DENNIS WOLF

INVESTIGATING EVENT SUBSCRIPTION MECHANISMS IN BPMN

INVESTIGATING EVENT SUBSCRIPTION MECHANISMS IN BPMN

DENNIS WOLF

< Any Subtitle? >

August 2017 – version 1

Dennis Wolf: *Investigating Event Subscription Mechanisms in BPMN*, < Any Subtitle? >, © August 2017

ABSTRACT

Business Processes have become an essential tool in organizing, documenting and executing company workflows while Event Processing can be used as a powerful tool to increase their flexibility especially in destributed scenarios. The publish-subscribe paradigm is commonly used when communicating with complex event processing platforms, nevertheless prominent process modelling notations do not specify how to handle event subscription.

At the example of BPMN 2.0, the first part of this work illustrates the need for a flexible usage of event subscription in process models and derives new requirements for process modelling notations. An assessment of the coverage of these requirements in BPMN 2.0 is presented and shortcomings are pointed out.

Based on the identified requirements, this work presents a new concept for handling event subscription in business process management solutions, predominantly built on the notion of event buffers. The concept includes an extension to the BPMN meta model, specifies the semantics and API of a new event buffering module and describes the changes necessary to the behaviour of the process engine.

For evaluation purposes, the concept has been implemented as a reusable Camunda Process Engine Plugin that interacts with the academic Complex Event Processing Platform UNICORN.

ZUSAMMENFASSUNG

Kurze Zusammenfassung des Inhaltes in deutscher Sprache…

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

2 BACKGROUND

- putting cep queries into bpmn models => heiko's thesis or other related work?

- throughout the chapters we are working with message events that are received via pub/sub from an external cep platform

- BPMN: always 2.0

3 PROBLEM STATEMENT

This section will further define the problem and derive formal requirements to event subscription mechanisms

3.1 MOTIVATING EXAMPLES

- one example for independent. The subscription does not depend on a prior precess result, the subscription can be done even before process instantiation

- one example for a process that uses an intermediate event that depends (subscription-wise) on the result of a previous step in the process.

==> If the event occurs at a certain time, the process gets delayed unnecessarily or even run into a deadlock

3.2 EVENT OCCURRENCE SCENARIOS

Given the motivating examples, I am deriving a generic set of event occurrence scenarios. Each of these scenarios can occur in the real world and process implementations need to be capable of handling them to avoid negative effects.

TIME OF EVENT OCCURRENCE The most important variable to consider is the time of event occurrence. According to the BPMN specification, it is possible to catch an event if it occurs after the event element is enabled. As shown before, it is often impossible to control occurrence time and events do occur outside of these time windows. We specify the possible event occurrence times in relation to the life cycle of a process that utilizes a BPMN Intermediate Event

ref process lifecycle

.

Figure 1 shows the life cycle steps of a process and an instance from the deployment of the process until the undeployment and uses a timeline to illustrate that an event might occur at any time during this cycle. More precisely, an event is always considered to occur before or after a life cycle step or in between two consecutive steps.

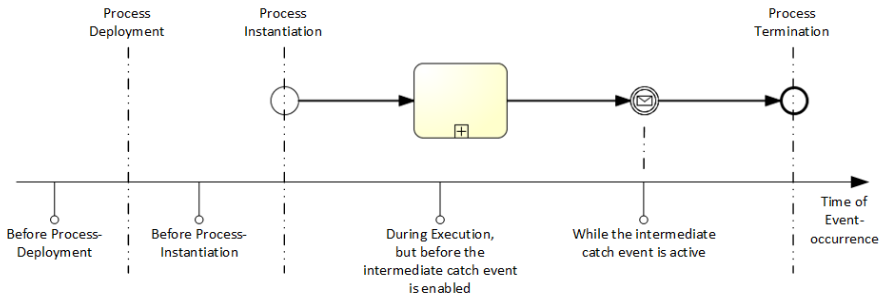
Process Deployment
Process Instantiation
Process Termination
Before Process-Deployment
Before Process-Instantiation
During Execution, but before the intermediate catch event is enabled
While the intermediate catch event is active
Time of Event-occurrence

Figure 1: Possible event occurrence times in relation to a process execution life cycle

How about system-deployment/process engine start and process undeployment? show in illustration, but say in text that we simplify this for now. after undeployment is essentially before deployment of a new process; Before Engine start is also before pr. deployment and we presume that an engine is running and does not stop.

