Distributed Orchestration with Camunda BPM, Part 2

Simon Zambrovski
Nov 4 · 7 min read ★

I strongly believe that orchestration of Microservices will become the next big thing to solve. At the time of writing, several solutions try to compete in this area, mostly building their own (textual) domain-specific languages to describe the orchestration. In my opinion orchestration should be expressed in BPMN 2.x instead, since it is a well-adopted, understandable and mature language designed exactly for this purpose.

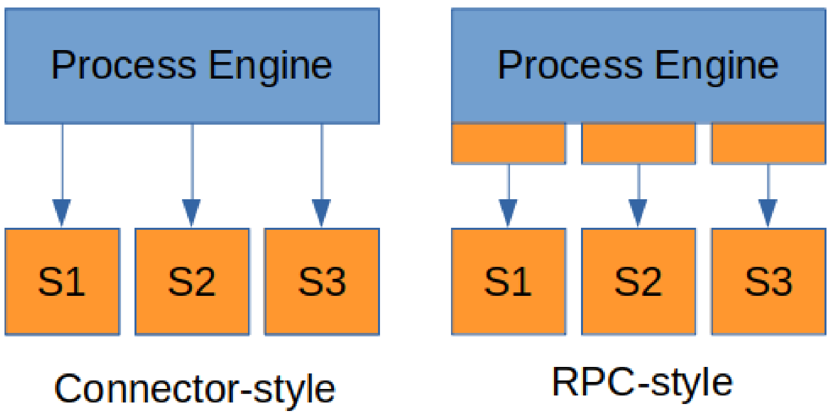
. . .

The term Orchestration in Microservice context might be ambiguous. To get it clearer, I would like to propose the following classification:

SOA-like orchestration

SOA focuses on remote communication between services, built around business capabilities. Central process engine synchronously calls distributed services remotely. The integration is performed between the state-handling process engine and the state-less service.

> *I'm over-simplifying it a little here and describing a "bad-design/misunderstood-SOA", since in essence SOA was NOT about stateless services, but was sometimes implemented this way.*



Synchronous orchestration

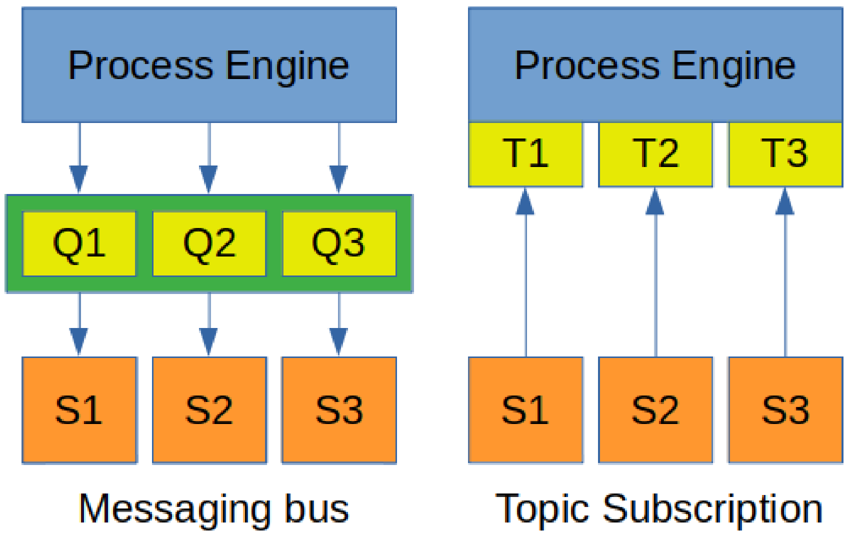
There are two different implementation styles of this class of systems.

1. The **Connector** integration pattern is used, if the process engine is calling the service (S1, S2, S3) using the selected protocol directly (usually HTTP).
2. The **RPC** integration pattern is used, if the engine calls a local delegate and these are invoking a remote service (S1, S2, S3) via selected protocol (HTTP, Java RMI or any other synchronous protocol).

In both cases, the integration requires the engine and the services to be online simultaneously. The engine might know the location of the services or use a registry or a broker (remember the Webservice triangle) to resolve this and the services use invocation-oriented implementation to execute work on behalf of the process engine.

