# SSE Project Report: CityDistance Service

Leonardo Gazzarri
September 2, 2016

## 1 Introduction

CityDistance is a SOAP web service that offers as output the distance "as the crow flies" between two cities in the world, offering to the user the opportunity to choose the desired unit of length.

CityDistance offers only one operation, that is `GetCityDistance`, and orchestrates three existing remote web services in order to provide its service:

- **Length/Distance Unit Convertor**, a RESTful and SOAP service that performs length/distance conversion calculation for metric and imperial units including Kilometers, Meters, Miles, Feet and Inches. In the following this service will be called simply "**LengthUnit**" (abbrev. LU). Service description: http://www.webservicex.net/length.asmx?WSDL
- **Nominatim**, a RESTful service that is able to derive the geographic coordinates of a named place on earth and much more. In the following this service will be called simply "**Geocode**" (abbrev. GC). API: http://nominatim.openstreetmap.org/search. DOC: http://wiki.openstreetmap.org/wiki/Nominatim
- **DesertHail Geodata Service**, a RESTful and SOAP service that offers the operation `GetLineDistance` to retrieve the distance (in kilometers) between two coordinates. In the following this service will be called simply "**Distance**". Service description: http://services.deserthail.com/GeodataService/ServiceMaster.asmx?WSDL

Figure 1.1 shows the orchestrator inputs, how the inputs and the outputs of the remote services are related, and the outputs. In the figure, the names in the black brace next to a

box represent the inputs for that box, the elements in the red brace next to an arc represent the output of the above invocation service; blue text is used to show some processing actions or to describe if a specific input for a remote WS is set to an "hardcoded" constant value in the orchestrator.