Given the relevant life cycle steps, *process deployment*, *process instantiation* and *Event enablement*, the following occurrence scenarios are distinguished in this work:

A. O1 After the enabling of the BPMN event (BPMN default)

B. O2 The event does not occur

C. O3 Between Process instantiation and the enabling of the BPMN event

D. O4 Between Process deployment and process instantiation

E. O5 Before Process deployment

add a back reference to the examples? In example XY, events can occur before... whereas in example...

For a flexible and efficient use of events in business processes, it must be possible to use events that occur in any of these phases. To make sure that an event can be caught, no matter at which time during the phase it occurs, the subscription to the CEP platform must happen at the beginning of the occurrence phase. It follows that the event subscription must be possible at system start, at process deployment, at process instantiation, at any time during process execution and when the BPMN Event element is enabled.

EVENT SUBSCRIPTION DEPENDENCIES It is important to note that the subscription to an event source can depend on additional context information or process data. This can be a severe limitation to the possible subscription time.

ref to process model

shows a logistics process that uses event data about the GPS position of a certain truck to keep the estimated time of arrival of the transport updated. Whenever it receives an updated GPS position, the ETA is re-calculated; once the *arrival*-event has been received, the process finishes.

this example is not good, because we are not interested in a gps event that occurs earlier. Find an example where you would like earlier events, but subscription is not possible

Before the subscription to that specific truck gps event can happen, the process must determine the *truckId* to use in the event query. Only when the *truckId* is available, the subscription can be executed. This example illustrates how a query filter expression can depend on context data, but it might as well be the event source itself that differs depending on the particular execution.

there could be an xor gateway and following two different events and only one of them can get executed

solution would be to listen to all gps, but potentially too much data. Decision must be made cautiously! <= Where should I mention this? maybe later in the concept

3.3 REQUIREMENTS DEFINITION

The previous sections have exemplified how the execution semantic offered by the BPMN specification limits users in the use of events in business processes. Now these shortcomings are formalized into an additional set of requirements that must be met by a process execution environment to enable event handling in the extended set of event occurrence scenarios. The formal requirements will later be used to evaluate the capabilities of current Process Management Solutions and to develop a new concept to handling event subscription in business processes.

ref to chapters

R1: FLEXIBLE EVENT SUBSCRIPTION TIME

R1.1: Explicitness: For each event that is used in a business process, it must be possible to derive the time of event subscription from the process model. The time of subscription may either be explicitly stated or defined implicitly.

R1.2: Flexibility: The time of subscription can be influenced to catch events according to any of the event occurrence scenarios O1, O2, O3, O4. In other words, the process model defines the earliest acceptable time for an event occurrence to be considered in the process execution. The necessary options are *since system start*, *since process deployment*, *since process instantiation*, from an arbitrary but *explicit time during process execution*, or *since enabling of the Event Process Element*.

limited by subscription dependencies

R2: AUTOMATIC SUBSCRIPTION HANDLING

R2.1: Subscription The subscription to event sources is handled implicitly by the process execution environment as defined by the process model.

R2.2: Removal of Subscription The removal of a subscription from the system is handled automatically as soon as a subscription becomes unnecessary.

R3: EVENT BUFFERING

To make all events since the subscription time available during process execution, matching events need to be stored temporarily.

buffer policies and scope?