Message-driven orchestration

Instead of synchronous invocation, the central engine might send messages to queues or topics and the stateless services subscribe to those. The simultaneous availability of the engine and the services is not required. As a result the services use a subscription-oriented implementation to execute work on behalf of the process engine.



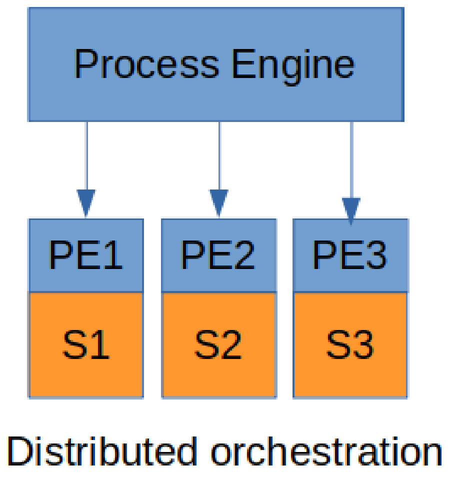
Asynchronous orchestration

There are two types of implementation depending on the messaging abstraction in use:

1. The messaging infrastructure might be middleware (for example using a central messaging bus) offering the concept of **queues** (Q1, Q2, Q3). The engine send asynchronous messages to services (S1, S2, S3) using queues.
2. Instead of using queues, the process engine may publish the information to pre-defined **topics** (T1, T2, T3). The topics subscription may be a part of the process engine (aka External Task Pattern as displayed above) or be on the centralized messaging middleware.

Distributed orchestration

The orchestration itself is distributed. Instead of separation between state-full engine and stateless services, the services become state-full (and get their own means of handling state e.g. using orchestration) and the integration takes place between business processes (e.g. running in process engines PE1, PE2, PE3).



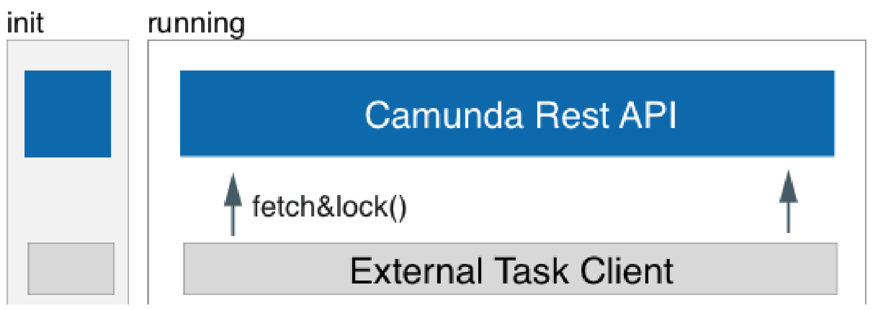
Distributed orchestration between process engines

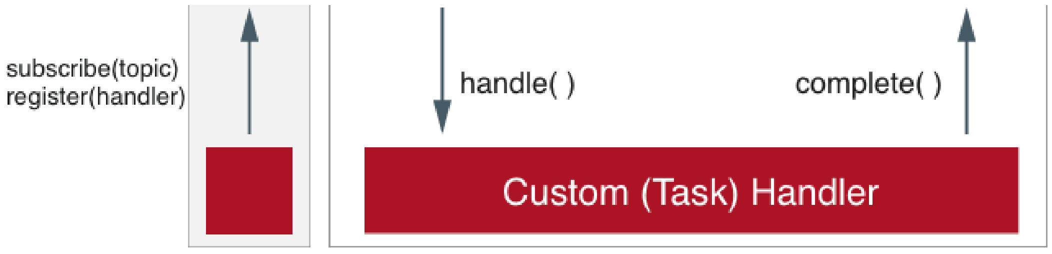
This style of orchestration has been introduced in the last article (see Part 1 of this series), in which I shared my thoughts about the decomposition patterns of orchestration. In this part, I focus on more patterns and implementation strategies using the External Task Pattern.

. . .

External Task Pattern

External Task Pattern has been introduced by Camunda BPM in version 7.4 and is one of the most important features to break with workflow monolith towards distributed workflow. Originally, it is intended to provide a subscription-oriented service task implementation in contrast to the invocation-oriented. That is, if the engine executes a service task, it is not calling a delegate to call a (remote) service, but creates an external task record and waits for a (remote) external task worker to fetch and execute it.





