Understanding Intermediate Events, Asynchronous Message Receipt and Correlation in NetWeaver BPM 7.20

Applies to:

NetWeaver BPM 7.20. For more information, visit the Business Process Expert homepage.

Summary

BPMN's Intermediate Message Events add correlation-based asynchronous message receipt to the NetWeaver BPM 7.20 feature set. As such, they implement the "Persistent Trigger" workflow pattern and are a prerequisite to synchronize a process instance to events happening in surrounding components and applications. This article elaborates on the principles of message correlation and receipt using Intermediate Message Events and introduces best practices to implement common scenarios such as process conversations, message collection and broadcast.

Author: Sören Balko

Company: SAP AG

Created on: 20 July 2010

Author Bio

Sören joined SAP in 2006 to help bootstrapping their recently released NetWeaver BPM product, code-named "Galaxy". He is a senior member of the NetWeaver BPM Architecture and Innovation team where he is working on a number of topics such as collaborative, on-demand business process modeling ("Gravity") and designing Galaxy's runtime, build, and lifecycle management components. Sören holds a doctorate in Computer Science from Magdeburg University (Germany) and has since worked at various international companies and institutions, including ETH Zurich (Switzerland) and SAP Research Brisbane (Australia).

Table of Contents

Introduction

With the introduction of Intermediate Message Events into the NetWeaver BPM 7.20 feature set, we provide for the means to synchronize a process to its environment. Intermediate Message Events are process steps where the respective process instance waits for a message to come in before the flow commences on the respective control flow branch. As such, they cater for a variety of use-cases, including (1) conversations between the process and another component, (2) intra-process communication, and (3) message collection and aggregation.

Technically, Intermediate Message Events reference message "triggers" that can be shared between multiple Intermediate Message Events (or even Start Message Events as we will see further below). A message trigger is a reusable entity that represents a single Web Service endpoint URL at runtime. As such, it relates to the WSDL portType and operation that is exposed behind that endpoint URL.

Consuming that Web Service will deliver the corresponding request message to the process server where it is dispatched to those process instances that contain an Intermediate Message Event for that trigger and where a so-called correlation condition holds. Roughly spoken, the correlation condition is a Boolean predicate atop the message "payload" and the process' data context that determines whether or not a message will be "matched".

It is important to understand that receiving a message happens in multiple transactions ("matching" and "delivery") that are asynchronously de-coupled and may happen at different points in time:

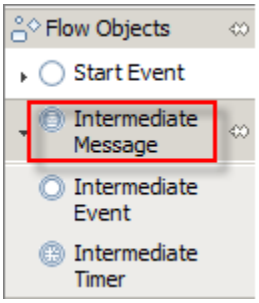
- **Matching:** The "matching" transaction is where the message is delivered to the process server. If message delivery happens through *WS-ReliableMessaging*, that transaction is initially opened by the SOA stack (otherwise, the process server will autonomously designate a new transaction for that purpose). In that transaction, an incoming message is "matched" against the correlation conditions of *all* process instances that contain an Intermediate Message Event for the corresponding message trigger. If at least one process instance "matches" the message (i.e., the correlation condition of its Intermediate Message Event evaluates to "true"), it is queued for being delivered to that process instance in the "delivery" transaction.
- **Delivery:** The "delivery" transaction is where the process server actually executes the Intermediate Message Event, i.e., receives the message, output maps its payload to the process data context, and proceeds on the respective control flow branch. If no ready-to-receive (matched) message currently exists in the queue, that process branch will wait for that message to come in and be matched. Also note that multiple process instances may have matched a single message (broadcast). In that case, the message will be delivered to all of those process instances in multiple "delivery" transactions which may, again, happen at different points in time.

The key takeaways for you is that (1) Intermediate Message Events will also receive messages that have come in before the process has progressed to the Intermediate Message Event but (2) Intermediate Message Events will also wait (i.e., suspend the flow on the respective branch) for a matching message if none is currently queued.

Note: Intermediate Message Events start matching incoming messages as soon as the containing process instance is started. Any message that comes in after the instance was spawned will attempted to be matched against the correlation condition of the Intermediate Message Event.

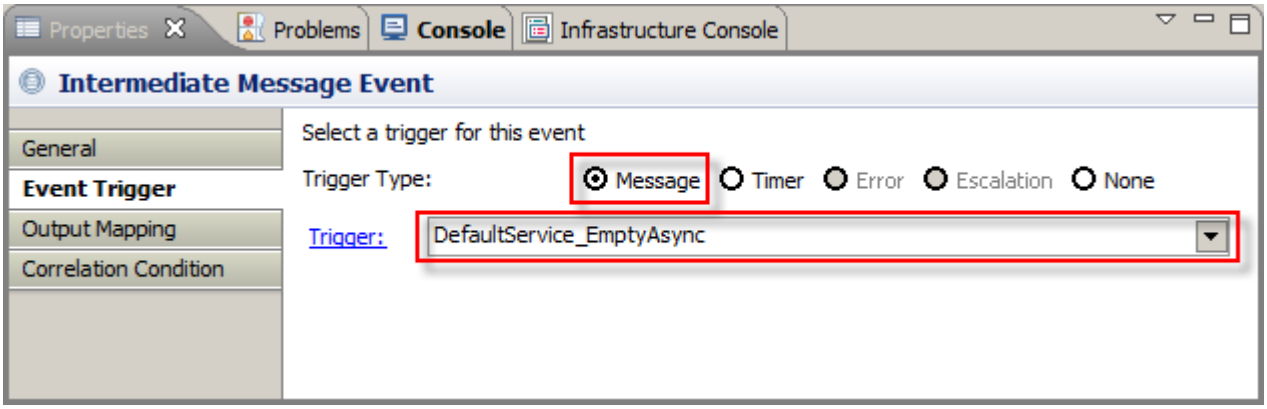
Modeling and Configuration

In the palette, Intermediate Message Events can be found below the "Intermediate Event" item. As usual, you can add them to a process model diagram both by dragging it from the palette or the speed buttons of a predecessor model element. Intermediate Message Events must both be connected to a predecessor and a successor model element and must not be placed into embedded subflows. Apart from that, there are no restrictions where to place an Intermediate Message Event into your process model. Moreover, you can have arbitrarily many Intermediate Message Events per process model, both on the same control flow edges.



Intermediate Event Properties

In the corresponding property sheet, you will need to configure (1) a message trigger and (2) a correlation condition. Conceptually, a message trigger represents the endpoint URL for a Web Service. Again, when consuming that Web Service (i.e., invoking the corresponding operation from within a Web Service client), the corresponding request "message" will be dispatched (i.e., matched and eventually received) to all process instance containing an Intermediate Message Event that refers to this trigger.