4 ASSESSMENT OF CURRENT BUSINESS PROCESS MANAGEMENT SOLUTIONS

The lack of flexibility in handling event subscription in business processes has been outlined in the previous chapters and a set of extended requirements to process management solutions have been presented. In this section I take a closer look at the capabilities of current solutions with regards to the event occurrence scenarios to get a better understanding of the issues that arise when working with event subscription in business processes. The assessment will be carried out using BPMN and Camunda, a state-of-the art and widely adopted business process engine. The main goal is to identify and illustrate the shortcomings of the current process technology stack. These shortcomings will be referenced in addition to the presented requirements to develop a more refined subscription handling model in the following chapter.

"subscription handling model"?

which functionality should be evaluated exactly?: all occurrence scenarios, but no buffer policies. The buffer will always store the last version of the event and also deliver that version.

4.1 BPMN MODELS IN PRESENCE OF THE EVENT OCCURRENCE SCENARIOS

Chapter X has revealed that processes can run into deadlocks if events do not occur at the right time

.

Figure 2 shows a generalized process that uses an Intermediate Catch Event just before process termination. In this section I first describe for each Event Occurrence Scenario how this simple event implementation behaves in presence of the given scenario. I then evaluate if it is feasible to create a BPMN model that is free from deadlock in these situations.

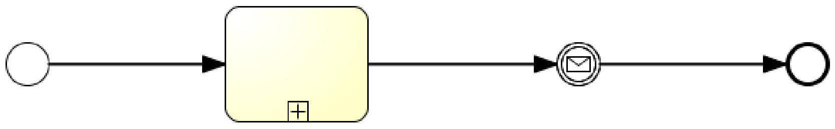

Figure 2: Abstract Process using an Intermediate Catch Event

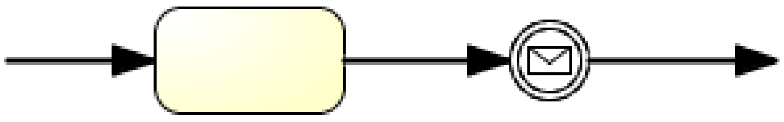

Figure 3: Standard Intermediate Catch Event

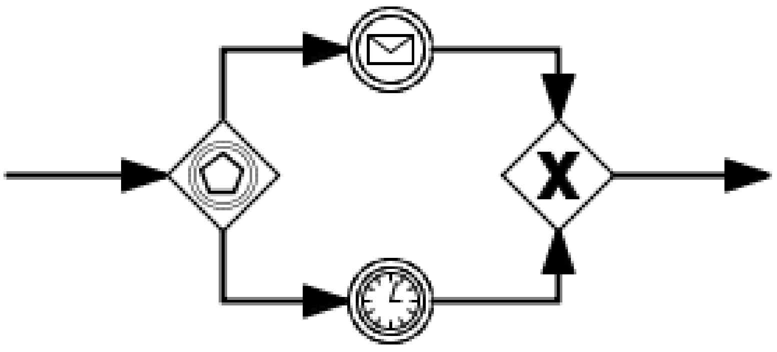

Figure 4: Intermediate Event with a parallel Timer Event

SCENARIO O1: THE EVENT OCCURS AFTER THE ENABLING OF THE BPMN EVENT The first scenario represents the most simple case, that is also natively supported by the BPMN 2.0 specification. When the event occurs after the Event element has been enabled, the event will be received and the process can proceed normally. The use of a standard Intermediate Catch Event does suffice to cover this situation.

SCENARIO O2: THE EVENT DOES NOT OCCUR In certain situations an event might not occur at all. Given a basic event implementation like in Figure 2, the process flow will get to a halt once it reaches the Intermediate Catch Event and will not be able to proceed. While, depending on the process design, this might be the desired behavior, in many situations this is not acceptable.

Let's consider a process that is supposed to wait for approval for a certain amount of time and trigger an additional request if the approval has not been issued before the deadline. Figure 4 shows how this behavior can be implemented using an Event-based Gateway which puts a Timer Event in parallel to the Intermediate Catch Event. This extension will make sure that a process does not run into a deadlock state if the expected event does not occur.