Camunda External Task

The external task pattern has several important properties:

- External task is *always a wait-state* in the engine. The process engine will commit the transaction after creating an external task record.
- It *reverts* the direction of communication. The process engine is not calling the service, but the external task worker (co-located with the service) is calling the process engine.
- Custom Handler gets notified, if the external task is available and can use the `ExternalTaskService` API (`complete()` , `handleFailure()` and `handleBPMNError()`) to inform process engine about the outcome.
- The meaning of timeout of `fetchAndLock` is to reserve the execution for a task worker for certain time. If the timeout occurs before the response is transmitted, the task get available for other task handlers again.

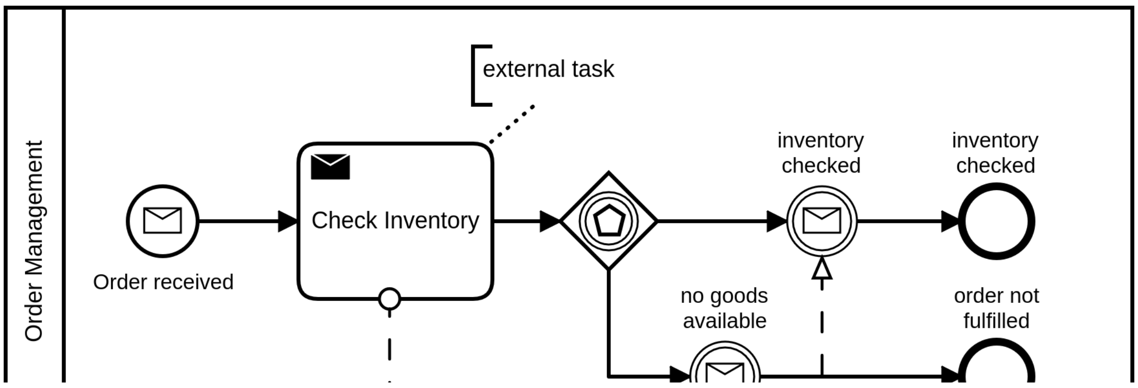
An external task client is required in order to perform work on specific topics. Camunda provides own implementations in Java and JavaScript, which can be used as a library.

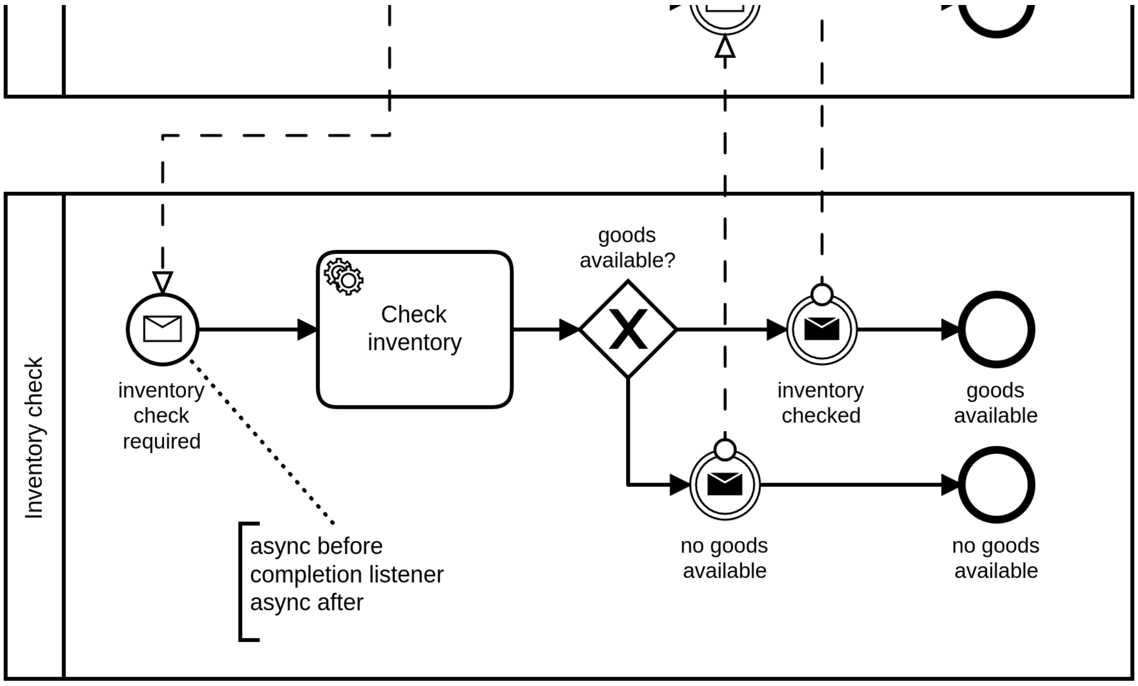
. . .