Please note, that intermediate events also support other trigger types such as timer events, please choose "Message" for Intermediate Message Events. From the drop-down list below, pick one message trigger that relates to an **asynchronous** Web Service operation. As Intermediate Message Events implement asynchronous message receipt, only, you cannot use message triggers that refer to a synchronous operation.

Note: Intermediate Message Events are for receiving asynchronous messages ("In-Only" and "Robust In-Only"), only. Other message exchange patterns (MEP) are not supported by Intermediate Message Events. Technically, the respective message trigger must refer to an asynchronous WSDL operation (which does not have a response parameter).

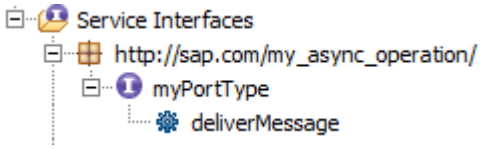
Message Triggers

In most cases, you will want to define your own message trigger. To do so, initially you need to import or create a WSDL file defining the portType and operation for that trigger. To do so, right-click on *Service Interfaces* in the *Process Modeling* tree and choose from "New WSDL…" and "Import WSDL…". Again, make sure that you create or import a WSDL containing an asynchronous operation:

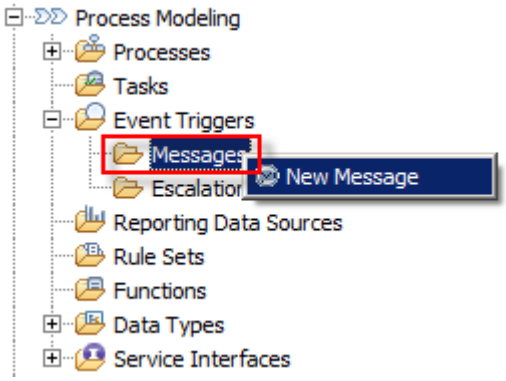
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<wsdl:definitions
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:tns="http://sap.com/my_async_operation/"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 name="my_async_operation"
 targetNamespace="http://sap.com/my_async_operation/">

 <wsdl:types>
  <xsd:schema targetNamespace="http://sap.com/my_async_operation/">
   <xsd:element name="message">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="id" type="xsd:int" />
     </xsd:sequence>
```

```
      </xsd:complexType>
     </xsd:element>
    </xsd:schema>
   </wsdl:types>

   <wsdl:message name="deliverMessageRequest">
    <wsdl:part element="tns:message" name="parameters" />
   </wsdl:message>

   <wsdl:portType name="myPortType">
    <wsdl:operation name="deliverMessage">
     <wsdl:input message="tns:deliverMessageRequest" />
    </wsdl:operation>
   </wsdl:portType>

  </wsdl:definitions>
```

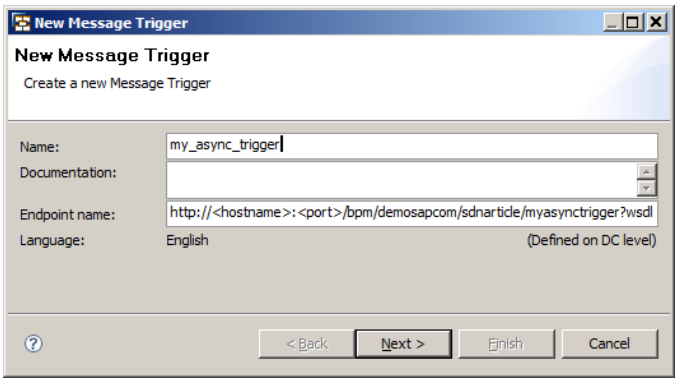
In the example scenario, we have defined a WSDL file containing a *portType* “myPortType” having a single *operation* “deliverMessage” specifying a body that comprises two parameters: “name” (string) and “id” (integer). After saving the WSDL file, this *portType* and *operation* appear below the *Service Interfaces* folder:

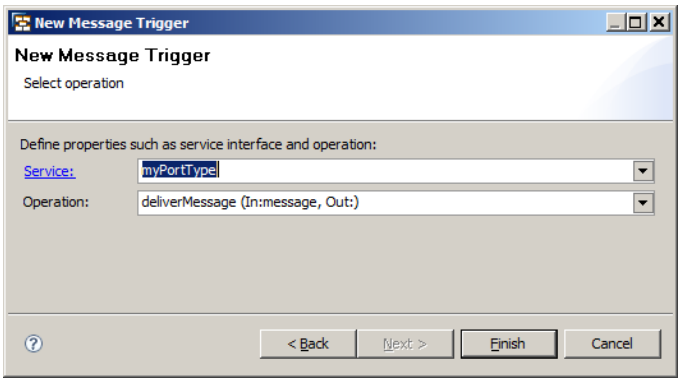


You are then ready to create a new message trigger based on these WSDL entities. To do so, open the *Event Triggers* folder in the *Process Modeling* tree (Eclipse *Project Explorer* view). Then right-click on *Messages* and choose *New Message*.



In a wizard-style dialog, you can name the newly created trigger and choose the *portType* and *operation* pair you have created before. Please note, that the WSDL file may optionally reside in another *Development Component* and you may still use it in a local message trigger. For that purpose, you need to specify a dependency to the other *Development Component* where the WSDL resides in.





Once you have defined the message trigger, it is ready to be referenced in the Intermediate Event Properties. Once again, you can also use message triggers from other referenced *Development Components*.

Note: Message triggers can be reused in multiple Intermediate Message Events which may both reside in the same process model or in different process models which may, in turn, belong to different *Development Components*. If message triggers are reused in different process models, the process server broadcasts messages to the respective process instances. Inside a single process instance, a message will only be delivered to the (single) Intermediate Message Event for that trigger that is reached first.

Correlation Conditions

Intermediate Message Events should formulate a correlation condition which is a Boolean test that determines whether or not a message will be “matched”, i.e., queued for the respective process instance. Correlation conditions may refer both to the message payload (here: the “name” and “id” elements) and the process data context (all data objects in the process model). Conceptually, correlation conditions are constrained to equality checks where the left (right) side is an expression exclusively relating to the message payload (the process context).

That is, these expressions may be arbitrarily complex and incorporate functions (including custom mapping functions, literals, and business rules). Suppose, the process context consists of a single string-typed data object “context”. The following examples are each valid correlation conditions:

```
true
message/name=context
message/name=substring(context,0,3)
concat(message/name,string(message/id))=substring(context,0,3)
message/name=context AND message/id=4711
message/name=context AND message/id=4711+string-length(context)
```

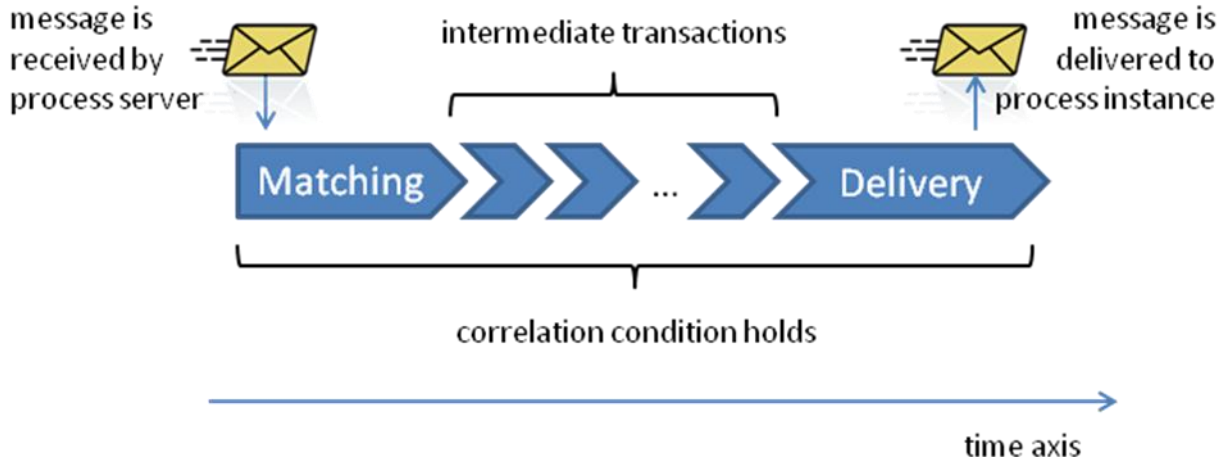
As exemplified by the latter two cases, correlation conditions can also compare complex keys whose fields are made up by expressions on the message payload and process context, respectively.

Note: A correlation condition is a logical test that can refer to the message payload and the process data context, respectively. It allows for equality comparisons between arbitrary complex expressions atop the respective contexts. To match any message coming in for a trigger, you should set the correlation condition to “true”.

An Intermediate Message Event’s correlation condition defaults to “true”. If the correlation condition is simply “true” (“false”), any (no) message for that trigger will be matched. If a correlation condition simply compares fields from the message payload to constants (e.g., “message/id=4711”), it merely acts as a filter and does not depend on the current state of the process context. In some boundary cases, a correlation condition could also relate to the process context, only (e.g., “context=’foo’”). In that case, the matching behavior is essentially de-coupled from the message’s content and will only take the current state of the process context

into account. However, most frequently you will want to formulate a correlation condition that compares some field(s) from the message payload to the process context. In that case, a message is initially matched before if and only if the correlation condition holds against the state of the process context at the time the message is received (“matching” transaction).

Please note that in order to successfully deliver the message to the Intermediate Message Event, the correlation condition needs to permanently hold until the control flow reaches the Intermediate Message Event (“delivery” transaction). If between the two transactions the process context is changes in a way that the correlation condition does no longer hold, the message is “un-matched” and removed from the queue. In effect, it will not be delivered.

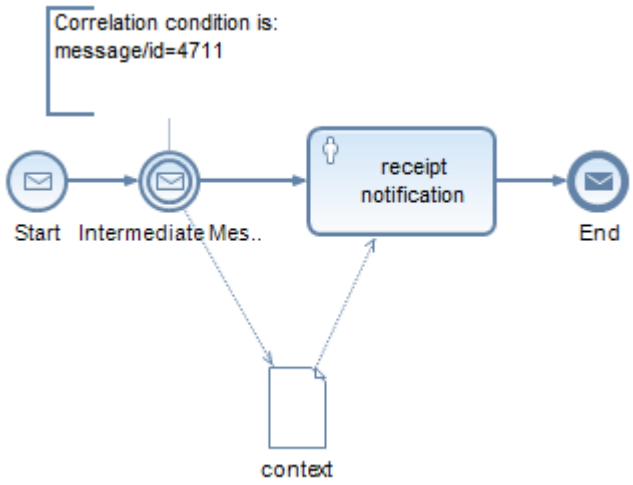


You may wonder what happens if between matching and delivery, the correlation condition is first invalidated and later holds true, again. The answer is that we do neither guarantee that the respective message (which was initially matched) is delivered to the process instance or not. It merely depends on whether other process instances exit that match the message such that it is not purged from the queue when the correlation condition is invalidated for the given process instance. In other words, you must not rely on the fact that simply “flipping” a correlation condition from true to false and back will stop a message from being delivered.

Note: You should always make sure that a correlation condition permanently holds between matching and delivery. That is, you have to be mindful when mapping to those data objects in the process data context which are referenced from the correlation condition to not invalidate the correlation condition. If a correlation condition is invalidated and remains “false” until the flow can no longer reach the Intermediate Message Event (like in a timeout scenario, see below), the message will never be delivered.

Web Service Endpoint URL

Let's have a look into a very simple process making use of an Intermediate Message Event. Here, the idea is to filter for any message whose "id" field is "4711". In this example, messages are not matched against the process context (we will later have examples where we evolve this scenario into process context matching). After having received a single message, a user task will show the message's payload (i.e., the "name" field).



In the message trigger wizard, the schematic URL for the corresponding Web Service endpoint is displayed at the bottom of the first page:

http://<hostname>:<port>/bpm/demosapcom/sdnarticle/myasynctrigger?wsdl

As you can see, we integrate the names of the Development Component and message trigger into the URL. You will need to replace `<hostname>` and `<port>` with the IP address or host name of the process server and the HTTP port number, respectively. The latter is by default formed from 5XY00 where XY is the two-digit instance number (if the instance number is below 10, you will need to prefix it with "0"). After deploying the *Development Component* and starting the process, it waits for a message to come in on the aforementioned URL.

Note: Web Services that are exposed by NetWeaver BPM processes (both for start and intermediate events) are secure. An authenticated user who plans to invoke any of these Web Services needs to have the SAP_BPM_SuperAdmin role assigned. Otherwise, the request message will not be received by the process server.

After having received a message having an "id" field value of "4711", the process progresses to the user task where it displays the value of the "name" field.

Reliable Messaging Configuration

Different scenarios using Intermediate Message Events may go along with varying consistency requirements. NetWeaver BPM comes with a Web Service stack supporting the *WS-ReliableMessaging* (WS-RM) protocol. That is, the underlying SOA layer may optionally be configured in a way such that a message that is sent by a Web Service client is guaranteed to be received by the Process Server. Accordingly, a Web Service that is provided by an Intermediate Message Event may have different "delivery assurances" assigned. Specifically, we support both unreliable ("best effort") and reliable ("exactly once") message delivery. The former is the default behavior if you define your WSDL file as shown before.

Many scenarios require reliable message receipt which caters for a consistent synchronization with the sender of a message. To enable "exactly once" message receipt, some configuration needs to happen both at design time and at runtime.

For one, the WSDL file defining the service interface (*portType*) of an Intermediate Message Event needs to be decorated with a “policy”, where some WS-RM options are defined. The following XML snippet needs to go into the `<wsdl:definitions>` section of your WSDL file:

