly required is when receiving the shipping response, since at this point it is necessary to determine the orchestration instance that the response is intended for. For this to work, the correlation must have been initialized previously, so it is required also when sending the shipping request. However, nothing precludes the same correlation to be used at other points in the orchestration as well.

This shows that the use of correlations is not confined to request–response scenarios, and in fact they can be extended to include any set of interactions with the outside world. Examples are a request that results in multiple responses, or a sequence of several request–response pairs within the same correlation.

Another important and often misunderstood issue has to do with receive actions. The first receive in an orchestration is the action that triggers the orchestration,

so that when a message is received, a new orchestration instance is created. As explained in Sect. 2.4, such kind of receive is called an *activating* receive. Clearly, no actions can precede an activating receive, since before that the orchestration has not been instantiated yet. However, after the activating receive it is possible to have other receive actions, as in the example of Fig. 4.5 where the orchestration receives the shipping response. Obviously, such kind of receive is non-activating, since the orchestration instance is already running. Now, if a receive is non-activating, then it must follow a correlation; otherwise, the messaging platform would be unable to forward messages to that running orchestration instance.

The conclusion is that a receive action must either be the initial, activating receive in an orchestration (and there is a single activating receive in an orchestration), or be a non-activating receive which follows a correlation (several such receives may exist in an orchestration). In the latter case, the correlation must have been previously initialized by a previous action, usually a send. Developers often find it strange that they have to make such an elaborate decision when using something as simple as a receive action. Hopefully, this section has contributed to clarify why such decision is necessary: it has to do with the fact that several instances of the same orchestration may be running at the same time on top of the messaging platform. The activating receive creates such instances; the non-activating receive requires correlation to determine which instance a message is intended for.

4.7 Using Asynchronous Messaging

In the previous sections we have explained that a message broker is a combination of a messaging platform and an orchestrator, such that the orchestrator works on top of the messaging platform, i.e., it coordinates the exchange of messages between applications connected to the messaging platform. In this context, the messaging platform has been presented as a messaging system similar to those described in Chap. 3. In practice, however, the features provided by a messaging system and by the messaging platform of a message broker are slightly different.

On one hand, there is no need for a message broker to replicate the functionality and reliability mechanisms of a messaging system; if such capabilities are required, the message broker can be integrated with an existing messaging system, such as a JMS provider or MSMQ. On the other hand, the message broker extends the capabilities of traditional messaging systems by making use of the concepts of receive and send ports. Receive ports are entry points of messages in the messaging platform, i.e., they are message publishers, and they are connected to applications that produce messages. Send ports are message subscribers, and they are connected to applications that consume messages. Therefore, a message broker works according to the publish–subscribe paradigm, while traditional messaging systems typically support point-to-point communication through channels or queues.

In addition, a message broker is able to coordinate message exchange between applications through an orchestrator, and for that purpose it is necessary to configure

the port bindings between the logical receive and send ports in an orchestration and the physical receive and send ports created or available at the messaging level.

Despite these differences, it is possible to use messaging systems of the kind described in Chap. 3 in combination with a message broker. For this purpose, it is possible to specify that a physical receive port is connected to a messaging system, so that it receives messages from a message queue. Similarly, it is possible to specify that a send port is connected to a messaging system, so that it sends messages from a message queue. The queue from which messages are received and the queue to which messages are sent may be in the same or in different message systems. In the latter case, one can see the potential of using a message broker to create a bridge between different messaging systems.

In Sect. 2.3 (Fig. 2.4) we have seen that both receive and send ports comprise an adapter, a pipeline, and an optional transformation map. (Furthermore, in Sect. 4.3 we have seen that a send port may contain an optional message filter as well.) The adapter in a port is the component that specifies the protocols and parameters to connect to the source (in case of a receive port) or the target application (in case of a send port). As explained in Sect. 2.3, a message broker typically includes a set of predefined adapters, so configuring the adapter for use in a port becomes a matter of selecting one of the existing adapters. In particular, there are adapters for messaging systems. For example, BizTalk includes an adapter for MSMQ that allows receive ports and send ports to connect to message queues.

To configure the MSMQ adapter in a receive port, one basically specifies the queue name and whether the message should be received within a transaction or not. When configuring the MSMQ adapter in a send port, a lot more options are available; besides the queue name and transaction support, one can specify the priority of the message to be sent, whether it should be sent in express or recoverable mode, whether acknowledgments should be generated, etc. In general, all those message properties that can be configured by a sending application using MSMQ can also be configured in the MSMQ adapter of a send port.

However, there is one important difference in an application that communicates with BizTalk through MSMQ, as opposed to an application that communicates with another application through MSMQ. When two applications exchange messages through MSMQ, as described in Sect. 3.6, it is possible to use the Body property. In this case, the sender application sets the Body property and the receiver retrieves the content of that property. During transmission, MSMQ automatically formats the body content into an XML message (an example is shown in Listing 3.7 on page 69). When using an integration platform such as BizTalk, such automatic formatting interferes with the processing of messages because the message broker is expecting an XML message with a user-defined schema, rather than the schema used by MSMQ. In fact, in integration platforms such as BizTalk, developing a new solution begins by defining the schemas of messages that will go across the message broker. If MSMQ is allowed to wrap the message content into a new, previously undefined schema, then the message broker will be unable to handle the message correctly.