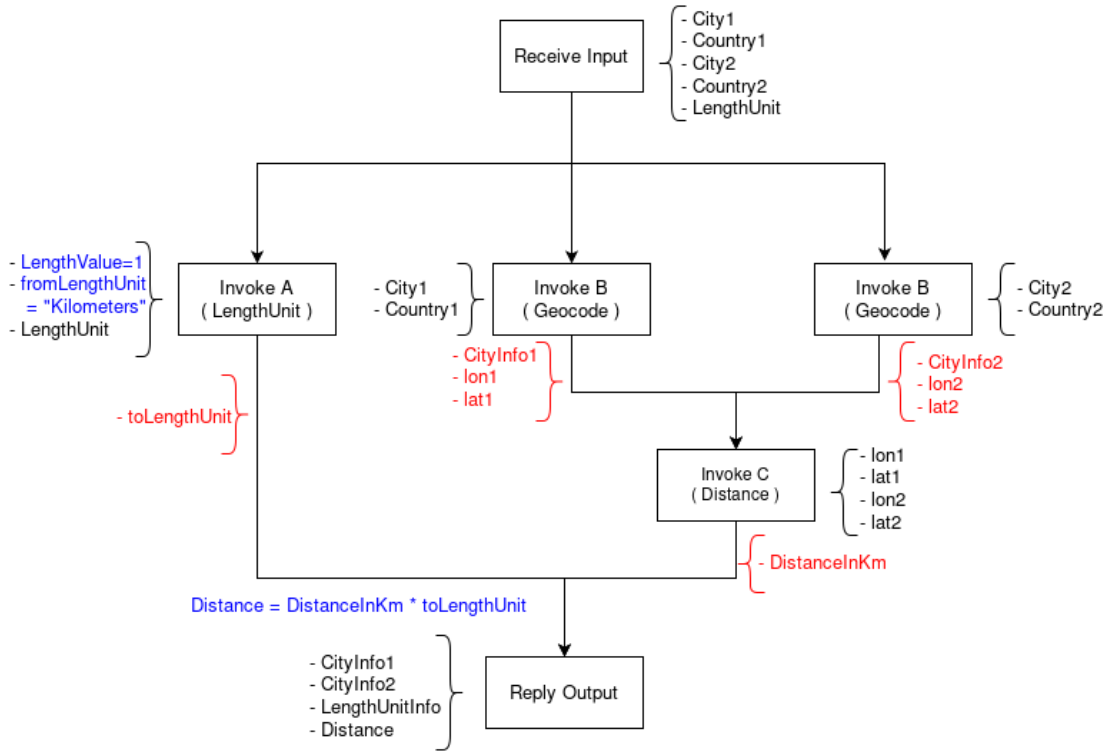


Figure 1.1: Abstract view of the relations in CityDistance's orchestration

Let's describe briefly the input information that the consumer has to provide to the orchestrator's service:

- **City1**: name of the first city *(e.g. Pisa)*
- **Country1**: name of the country of the first city *(e.g. Italy)*
- **City2**: name of the second city *(e.g. Rome)*
- **Country2**: name of the country of the second city *(e.g. Italy)*
- **LengthUnit**: name of the unit of length *(e.g. Kilometers, Meters, Miles, Feet, ...)*. A LengthUnit is not admissibile when it does not exist or if it is smaller than a foot.

... and the replied output information:

- **CityInfo1**: information of the first city *(e.g. Pisa, PI, TOS, Italia)*. If the name of the City1 doesn't exist, then no information is provided

- **CityInfo2**: information of the second city *(e.g. Roma, RM, LAZ, Italia)*. If the name of the City2 doesn't exist, then no information is provided
- **LengthUnitInfo**: the unit length used to express the Distance value. If the LengthUnit is admissible, LengthUnitInfo is equal to LengthUnit, otherwise LengthUnitInfo is set to the default "Kilometers" *(e.g. Kilometers)*.
- **Distance**: the "as the crow flies" distance expressed using the unit of length specified in LengthUnitInfo *e.g. 264.32563984660715*

# 2 WS-BPEL Implementation

The orchestration described in the previous section has been implemented in WS-BPEL and made up by the following two processes:

- **AsyncLength**, an asynchronous process that acts as a proxy for the remote **LengthUnit** service
- **CityDistance**, the real orchestrator process

These two processes will be examinated in the following subsection.

## 2.1 WS-BPEL Processes

Figure 2.1 shows the whole proxy **AsyncLength**.

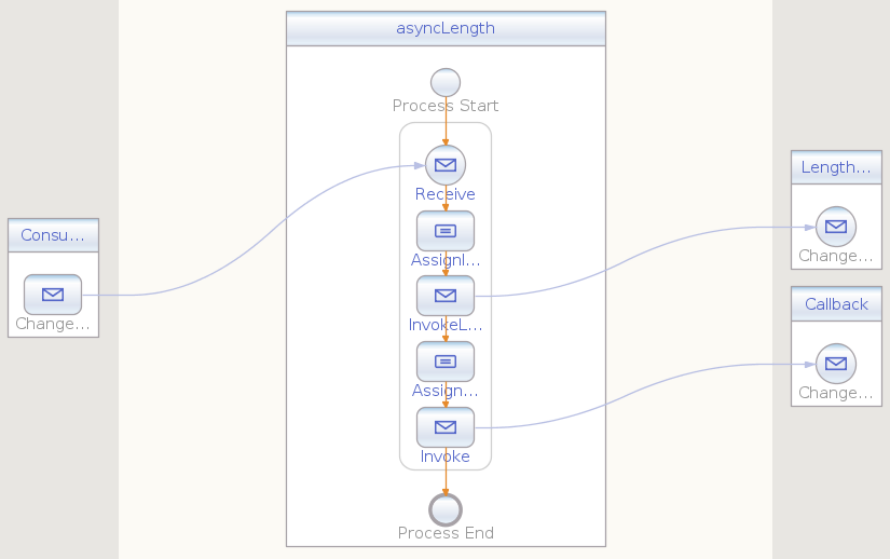


Figure 2.1: AsyncLength BPEL process.