I mention an example, but that example is not exactly illustrated in the process

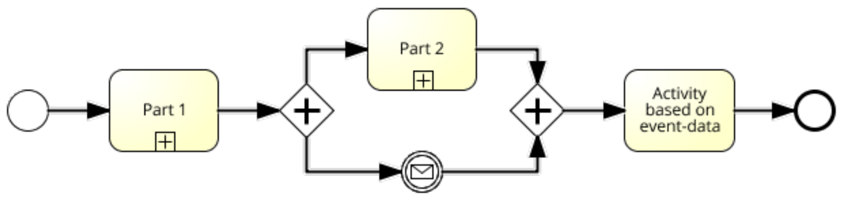

Figure 5: Event Element in parallel process flow

according to the spec: what exactly will happen to the active catch event once the timer fires?

SCENARIO O3: OCCURRENCE BETWEEN PROCESS INSTANTIATION AND THE ENABLING OF THE BPMN EVENT In case the event occurs during process execution, but before the BPMN event element is enabled and thus listening for events, the occurrence will not be considered in the execution. The process will get stuck at the Event Process Element as if the event did not happen at all. To avoid a deadlock in this scenario, a solution is to execute the Intermediate Catch Event in parallel to the rest of the process flow using a Parallel Gateway. This is illustrated in Figure 5. The time of subscription to the event can be controlled by the position of the parallel split: To implement an event subscription right after process instantiation, the Parallel Gateway has to be the first element after the Start Event (that means *Part 1* in the illustration is empty). To implement event subscription at a specific point during process execution, part of the process can execute before reaching the Parallel Gateway. In Figure 5, the event may occur at any time during the execution of the collapsed sub-process *Part 2*.

SCENARIOS O4 AND O5: BEFORE PROCESS INSTANTIATION Any Events that happen before process instantiation will not be considered in a standard Intermediate Catch Event. That applies to both scenarios, the occurrence between deployment and instantiation (*O4*) and an occurrence time before the deployment of the process in the Process Engine (*O5*).

To create a Process Model that allows to catch an event before the process instance exists, three new elements are introduced: (1) An additional *Auxiliary Buffering Process* that can catch an incoming event, (2) an *Event Buffer*, a temporary data-store that keeps event data until required by the *Original Process*, (3) an *Auxiliary Event Delivery Process*, that retrieves events from the buffer and makes them available to the *Original Process*. Figure 6 reveals the interaction of the *Original Process*, the two auxiliary processes and the data-store. To start listening for an event, the *Auxiliary Buffering Process* has to be instantiated through

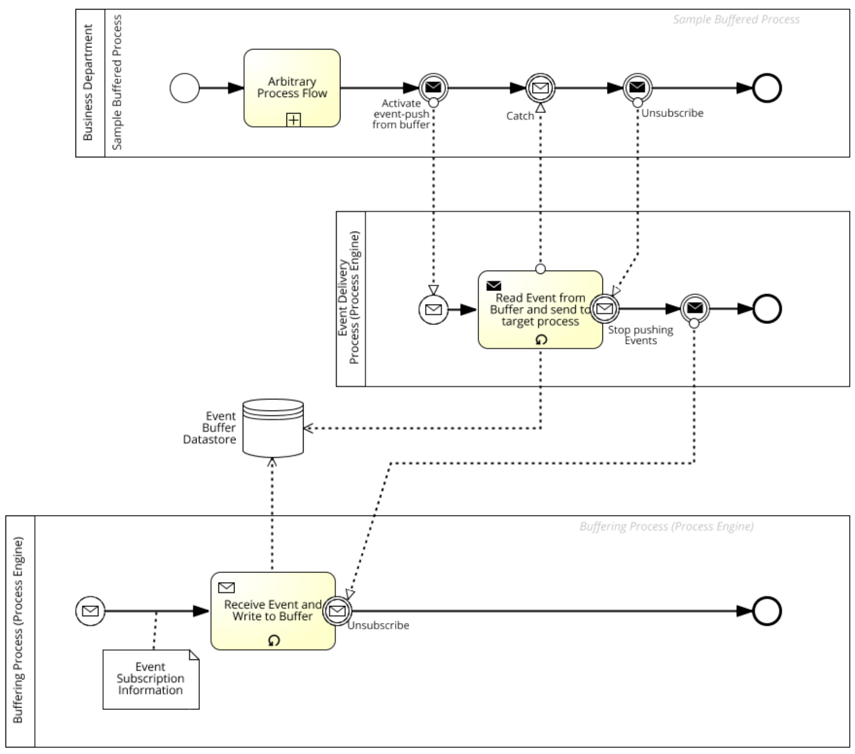
Business Department
Sample Buffered Process
Sample Buffered Process
Arbitrary Process Flow
Activate event-push from buffer
Catch
Unsubscribe
Event Delivery Process (Process Engine)
Read Event from Buffer and send to target process
Stop pushing Events
Event Buffer Datastore
Buffering Process (Process Engine)
Buffering Process (Process Engine)
Receive Event and Write to Buffer
Unsubscribe
Event Subscription Information