```
<ns1:UsingPolicy
 xmlns:ns1="http://schemas.xmlsoap.org/ws/2004/09/policy"
 wsdl:required="true" />

<ns2:Policy
 xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/09/policy"
 ns3:Id="my_policy"
 xmlns:ns3="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd">

 <ns4:enableCommit
  xmlns:ns4="http://www.sap.com/NW05/soap/features/commit/">
true
 </ns4:enableCommit>

 <ns5:enableBlocking
  xmlns:ns5="http://www.sap.com/NW05/soap/features/blocking/">
  false
 </ns5:enableBlocking>

 <ns6:required
  xmlns:ns6="http://www.sap.com/NW05/soap/features/transaction/">
yes
 </ns6:required>

 <ns7:enableWSRM
xmlns:ns7="http://www.sap.com/NW05/soap/features/wsrm/">
true
 </ns7:enableWSRM>
</ns2:Policy>
That policy (“my_policy”) is then referenced from within the <wsdl:portType> section
(before the <wsdl:input> element) which looks like this:

<wsdl:portType name="myPortType">
 <wsdl:operation name="deliverMessage">
  <ns8:Policy xmlns:ns8="http://schemas.xmlsoap.org/ws/2004/09/policy">
   <ns8:PolicyReference URI="#my_policy" />
  </ns8:Policy>
  <wsdl:input message="tns:deliverMessageRequest" />
 </wsdl:operation>
</wsdl:portType>
```

Note: There is currently no dedicated tool and validation functionality available to support the manual modification of the WSDL files. Make sure to follow the given instructions with greatest care. Using WS-RM has a performance impact. The protocol requires additional database persistency and acknowledgments being exchanged between senders and receivers. SAP NetWeaver BPM support WS-RM 1.0, not WS-RM 1.1. To set up communication with for example with SAP NetWeaver Process Integration, use the PI WS adapter version 7.10. In AS ABAP you can select the WS-RM version in transaction SOAMANAGER. WS-RM cannot be used for local transport in one physical system.

Features

Now that you know the basic principles of Intermediate Message Events, let us proceed to its detailed features which we have only briefly touched upon before. You will need to understand those features when building up complex system-to-system integration and communication scenarios using Intermediate Message Events.

Exactly-Once Message Delivery

The most important feature of our Intermediate Message Event implementation is an “exactly-once” message delivery per process instance. Basically, we take care of neither loosing nor repeatedly delivering a message to a process instance such that any message that is ready to be received will also be delivered to the process instance. In detail, that requires (1) a message being matched (i.e., the correlation condition needs to hold true) and (2) the process ultimately reaching the corresponding Intermediate Message Event. Please be aware of the fact that a single triggering of an Intermediate Message Event will deliver a single message to the process instance. If multiple messages are queued up to be delivered, the corresponding Intermediate Message Event will need to be triggered as many times as there are messages waiting in the queue.

Note: Messages that were matched after starting a process instance will be queued for being delivered to the Intermediate Message Event(s) in the respective process instance. In turn, that requires the process to reach (pass a token to) the Intermediate Message Event(s) to ultimately deliver the message. Each triggering of an Intermediate Message Event will consume a single message from the queue where we do not provide for an “in order” delivery of those messages. Also note that even when consuming a message from the queue, that message is still queued for other matching process instances.

Have a look at the example flow below which is a plain sequence comprising a start event, an activity “Subflow 1”, an Intermediate Message Event, another activity “Subflow 2”, and an end event. Messages that come in after the process instance was kicked off and until it is terminated will be matched. A message that is received when “Subflow 1” is being processed will then be delivered to the “Intermediate Message Event”. If more than one message arrives during that time, only one of them will be delivered. The same is true for messages that come in after the flow has progressed beyond the Intermediate Message Event. These messages will still be matched. But because the flow never reaches the Intermediate Message Event, again, they can never be delivered.

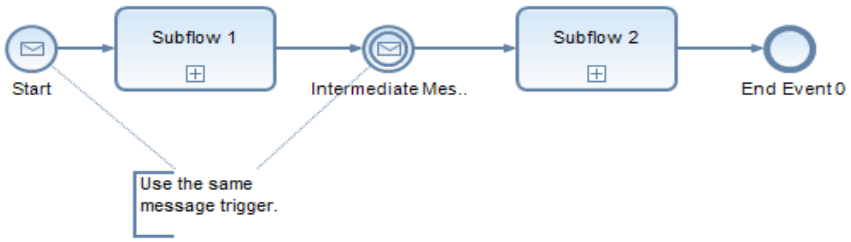
So what happens to those messages? For as long as the process instance has not terminated and continues to match the message (i.e., the correlation condition holds true), they will stay in the queue. And as always, other process instances which concurrently match the same message will still be able to get that message delivered.

Conditional Start

But what if you want a new instance of the same process model to receive this message? As I have pointed out before, a message is normally matched for the lifetime of a process instance. Unless there are other matching process instances at the same time, the message will be purged from the queue as soon as the process has terminated. If you want to avoid this from happening, you can make use of a feature we call “conditional start”. Basically, conditional start broadens our definition of “exactly-once” message delivery to whole process definitions. That is, the usual “exactly-once” guarantees (no message loss, no duplicate delivery) apply to all instances of a given process definition. Conditional start is simply enabled by using the same message trigger for the process’ start event and one (or multiple) contained intermediate events.