Using External Tasks in Distributed Orchestration

An interesting approach is to use External Task Pattern in distributed orchestration. Remember the example from Part 1 of this series? Let us try to model it using external task pattern.

Using External Task Pattern, I propose the following implementation:





Let me explain, how this implementation can work. There are some requirements to the `Inventory check` process in order to work.

1. The Order Management's `Check Inventory` send task has an `External Task` implementation.
2. The Order Management has NO wait states between external task and the message catch events.
3. The Inventory Check component provides an External Task worker, connecting to Order Management process engine and starting the Inventory process.
4. The start of Inventory process is IDEMPOTENT. That is, if the process is already running, the external task worker should not start a new one but just do nothing.
5. The external task worker only fetches the external task from Order Management, but is NOT completing it.
6. The complete of the external task is performed by a Completion listener, attached as an Execution Listener at the end of the message start event `Inventory check required`. This completion is synchronous. Since the external task id is required for completion, it may be stored in a process variable of the Inventory check process.
7. The start of the `Inventory check` process MUST be marked as `async before`. If the completion of the external task fails, the process remains running, so the job executor of the `Inventory check` process will retry the completion.
8. The start of `Inventory check` process SHOULD be marked as `async after`. If any activity after the start fails, the transaction will not be rolled back before the Completion listener (because this already has completed the external task).

Let me explain why the external task should be completed by the completion listener, instead of external task worker and how this solves the concurrency problem. The completion of the external task by the listener is a synchronous call to the `Order Management` process engine, executed e.g. via REST. This call starts a new transaction in the `Order Management` process engine, which completes the external task and writes the message subscriptions for messages `inventory checked` and `no goods available` BEFORE the response is sent back to the `Inventory check` process engine. Because of this fact, there is no race condition between two engines and `Order Management` is waiting for the message before it can be sent by the `Inventory check` process engine.

In order to implement this behavior, we need a modified External Task Client, capable to separately fetch and complete an external task. The External Task Client library provided by Camunda is not capable of doing this and, a small modification of existing client (and client builder) is required. Here is the code:

```
package io.holunda.spike.external.task.client;

import org.camunda.bpm.client.ExternalTaskClient;
import org.camunda.bpm.client.impl.ExternalTaskClientBuilderImpl;
import org.camunda.bpm.client.task.ExternalTaskService;
import org.camunda.bpm.client.task.impl.ExternalTaskServiceImpl;

/**
 * Overriding builder to be able to setup client-related stuff.
 */
public class MyExternalTaskClientBuilder extends ExternalTaskClientBuilderImpl {

  /**
   * Creates working instance.
   */
  public static MyExternalTaskClientBuilder create() {
    return new MyExternalTaskClientBuilder();
  }

  /**
   * Retrieves the external task wrapper, containing the external task client and the external t
   */
  public ExternalTaskClientWrapper buildClientWrapper() {
    /*
     * This creates the external client and initializes the engine client
     */
    final ExternalTaskClient client = super.build();
    /*
     * It is ok to create our own external task service instance, since it is stateless
     * and only provides methods to operate on tasks, which delegate to the engine client.
     */
    final ExternalTaskService service = new ExternalTaskServiceImpl(engineClient);
    /*
     * Return wrapped instance.
     */
    return new ExternalTaskClientWrapper(client, service);
  }
}
```

MyExternalTaskClientBuilder.java hosted with ♡ by GitHub view raw

Extending builder to supply the client and the service

Using the builder above it is possible to create a pair consisting of External Task Client and the client-side External Task Service (the returned `ExternalTaskClientWrapper` is just a POJO holding two references).