Figure 6: Event Buffering through an auxiliary Buffering Process

the a message start event containing the information necessary for the event subscription. The process starts listening for the event and writes the received event to the temporary data-store. The given process design is able to handle multiple event occurrences, because the receiving activity is looping. The buffering process terminates once the *Unsubscribe* event is received.

how should the ABP be started? manually?

The *Original Process* can be started any time after the buffering process. In Figure 6, the Intermediate Catch Event has been explicitly split into three events: An initial Send Event to request events, a Catch Event to receive and a final Send Event to signal that no events shall be received anymore. The initial Send Event instantiates the *Auxiliary Event Delivery Process*, which tries to read from the Event Buffer and deliver the event to the Original Process. Once there is data available in the buffer, it is sent by the sending activity. The central looping activity will retry reading from the buffer until data becomes available and will only be terminated once the *Stop*-event occurs. The Original Process can receive the event using a standard Intermediate Catch Event even when the event occurs before the instantiation of *Original Process*, so it handles scenario O4. Moreover the *Auxiliary Buffering Process* is not bound to a specific event, it works generically with any event information that is passed on to it. For that reason it is also not bound to a specific process deployment and can buffer events even before a process has been deployed, so it handles scenario *O5*. Given that the buffering process can alternatively be started using an explicit Message Send Event during process execution and the process does not stop listening until the Original Process has received the event, scenarios *O1* and *O3* are also supported.

what exactly is passed around | how many instances of each process | overwrite or append to buffer | this is only one solution to do this, one that requires minimal changes in the original process

4.2 IMPLEMENTION OF EARLY EVENT SUBSCRIPTION USING STANDARD CAMUNDA

The previous chapter has shown that that it is possible to create BPMN models to match each of the Event Occurrence Scenarios, though for the scenarios *O4* and *O5* the solution becomes increasingly complex. In the next step I investigate the capabilities of Camunda, a modern and actively developed Business Process Engine that is available under an open source license. Camunda shall be used without any code customization, that means as offered on the website. The solution presented for the last two scenarios has proven capable enough to handle all Event Occurrence Scenarios, therefor the goal is to implement this solution. It will be necessary to create the two auxiliary processes and a data-store in addition to the original process that makes use of the event buffering.

Two generic sample processes have been modeled for demonstration purposes. Figure 7 shows a simple process with an explicit subscription activity to represent the listening to the event after process instantiation but before reaching the Catch Event (Scenario *O3*). It follows a sample activity that takes 15 seconds (implemented using a *Script Task*), the Intermediate Catch Event and another Script Task that displays the content of the received message. The example for scenarios *O4* and *O5* (see Figure 8) comprises the following elements: After the start event follows an Intermediate Catch Event, then an activity that prints the message of the event to console and last the Process End Event. Both figures show the processes as modeled in the Camunda Modeler.