Note: Conditional start is a feature that allows you to define an “exactly-once” message receipt across all instances of a given process definition. It conditionally starts a new instance of that process definition if and only if there is no matching instance for that definition.

Have a look into the example process below which is a slightly altered variant of the process we have introduced before. It differs in using one and the same message trigger for both the start and the intermediate event. As message triggers used in intermediate events must use asynchronous Web Service operations, the process’ itself is started asynchronously.



Matching messages that cannot be delivered to an instance of that process will be queued and delivered to a new instance of that process definition. That is, the start event of that new instance consumes the message. In total, instances of that process get two messages delivered: one to start the process and another one for the Intermediate Message Event.

You may have noticed that start events can optionally be equipped with a start condition. Very much like correlation conditions in Intermediate Message Events, start conditions serve as a criterion to “filter” for certain messages. As the process data context does not yet exist when a process is about to be started, start conditions may exclusively refer to the message payload.

In the context of the conditional start feature, there are no constraints whatsoever regarding start conditions. That is, a message that is matched by a first process instance may or may not pass the start condition of the process. In effect, the once-matched message is deliberately lost and will not spawn a new process instance when not passing through the start condition. However, when making use of the conditional start feature, it is normally a good idea to not formulate conflicting start and correlation conditions.

Message Broadcast

Message triggers can be re-used, not only inside but across different processes. If a single message trigger is used by multiple start and/or intermediate events inside the same process definition, only one event will get the message delivered. It depends on the contextual situation which one will “win” and get the message:

- Start Event: A message will be delivered to the start event if the start condition holds and (1) no other instance of the corresponding process definition currently matches the message or (2) if a previously matched message is “unmatched” (conditional start semantics).
- Intermediate Message Event: A message will be delivered to an Intermediate Message Event if its correlation condition holds (i.e., it matches the message) and the control flow reaches that Intermediate Message Event before other matching Intermediate Message Event(s) in that process instance.

Conceptually, it is the aforementioned “exactly once” message delivery that applies in this case. When we talk about multiple process instances that currently match a message (no matter whether they relate to the same or different process definitions), a broadcast pattern applies.

That is, a message may be matched by more than one process instance. In effect, it may also be delivered to multiple process instances. In this way, one may independently define processes that “hook up” to the very same message trigger and receive copies of the same message, each.

Note: A word of caution is in order here. That is, you should re-use message triggers with care in practice. Doing so results in a broadcast pattern at runtime, meaning that a single message needs to be matched against multiple correlation conditions and may be delivered to more than one process instance. In essence, broadcasting a message to multiple processes goes along with noticeable performance penalties.

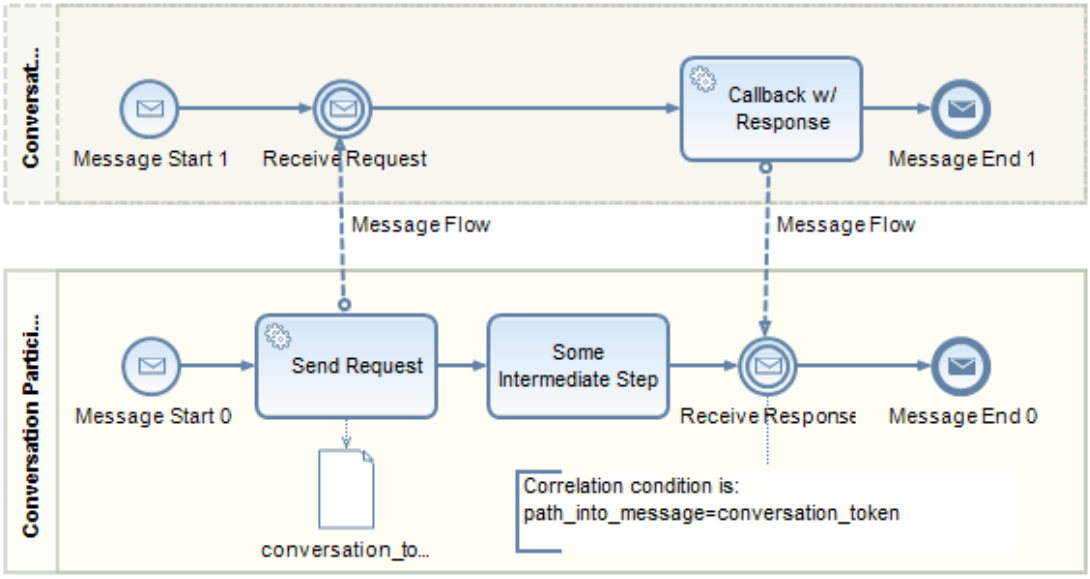
Best Practices

We are now ready to introduce some common use-cases for Intermediate Message Events with correlation and propose best practices to implement these.

Conversations

Typically, Intermediate Message Events will be used to establish bi-directional conversations to other processes or external components. Those conversations are not confined to a single message exchange but may entail multiple messages going back and forth in an arbitrary order. For that purpose, Intermediate Message Events are used in conjunction with Automated Activities which cater for outbound messages.

A single conversation may be identified by a conversation “token” (not to be confused which a control flow token which represents a thread of control) which is some unique key that is part of (or can be reconstructed from) a message’s payload. When a process initiates the conversation by sending a “request” message to the remote party, that process should generate the conversation token and store it in the process data context.