Listing 4.1 Application code to send a message to BizTalk through MSMQ

```
1 string requestMsg = File.ReadAllText("PurchaseRequest.xml");
2
3 string queueName = @"\\.private$\requests";
4 MessageQueue queue = new MessageQueue(queueName);
5
6 Message queueMsg = new Message();
7
8 StreamWriter writer = new StreamWriter(queueMsg.BodyStream);
9 writer.Write(requestMsg);
10 writer.Flush();
11
12 queue.Send(queueMsg);
```

Clearly, something must be done in order to avoid having MSMQ interfere with the message content. In particular, the message body must be written to the message in such a way that it reaches the message broker without having been changed or reformatted. The solution to this problem is to write the message body through the `BodyStream` property rather than through the `Body` property. When writing to the `BodyStream` property, the content is written directly to the message body, as if it would be a file or stream. At the receiving end, the message broker reads the message body data through the `BodyStream` property as well. This way, it is possible to have the message content arrive at the message broker intact.

The only problem with this approach is that the source application must guarantee that the body content adheres to the schema that the message broker is waiting for. This is easy to achieve if the source application has access to an example of the message to be sent. For this purpose, it is possible to create a sample instance from the predefined schema and provide it to the source application, so that this application has only to modify the data elements and send the whole message to the message broker, in the body stream. The procedure is illustrated in Listing 4.1.

This example is based on the same scenario as in Sect. 4.4, particularly Fig. 4.4 on page 82. Here we suppose that the physical receive port in Fig. 4.4 is connected to a message queue, and Listing 4.1 shows the application code for the source application that sends a message to that queue. For this purpose, we assume that an instance of the purchase request schema is available in an XML file (line 1). For simplicity, we also assume that such instance already contains the correct data to be sent. Now it is a matter of opening the specified queue (lines 3–4), creating a new message (line 6), writing the message body (lines 8–10), and sending the message (line 12). The main difference to the example of Listing 3.6 on page 69 is that the message body is being written directly as a stream through the `BodyStream` property. Line 8 opens the stream, line 9 writes the content, which comes from line 1, and line 10 ensures that the content has actually been written to the message object before sending it on line 12. This way, the message broker (BizTalk) will receive the exact same content as in the XML file specified in line 1.

If the `Qty` element in the purchase request contains a value that is less than or equal to 500, then the orchestration in Fig. 4.4 will approve the request and forward it to the ERP system (right-side branch). Assuming that the send port is connected to

Listing 4.2 Application code to receive a message from BizTalk through MSMQ

```
1 string queueName = @".\private$\requestsaccepted";
2 MessageQueue queue = new MessageQueue(queueName);
3
4 Message queueMsg = queue.Receive();
5
6 StreamReader reader = new StreamReader(queueMsg.BodyStream);
7 string requestMsg = reader.ReadToEnd();
8
9 Console.WriteLine(requestMsg);
```

a message queue, then the application code at the receiving end should do something similar to Listing 4.2 to retrieve the message and its content.

In lines 1–2 the application just opens the message queue where the message sends the message to. In line 4, there is a synchronous receive, but an asynchronous receive, as in Listing 3.9 on page 71, could be used as well. The important feature is in lines 6–7, where the application reads the message body. Again, this is done by accessing the body content as a stream, through the `BodyStream` property. In this example, the application just reads the entire body content (line 7) and writes it to the command line. A real ERP system would probably store the purchase request in a database or handle it in some other way.

4.8 Conclusion

Message brokers provide the possibility of implementing the integration logic outside the applications and in the integration platform itself. This provides a much-desired centralized point of control, where the integration logic can be maintained and changed according to business needs. A message broker differs from a traditional messaging system in that it makes extensive use of the publish–subscribe paradigm, and it includes an orchestrator to coordinate message exchanges between applications. The desired behavior can be implemented by means of orchestrations which define the routing of messages between receive ports and send ports in the underlying messaging platform.

Although orchestrations represent a process-oriented and flexible way to implement the integration logic, it is possible to integrate applications through a message broker without the use of orchestrations. In this case, the integration solution will rely on the publish–subscribe mechanisms of the messaging platform alone, and for this purpose it becomes necessary to define message filters to specify which messages should be sent through each send port. The use of transformation maps in receive and send ports also allows messages to be transformed if necessary. A disadvantage of this approach is that the overall behavior is embedded in the messaging platform and is not as easy to understand as in the case of an orchestration.

In conclusion, as far as the external applications are concerned, the same behavior can be implemented at the messaging level or at the orchestration level. For solutions at the messaging level, it becomes necessary to promote certain message properties in order to evaluate filter conditions and determine which send ports the message should be sent to. The message properties that must be promoted are the ones that are used to define the filter conditions. For solutions at the orchestration level, it is necessary to promote certain message properties as well, in order to determine which orchestration instance the message should be sent to. In this case, the message properties that must be promoted are the ones that are used to establish a correlation with a previous message belonging to the same orchestration instance.

The last section (Sect. 4.7) has shown how to connect a message broker with a messaging system, so that the broker receives messages from and sends messages to a message queue, which provides asynchronous interaction with the external applications. The same section also explained how the applications should write and read the message body in order to exchange messages with a predefined schema.

In the same way as messaging systems can be connected to a message broker, other kinds of systems can be connected to a message broker as well. We have already discussed that such connection can be made through the use of adapters in receive and send ports. In the next chapters, we will have a closer look at the concept of adapter and at the way other systems—namely databases and Web services—can be invoked from within an orchestration running in the message broker.



<http://www.springer.com/978-3-642-40795-6>

Enterprise Systems Integration

A Process-Oriented Approach

Ferreira, D.R.

2013, XIV, 387 p. 147 illus., Hardcover

ISBN: 978-3-642-40795-6