AUXILIARY BUFFERING PROCESS The task of this process is to subscribe to a CEP Platform using a provided event query and start listening for events. Any incoming event must be stored in a data-store (*Event Buffer*). UNICORN, an Esper-based academic event processing platform, will be used in this example. A local MySQL database has been chosen for buffering the event data because it's freely available, quick to set up, offers standardized access via SQL queries and Java

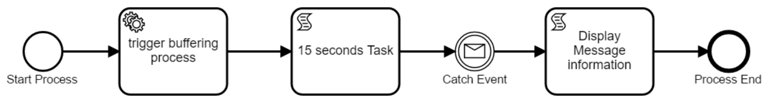

Figure 7: Generic Example Process in Camunda for Occurrence Scenario *O3*

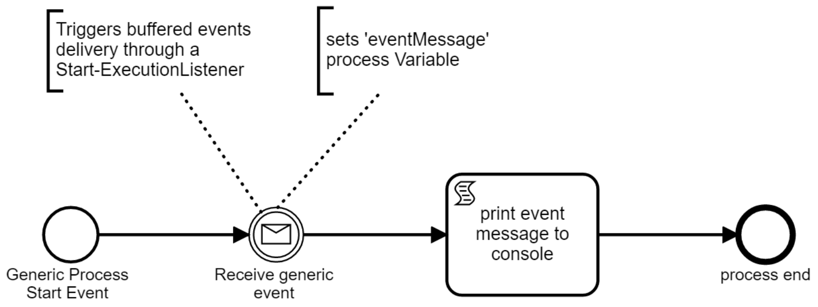
Triggers buffered events delivery through a Start-ExecutionListener
sets 'eventMessage' process Variable
Generic Process Start Event
Receive generic event
print event message to console
process end

Figure 8: Generic Example Process in Camunda for Occurrence Scenarios *O4* and *O5*

connectors and will persist data to the local harddrive by default. As UNICORN also requires an SQL database, the MySQL instance can be used in both cases.

Figure 9 shows the final Buffering Process modeled in the Camunda Process Modeler. The process can be instantiated by issuing a *Buffering Task* message. This message must contain three data fields: *processDefinitionId*, to know which process definition the buffered messages belong to; *messageName*, the name of the message event within the process; *query*, the event query in the Esper Query Language. Camunda will make the message data automatically available in the process instance as process variables, so they can be used during the execution of the Buffering Process. After instantiation, the process reaches the activity *Subscribe to Event Source*, a *Java Service Task* that executes a HTTP call to the UNICORN platform. That call registers the event query in UNICORN.

it tells unicorn its own instanceId, so that unicorn can correlate events to that exact instance

Afterwards, the process reaches the receiving activity *Wait for unsubscribe event* that will terminate the process as soon as the *Unsubscribe* event has been received. As long as this activity is active, events can be received through the attached Non-Interrupting Boundary Event. Incoming events have a field *eventBody*, which contains the

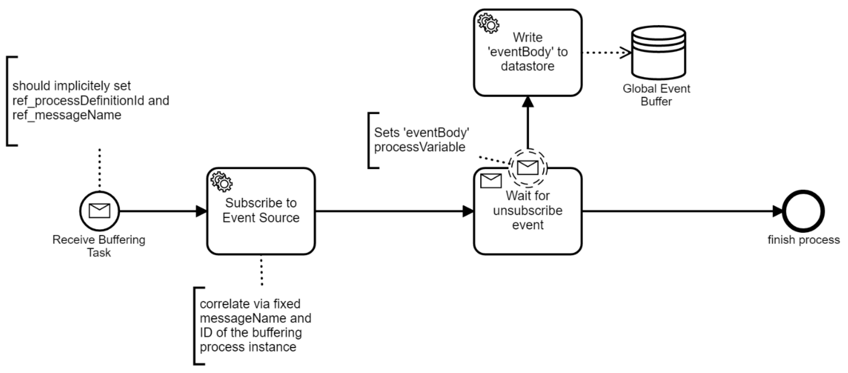
should implicitely set ref_processDefinitionId and ref_messageName
Receive Buffering Task
Subscribe to Event Source
correlate via fixed messageName and ID of the buffering process instance
Sets 'eventBody' processVariable
Write 'eventBody' to datastore
Global Event Buffer
Wait for unsubscribe event
finish process

Figure 9: Auxiliary Buffering Process in the Camunda Modeler

event information and becomes available through a process variable with the same name.

does the message from UC have a field 'eventBody'?