The example process illustrates a very simple conversation example where an initial outbound request is followed by an inbound response message. When sending the request, we populate a special-purpose data object “conversation_token” in the output mapping of the “Send Request” Automated Activity. The downstream Intermediate Message Event later refers to that data object in its correlation condition which extracts the counterpart from the incoming message by means of some custom expression.

Note**:** When implementing bi-directional conversation patterns between a process and some remote party (process, component, system, etc.) it is typically a good idea to introduce a special-purpose “conversation token” into the process context and make use of that data object to store a unique identifier for the conversation. Intermediate Message Events that belong to the conversation can easily reference the conversation token in their correlation conditions.

Be aware that this pattern may potentially result in a race condition if multiple control flow tokens enter the corresponding process fragment. In that case, a single “conversation token” may be overwritten when still in use if successor tokens map to the corresponding data object (“lost update” phenomenon). To avoid the latter, it may be a good idea to place the whole conversation into a separate process which is invoked from a subflow activity. In this way, each subflow instance has its own conversation data object.

Please note that (1) between sending the request message and receiving the corresponding response, an arbitrary number intermediate steps may occur and that (2) other message exchange patterns (such as initially receiving a request and responding with another message; receiving multiple responses, etc.) can easily be implemented with other combinations of Intermediate Message Events and Automated Activities.

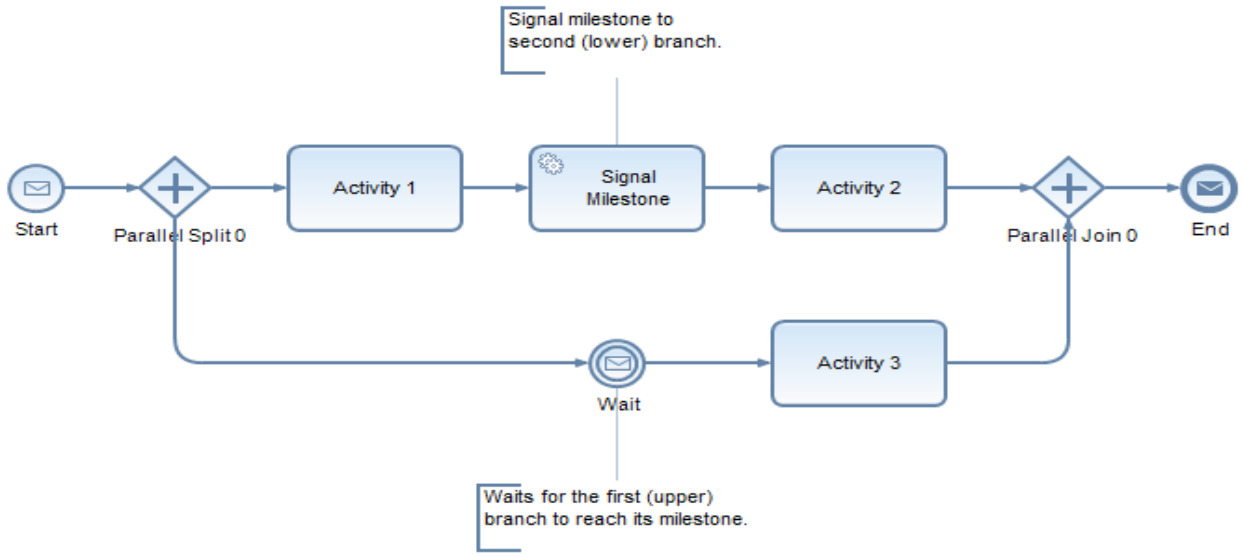
Intra-Process Communication

Intermediate Message Events may be used to implement various intra-process communication use-cases where otherwise unrelated process fragments need to exchange messages. Examples include:

- A parent process calling into a subflow when it has already been started (as needed for various “Cancellation and Force Completion” workflow patterns);
- A nested subflow instance calling out to some (grand-)parent process in the call stack without requiring boundary events;
- Parallel process branches synchronizing their state (which comes in handy for a straightforward implementation of the “Milestone” workflow pattern);

I have already discussed the first example in another blog article on workflow pattern improvements in NetWeaver BPM 7.20 (please refer to Section “Cancellation and Force Completion Patterns” for an example). The opposite concept of calling out from a subflow to some parent process in the “call stack” (i.e., the cascade of processes that have invoked one another up to the subflow instance) is actually very similar and basically corresponds to an Automated Activity placed into that subflow and sending a message that is received by an Intermediate Message Event in some parent process.

The “Milestone” workflow pattern actually synchronizes the progress on multiple parallel process branches. In effect, a first process branch indicates having progressed to some state to a second process branch which may only then proceed beyond a certain stage once. The example implementation in my article on workflow patterns made use of some explicit token bookkeeping and conditional gateways to decide whether or not to proceed downstream on the second branch. This implementation is confined to an interpretation of the “Milestone” pattern where the second branch immediately decides whether or not to proceed when it reaches a certain stage. In other words, the second branch does not “wait” for the first branch to reach the milestone. Thanks to Intermediate Message Events, waiting for the first branch to reach its milestone can now be implemented as illustrated below:

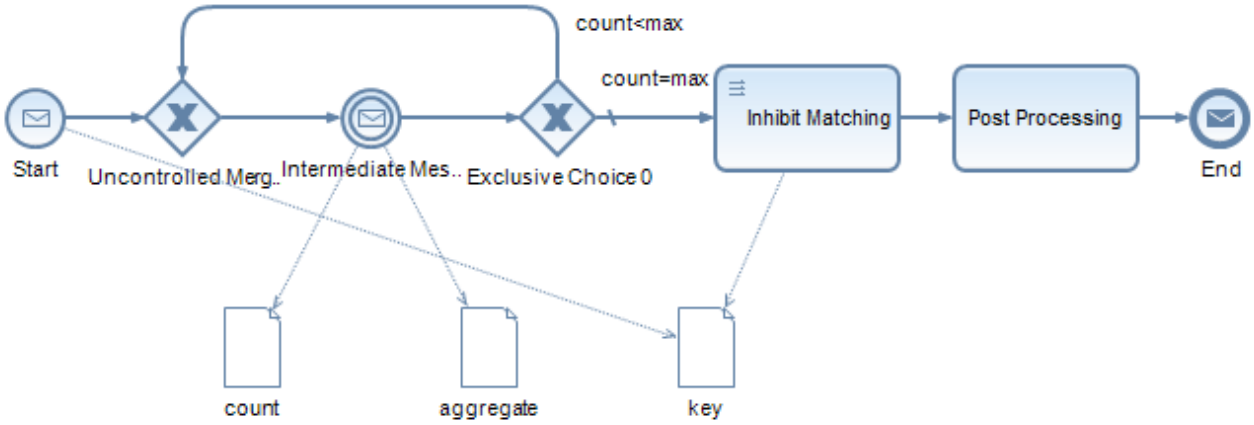


The first (upper) branch reaches the milestone after having completed “Activity 1”. The second (lower) branch waits for that milestone being indicated by an incoming message and only then starts processing “Activity 3”.

Note: Intermediate Message Events are a way of implementing intra-process communication. However, be aware that messages exchanged between process (fragments) take a whole roundtrip through the Web Services stack which is way more costly than other event types like escalation and error events. Whenever possible, use the latter event types instead. Also remember to make sure that an Automated Activity (which sends a message to an Intermediate Message Event) needs to be invoked by a principal who has the SAP_BPM_SuperAdmin role.

Message Collect

Sometimes, processes are meant to solely collect a number of messages from a single source, aggregate them, and initiate some action in another business system. Take a look into the example process below which leverages the “conditional start” feature to initially receive a message carrying some key which shall also serve as the joint correlation criterion for all follow-up messages.



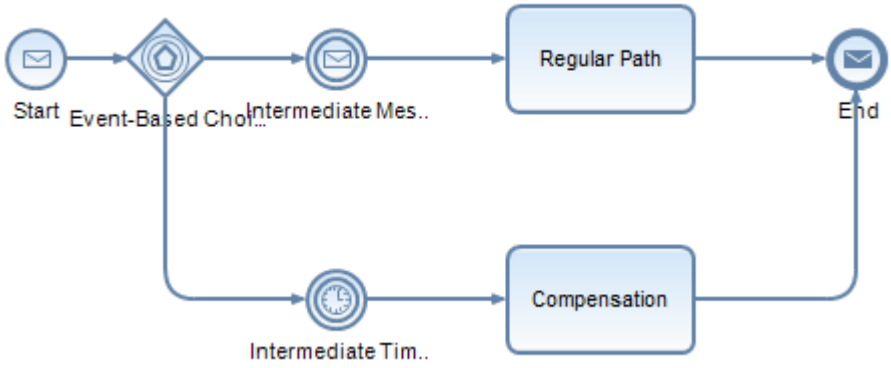