For the instantiation of the builder, the following code can be used:

```java
package io.holunda.spike.external.task.client;

import lombok.val;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * Configuration.
 */
@Configuration
public class MyConfiguration {

  /**
   * Creates the task client wrapper with configured client.
   * @return task client wrapper.
   */
  @Bean
  public ExternalTaskClientWrapper externalTaskClient() {

    MyExternalTaskClientBuilder builder = MyExternalTaskClientBuilder.create();

    builder
      .workerId("my-worker-id")
      .baseUrl("http://localhost:8080/rest/")
      .asyncResponseTimeout(80_000);

    return builder.buildClientWrapper();
  }

}
```

MyConfiguration.java hosted with ♡ by GitHub view raw

Instantiation of the extended builder

Now imagine the External Task Worker, responsible for starting the Inventory Check process. It consists of an External Task Handler and the Completion listener (both encapsulated by the same Spring Component):

```java
package io.holunda.spike.external.task.client;

import org.camunda.bpm.client.task.impl.ExternalTaskImpl;
import org.camunda.bpm.engine.RuntimeService;
import org.camunda.bpm.engine.delegate.ExecutionListener;
import org.camunda.bpm.engine.runtime.ProcessInstance;
import org.camunda.bpm.engine.variable.VariableMap;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.util.List;

import static org.camunda.bpm.engine.variable.Variables.putValue;

@Component
public class InventoryCheckWorker {

  private static final String EXTERNAL_TASK_ID = "EXTERNAL_TASK_ID";
  private static final String MESSAGE = "inventory_check_needed";

  private final ExternalTaskClientWrapper wrapper;
  private final RuntimeService runtimeService;

  public InventoryCheckWorker(ExternalTaskClientWrapper wrapper, RuntimeService runtimeService)
    this.wrapper = wrapper;
    this.runtimeService = runtimeService;
  }

  /**
   * Setup worker
   */
  @Scheduled(initialDelay = 60_000, fixedDelay = Long.MAX_VALUE)
  public void registerWorker() {
    wrapper
      .getExternalTaskClient()
      .subscribe("topic")
      .lockDuration(60_000)
      .handler(
        (externalTask, externalTaskService) -> {
          String businessKey = externalTask.getBusinessKey();
          // check if already started
          List<ProcessInstance> running = runtimeService
            .createProcessInstanceQuery()
            .processInstanceBusinessKey(businessKey)
            .list();
          // be idempotent, start only if not already started
          if (running.isEmpty()) {
            VariableMap variables = putValue(EXTERNAL_TASK_ID, externalTask.getId());
            runtimeService
              .startProcessInstanceByMessage(
                MESSAGE,
                businessKey,
                variables
              );
          }
        }
      )
      .open();
  }

```

```
    /**
     * Completion listener.
     *
     * @return an execution listener which completes the external task that started the process.
     */
    public ExecutionListener acknowledgeStarted() {
      return execution -> {

        String externalTaskId = (String) execution.getVariable(EXTERNAL_TASK_ID);
        ExternalTaskImpl task = new ExternalTaskImpl();
        task.setId(externalTaskId);

        this.wrapper.getExternalTaskService().complete(task);
      };
    }
  }
```

Please note, that the handler is only fetching the task, starts the process if it is not already running and stores the external task id in a process variable (but is NOT completing it). The completion listener is responsible for reading the external task id from the variable and completing the external task.

· · ·

Conclusion

This article proposes a classification of orchestration based on the communication pattern properties. I focus on the distributed orchestration, as already discussed in Part 1 of this series. Since this pattern is not easy to implement if you are trying naive approach, I propose to use the External Task Pattern and slightly modified Camunda External Task Client (for Java). In doing so, we get the best from both worlds:

- The BPMN model expresses exactly what is intended, remains clean and is not polluted with any framework / implementation workarounds.
- The simultaneous availability of components is not required (if this is a real issue, there is also a possibility to replace the delivery of the message from the `Inventory check` process by another External Task Worker at the costs of the more polluted process model)
- The communication direction is reverted avoiding construction of the distributed monolith depending on multiple services.

Microservices Orchestration Camunda Bpm External Task Pattern