The boundary event triggers the service task *Write eventBody to datastore*, which takes the data from the process variable and writes it to the MySQL Database Instance (*Global Event Buffer*).

AUXILIARY EVENT DELIVERY PROCESS The delivery process (see Figure 10) reads the latest data from the buffer and sends it to the process instance. It can be started with a message that contains the *processInstanceId* and the *processDefinitionId* of the requesting process and the *messageName* of the Message Event that is requested from the buffer. A *Delay* Timer Event has been inserted to make sure that the receiving process is already in listening state, the execution happens asynchronously. It follows the service task *Retrieve event from buffer*, which executes Java code to read from the MySQL Database *Event Buffer* and store the event information in a process variable named *eventMessage*. The content of that process variable is sent to the *Original Process* in the Send Event, afterwards the execution is finished.

INTERACTION OF THE PROCESSES In this implementation of flexible event subscription, the action of subscribing to the event source and the reception of events in the Original Process are splitted into two separate parts, each supported by an auxiliary process. To initiate the subscription at the event source, the Auxiliary Event Buffering process has to be started. For scenario *O3*, this happens through an extra activity (*Trigger Buffering Process*) during process execution, so that events after process instantiation are received by the Buffering Process. In scenarios *O4* and *O5*, the subscription and thus the instantiation of the Buffering Process must happen before the instantiation of the Target Process. As there is no such mechanism in the standard Camunda Process Engine, the Buffering Process must be started by

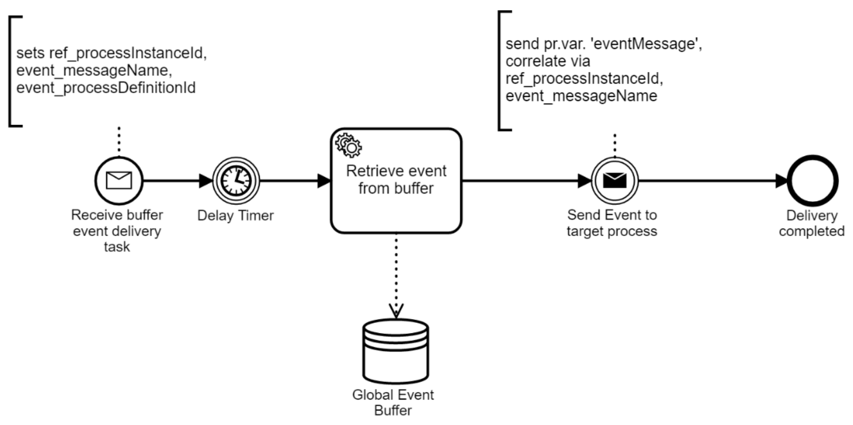
sets ref_processInstanceId,
event_messageName,
event_processDefinitionId
send pr.var. 'eventMessage',
correlate via
ref_processInstanceId,
event_messageName
Receive buffer event delivery task
Delay Timer
Retrieve event from buffer
Send Event to target process
Delivery completed
Global Event Buffer

Figure 10: Auxiliary Event Delivery Process in Camunda Modeler

hand, providing the *processDefinitionId*, the *messageName* and the *eventQuery*.

Now that the *Buffering Process* is running, any events matching the query will be stored to the buffer. When the Target Process reaches the Catch Event, a request for buffered events is sent as a message to trigger the *Auxiliary Event Delivery Process*. This message is sent using a short piece of Java code that gets executed when the Catch Event is reached. The code is invoked by a Start ExecutionListener attached to the Catch Event. ExecutionListeners are offered by Camunda to execute own Java programs before or after relevant events during process execution, like the execution of an element in the process. While the Original Process will now start listening for the desired events, the Event Delivery Process will send the buffered events as messages to the Original Process.