For instance, think of scenario, where the process server receives red, blue, and yellow messages. A single process instance is supposed to only collect messages of a single color which is determined by the first message that is received. When a message is delivered to the contained Intermediate Message Event, an integer “count” data object is incremented and the message payload (or parts thereof) are mapped to the “aggregate” data object (which could simply be a list). When “count” has reached some threshold value, the process breaks out of the loop and initially inhibits matching further messages (thus enabling newly incoming messages to start a new process instance) before some post processing of the aggregated messages happens. The latter typically entails sending the post-processed aggregate to another business system.

Note: Message collect scenarios can be implemented by a sequential loop pattern, iterating over an Intermediate Message Event. In order to avoid the matching queue from growing, no additional time-consuming steps should be placed into the loop (in particular no Human Activities) and further matching should be inhibited after breaking out of the loop. In particular, costly post processing activities should only happen after that stage.

A good example for a message collect pattern is a conveyor belt transporting goods to a packing station where the goods are loaded onto a pallet for shipment. On their way to the packing station, these goods pass by an RFID reader which scans the good identifier and category and feeds this data to the process server. A process may then keep track of which goods (grouped by category) of a certain category were already loaded and jointly shipped.

Timeout Patterns

Sometimes, messages may or may not be received by the process server. A process waiting for a certain message to be received may implement a timeout pattern to stop waiting after a period of time and compensate for the missing message in some appropriate manner. In conjunction with Intermediate Timer Events, the newly introduced Event-based Gateways serve this purpose. That is, Event-based Gateways implement the “Deferred Choice” workflow pattern providing for a “race” between multiple inbound events.



In the example process, it is either the message delivery which aborts the timer event on the lower branch or vice versa: the timeout signal stops the Intermediate Message Event from waiting for the message. That is, that both branches are mutually exclusive where one branch revokes the flow from the other branch upon activation (i.e., occurrence of the respective event).

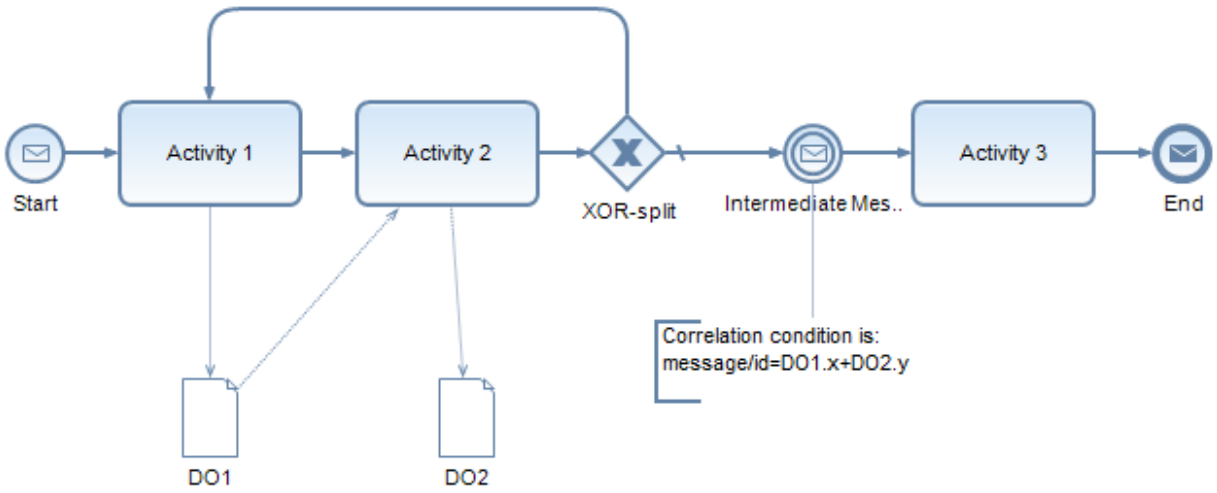
Note: The timeout patter is a great way of introducing a controlled handling and compensation of delayed messages. It is highly advisable whenever message receipt is uncertain and processes must not exceed a certain turnaround time because of waiting for messages at Intermediate Message Events.

Performance Tuning

Aside from using broadcast pattern rarely, there are other performance tuning techniques that you may want to make use of. As a guiding principle, you should be aware of the fact that matching is a costly operation. However, there are ways to both reduce the frequency of matching operations and also make matching more efficient as such.

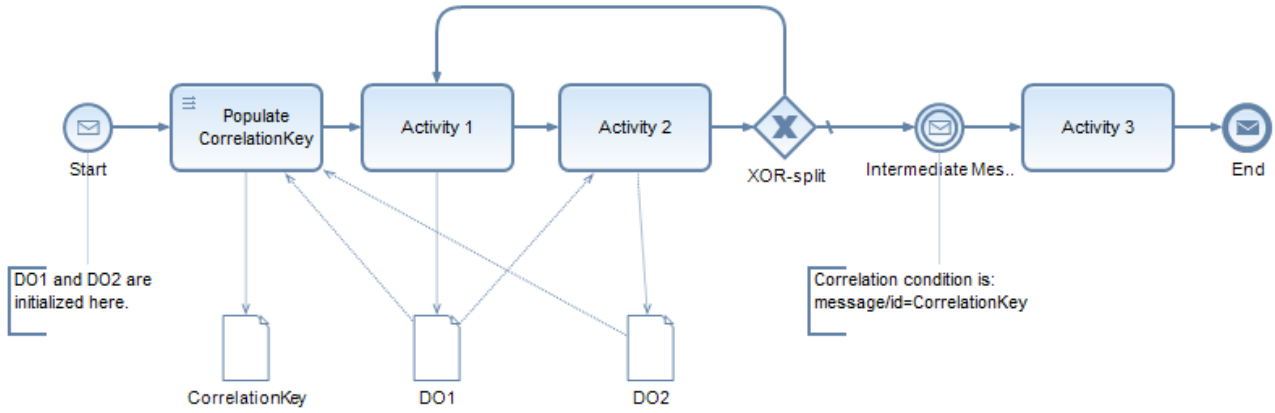
Correlation Condition Tuning

Correlation conditions base upon a number of data objects from the process context. Whenever these data objects are updated due to a mapping operation, the correlation condition has to be re-evaluated for all queued messages. In effect, these messages stay “matched” or are “unmatched”. That re-evaluation happens regardless of how the data objects were altered and whether or not the correlation is affected. This is why you should avoid updating data objects when there may still be matched messages queued up. Have a look into the example process below:



The contained Intermediate Message Event's correlation condition bases on two data objects "DO1" and "DO2". Those two data objects are updated by two upstream activities which are even placed into a loop. Now suppose a message is received and was successfully matched while the process is running these activities. Any time "Activity 1" or "Activity 2" update "DO1" or "DO2", that message is re-matched (or unmatched, respectively).

Despite the fact that "DO1" and "DO2" are presumably not even updated in a way that the message is unmatched, the correlation condition will be re-evaluated because there is no way of knowing in advance whether or not it will continue to hold. Moreover, a pattern where an initially matched message is deliberately un-matched before that message is delivered is suspicious, to say the least. In other words, you would not normally change the process context in a way that queued messages are intentionally un-matched (read further to see exceptions from this rule). If the correlation condition actually relates to attributes from "DO1" and "DO2" that are not affected by the output mappings of "Activity 1" and "Activity 2". In that case, it may be a good idea to introduce a special-purpose data object which contains the relevant keys from the process context and have the correlation condition use that data object:

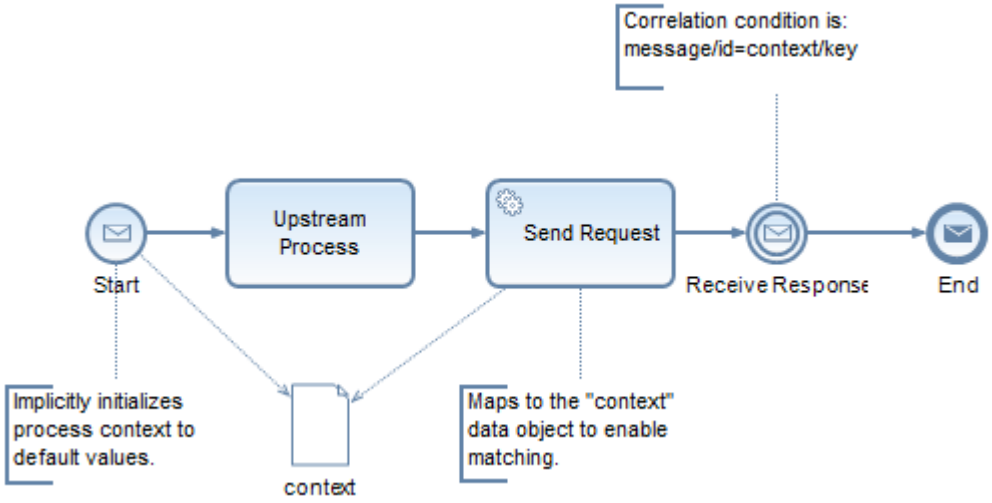


In the example process, we have added an integer-typed data object "CorrelationKey" which is populated from the expression "DO1.x+DO2.y" in a newly introduced mapping activity. The Intermediate Message Event's correlation condition was altered to exclusively use the "CorrelationKey" data object. In effect, updates to "DO1" and "DO2" will no longer cause the correlation condition to be re-evaluated for queued messages.