This process works in an asynchronous way. Namely it exposes the one way operation `ChangeLengthUnit` and, after received the input from the orchestrator, it calls the

synchronous operation `ChangeLengthUnit` offered by the **LengthUnit** SOAP service. Therefore, if faults don't occur, the response of the previous operation is sent back invoking a callback. The callback mechanism is implemented in such a way the response contains the original input plus the response of the synchronous operation. This is necessary for the correlation mechanism at the orchestrator's edge. The one way operation and the callback operation are exposed respectively in the "AsynchLength.wsdl" and "AsynchLengthCallback.wsdl" files.

The **CityDistance** orchestrator process, after received the input, starts a *<flow>* activity which creates two concurrent scoping activities that are `ScopeA` and `ScopeBC`. Figure 2.2 shows how `ScopeA` is structured. `ScopeA` packages a *<sequence>* activity which invokes the one-way operation `ChangeLengthUnit` from the asynchronous proxy process **AsyncLength** (the partner link is named LengthUnitProxy). After the one way *<invoke>* the pick activity named "WaitResponse" starts. This activity contains two branches, one for the message handler of the callback message from the proxy and another one for an *<onAlarm>* activity. If the reception of the callback message does not occur within 10 seconds, the branch associated to the *<onAlarm>* is triggered by the occurrence of the timeout and a `to_fault` is thrown. The throwing of `to_fault` causes the end of the whole process and gives back to the user an error message. If the callback message is received before the timeout, the desired conversion of the value of one kilometre is received. Then this value is tested (`CheckConditionA`) in order to check if the value is admittable (foot is the smallest unit of measure allowed). If this value is greater than 3281 (number of feet in one kilometre) then `a_reply_fault` is thrown, the scope stops its execution and the control flow goes to the fault handler associated to the `ScopeA`, which simply sets the incriminated value to one and "Kilometers" as the default unit of measure.

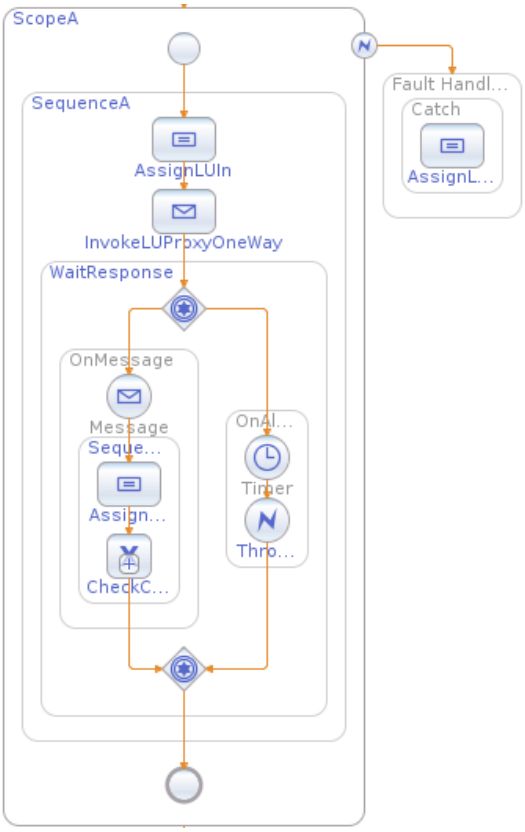


Figure 2.2: ScopeA with "CheckConditionA" collapsed

Instead, figure 2.3 shows how `ScopeBC` is structured. `ScopeBC` packages a *<sequence>* activity that at the very beginning starts a *<flow>* activity. This *<flow>* activity creates two concurrent sequence activities which invoke the **Geocode** service in parallel. The first

sequence, invokes the `GetGeocode` operation passing as parameters *City1* and *Country1* and the second sequence invokes the same operation passing as parameters *City2* and *Country2*. In reality, for the input, there is also a parameter that specifies to reply with an xml message and another parameter that specifies the limit to one of the number of returned results (one city name can match with more places).