If no events have been received yet, all the involved processes remain active: the Buffering Process will keep listening for an external event. The Delivery Process will send an event to the Original Process as soon as there is one in the buffer. The Catch Event in the Original Process will keep listening for an Event.

the termination of the processes is not yet implemented in Camunda

get wording straight: Original process, target process, requesting process, main process | also always italic or never

Fehlende Abbildung

overview of the 3 processes, mysql and UNICORN note that this is an investigative implementation that matches exactly the given use-case and is not meant to be used in production. It is neither flexible nor robust enough for that purpose, but suits very well in understanding the capabilities and the shortcomings of BPMN and Camunda when it comes to handling the Event Occurrence Scenarios

4.3 DISCUSSION

Thanks to the flexibility of BPMN 2.0 and Camunda, it has proven possible to implement a flexible event subscription time using these tools. It was necessary to create two generic auxiliary processes for event buffering, to connect to a MySQL data-store and use ExecutionListeners to execute custom Java code in Camunda to cover all scenarios *O1* to *O5*.

give them short names for reference?

NO AUTOMATIC SUBSCRIPTION HANDLING
subscription has to be modelled explicitly or issued manually. event request has to be modelled explicitly

NO OPTION TO MODEL SUBSCRIPTION INFORMATION in bpmn

MORE COMPLEX PROCESS MODELS
Process models using flexible subscription time become overly complicated

BUFFERING IS AN IT TASK
Event Buffering is an IT Task, not a Business Task

ADDITIONAL LOAD ON THE PROCESS ENGINE
Buffering puts extra load on the process engine

HIDDEN PERFORMANCE LIMITATIONS OF THE PROCESS ENGINE
The Process Engine might pose performance limitations that we cannot influence

5 FLEXIBLE EVENT SUBSCRIPTION

Present an abstract framework for flexible event subscription. > Including: Model <> Process Engine <> Buffer <> CEP > How does event subscription currently affect the workflow? > What should a workflow look like that allows early event subscription? > What must be explicitly stated by the user? What should be done automatically in the background? (1 page)

5.1 BPMN EXTENSION

To fulfill requirements R1.1 and R1.2, additional information has to be included in the BPMN model. By default, a BPMN intermediate event does not have information on the time of subscription or the event query. The BPMN specification offers BPMN-X extensions to add custom properties or elements to a model.

To accomodate the required information, the following extension is proposed: > The extension should apply to MessageIntermediateCatchEvent and MessageBoundaryEvent > extend tMessage => tBufferedCEPMessage, so that the messageRef can be reused > OR extension to messageEventDefinition: ExplicitSubscriptionMessageEventDefinition > [subscriptionQuery, subscriptionTime, bufferPolicy]

A buffer shared across multiple instances or events is more complex than a simple single-event-buffer (that one does not require buffer policies). As soon as the requestEvent call can be executed multiple times for the same queryId, we need to specify the following aspects:

Buffer policies: (widely based on [Ref paper Sankalita]) RetrievalPolicy, ConsumptionPolicy, LifetimePolicy + buffer maximum age (= combination of lifetime policies)

5.2 BUFFERED EVENT HANDLING

Why do we need a buffer to allow early event subscription?

What is the desired functionality of the event buffer? What functionality (API) does it expose? > this could be seen as an extension to the API that is exposed by a standard CEP Platform > standard platform API: registerQuery(queryString, notificationRecipient) : queryId, deleteQuery(queryId) > extended API: registerQuery(queryString): queryId, requestEvent(queryId, notificationRecipient), unsubscribe(queryId), deleteQuery(queryId)

5.3 EXTENDED PROCESS ENGINE BEHAVIOUR

=> As a link between the BPMN model and the Buffered Event handling

there must be a "subscription-garbage-collection" for any events that cannot be reached anymore in the current process execution! e.g. two different events behind an xor-gateway. the garbage collection could be executed on every transition

DECLARATION

Put your declaration here.

Potsdam, August 2017

Dennis Wolf

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/