Note: For performance reasons, processes should not update a correlation condition's underlying process context if queued messages (messages that were matched by that correlation condition before) should not be deliberately un-matched.

Switching Off Matching

There are cases when we do want to break a correlation condition, tough. The idea is to either stop further messages from being matched and queued or to un-match messages and purge them from the queue. Let us discuss the former scenario first. When a process in engaged in a conversation that entails sending a request to some remote process or system and later receiving a response in turn, it does not make any sense to already match messages before the request was sent. In other words, we do want to deliberately inhibit matching before that step.



While you cannot change a correlation condition at runtime, you can alter the underlying process context in a way that the correlation condition does not hold. Have a look into the example process below where an Intermediate Message Event uses a correlation condition that refers to an attribute “key” in a data object “context”. When the process context is initialized in the process’ start event, all data objects are instantiated and pre-filled with default values (i.e., primitive types get the XSD defaults for the respective type assigned). Most likely, those initially assigned values will already inhibit matching. To be on the safe side, one could additionally fill the “context/key” attribute with some custom value that is known not to match any incoming message’s “id” field.

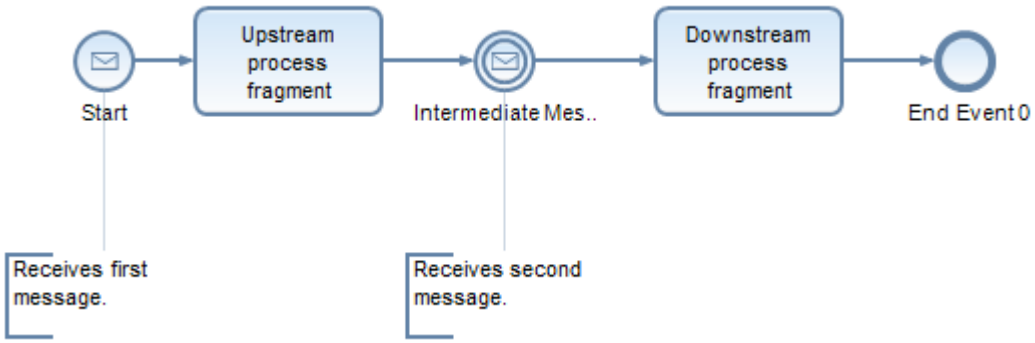
At the process step from which onward messages shall be matched, you need to enable correlation in the same way. In the example scenario, the “Send Request” activity would populate the “context” data object in its output mapping. That is, it would need to assign a value to its “key” field that corresponds to the expected “id” of incoming response messages.

Note: To inhibit messages from being matched and queued, processes should only enable correlation at the process stage from which onwards messages can actually be expected.

Message Throughput Optimization

Vice versa, you may also want to stop messages from being matched. In this case, inhibiting a correlation condition by altering the underlying context may be beneficial from two angles: For one, it stops messages from being queued. It thus, avoids a potentially costly operation that involves database accesses and occupies additional main memory. Secondly, that measure may even significantly improve message throughput characteristics in conjunction with the “conditional start” feature.

Suppose a single process instance receives two messages in total. The first message “kicks off” the process instance (i.e., is received at the start event). The second message is received in an Intermediate Message Event. Further downstream, the process performs some costly activities which delay the process’ termination for a while:



If the Intermediate Message Event’s correlation condition holds for the whole lifetime of the process instance (and also after the Intermediate Message Event has received that second message), other incoming messages are matched. Those messages are only freed from the queue and dispatched to a new process instance once the first process instance has terminated. In effect, processes’ lifetimes are disjoint such that a new process instance only starts when the predecessor instance has terminated.

Remember that as long as there is no process instance to get a new message delivered, it is queued. If the process server frequently receives messages for a certain trigger, that queue may grow over time because message delivery is delayed for an unnecessarily long time. In the worst case, queue growth may ultimately lead to a depletion of main memory resources. Apart from that, message processing throughput suffers accordingly.

To effectively mitigate the aforementioned problem in a straightforward manner, we may, again, implicitly inhibit matching by invalidating the correlation condition downstream of the “Intermediate Message Event” (please refer to Section “Message Collect” for details of how this is done). As a result, process lifetimes form a “sliding window” where the successor instance is kicked off when the predecessor instance has passed beyond the Intermediate Message Event.

Note: If messages are received with high frequency for a certain trigger, you should tune matching processes in a way that preferably only those messages are matched which can be delivered to the current process instance. That entails inhibiting matching by invalidating the correlation condition after the last message is being delivered to the process instance.

Related Content

Workflow Pattern Coverage in NetWeaver BPM 7.11

Enjoy NetWeaver BPM - Part 3: Workflow Patterns Reloaded - NetWeaver BPM 7.20

SAP NetWeaver Composition Environment 7.2 Generally Available Now

SAP NetWeaver BPM Troubleshooting Guide

For more information, visit the Business Process Expert homepage

Copyright