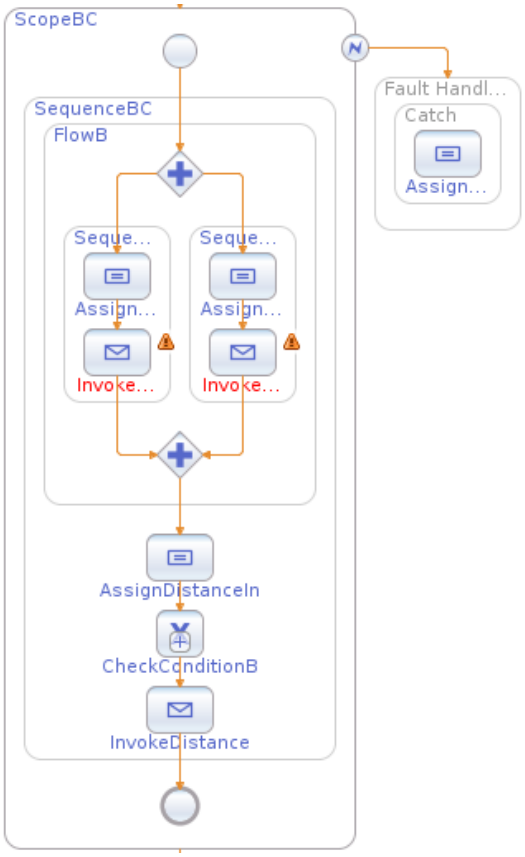


Figure 2.3: ScopeBC with "CheckConditionB" collapsed

Then the responses are parsed in order to get the respective geographic coordinates. If an error occurs during the parsing (`CheckConditionB`) then `b_reply_fault` is thrown and the control flow goes to the fault handler associated to `ScopeBC`, which sets to zero the distance value that will be returned to the consumer. Otherwise the `GetLineDistance` of **Distance** is called getting the actual distance in kilometers between the two points.
When the external <*flow*> activity ends, the process builds the reply message using the information about cities collected through the invocations of the Geocode service, the unit of measure chosen by the user (or "Kilometers" if the user choice was for a length unit too much small) and the distance obtained multiplying the result of `GetLineDistance` with the value given by the conversion of one kilometre into the desired unit of length that has been replied by the proxy.

## 2.2 Tests

In the following are presented some developed test cases. For each item on the list, there is a brief description of the test followed by the content of a SOAP request message, and after an empty line, the content of a SOAP response message.

- Example of a correct request message and the relative response. Feet is supported as the smallest unit of length for the service.

```
<cit:GetCityDistance>
<City1>Pisa</City1>
<Country1>Italy</Country1>
<City2>Rome</City2>
<Country2>Italy</Country2>
<LengthUnit>Feet</LengthUnit>
</cit:GetCityDistance>

<m:GetCityDistanceResponse xmlns:m="http://j2ee.netbeans.org/wsdl/
    CityDistance/src/CityDistance">
<CityInfo1>Pisa, PI, TOS, Italia</CityInfo1>
<CityInfo2>Roma, RM, LAZ, Italia</CityInfo2>
<LengthUnitInfo>Feet</LengthUnitInfo>
<Distance>867210.1044836192</Distance>
</m:GetCityDistanceResponse>
```

- Example of a request message that triggers `to_fault` in the process. Since "Knees" is an invalid length unit for **LengthUnit**, a fault is thrown in the Proxy process. This fault is not handled and therefore the callback operation is not invoked.

```
<cit:GetCityDistance>
<City1>Pisa</City1>
<Country1>Italy</Country1>
<City2>Rome</City2>
<Country2>Italy</Country2>
<LengthUnit>Knees</LengthUnit>
</cit:GetCityDistance>

<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Client</faultcode>
<faultstring>to_fault</faultstring>
<detail>
<FaultInfo>timer expired</FaultInfo>
</detail>
</SOAP-ENV:Fault>
```

- Example of a request message that triggers the `a_reply_fault` handling in the process. The angstrom is smaller than feet length unit, therefore the response message shows the "Distance" value in kilometers (the default unit of length).

```
<cit:GetCityDistance>
<City1>Paris</City1>
<Country1>France</Country1>
```

```
<City2>Bruxelles</City2>
<Country2>Belgium</Country2>
<LengthUnit>Angstroms</LengthUnit>
</cit:GetCityDistance>

<m:GetCityDistanceResponse xmlns:m="http://j2ee.netbeans.org/wsdl/
    CityDistance/src/CityDistance">
<CityInfo1>Paris,  le -de-France, France m tropolitaine, France</
    CityInfo1>
<CityInfo2>BXL, Bruxelles-Capitale - Brussel-Hoofdstad, R gion de
    Bruxelles-Capitale - Brussels Hoofdstedelijk Gewest, Belgi  -
    Belgique - Belgien</CityInfo2>
<LengthUnitInfo>Kilometers</LengthUnitInfo>
<Distance>263.65527147440844</Distance>
</m:GetCityDistanceResponse>
```

- Example of a request message that triggers the `b_reply_fault` handling in the process. **City2** does not exist and "Distance" shows 0 as value.

```
<cit:GetCityDistance>
<City1>Madrid</City1>
<Country1>Spain</Country1>
<City2>Xyz</City2>
<Country2>Unknown</Country2>
<LengthUnit>Miles</LengthUnit>
</cit:GetCityDistance>

<m:GetCityDistanceResponse xmlns:m="http://j2ee.netbeans.org/wsdl/
    CityDistance/src/CityDistance">
<CityInfo1>Madrid,  rea  metropolitana de Madrid y Corredor del
    Henares, Comunidad de Madrid, Espa a</CityInfo1>
<CityInfo2/>
<LengthUnitInfo>Miles</LengthUnitInfo>
<Distance>0</Distance>
</m:GetCityDistanceResponse>
```

# 3 Analysis of the WS-BPEL specification

Some comments about the proposed workflow net in figure 3.1, modelling the control flow of the orchestrator process:

1. **Orchestrator's Activities** Not all the basic BPEL's activities of the orchestrator have been represented as transition nodes. For example, all the assignments before the invocations are not explicitly represented as transition nodes in the model. This kind of activities is considered as grouped into only one. For example, a transition node modelling an assignment activity is considered grouped in each of the "Invoke" transition nodes.

2. **Pick Modelling** The pick activity has been modelled using an **implicit OR-split construct** because the choice here is determined by the actual order of triggering between the tasks that share the pick node as the input place.

3. **Flows And Conditions** Flows has been modelled by means of AND-split constructs and AND-join constructs. The conditions as explicit OR-split constructs.

4. **The Fault Handling** The most intricate part of the modelling is the one that concerns the `to_fault` handling. While in the other fault's handling cases the situation is easy to model since the fault affects only one single "flow of execution", in this case, the throwing of `to_fault` has to stop also the "flows of execution" that regards to the services B and C. This is necessary to provide the soundness of the net. In order to provide the correct behavior three places has been added to the net, namely $tS$ ("to be stopped"), $s$ ("stopped") and $n$ ("normal"). The invariant about these three new place nodes is described by the following points:

   a) The place node $n$ is always filled if **to_fault** doesn't occur and until the firing of *ReplyOutput* transition

   b) Whenever the *FaultHandleTO* transition fires, the place node $tS$ is filled while $n$ goes empty

   c) Whenever $tS$ is filled and $n$ is empty one and only one of the transitions sharing $s$ as output place is enabled.

   The point c) is very important because it leads to the filling of $s$. Filling $s$ means the disruption of the scope execution and eventually the firing of the *ReplyFault* transition.

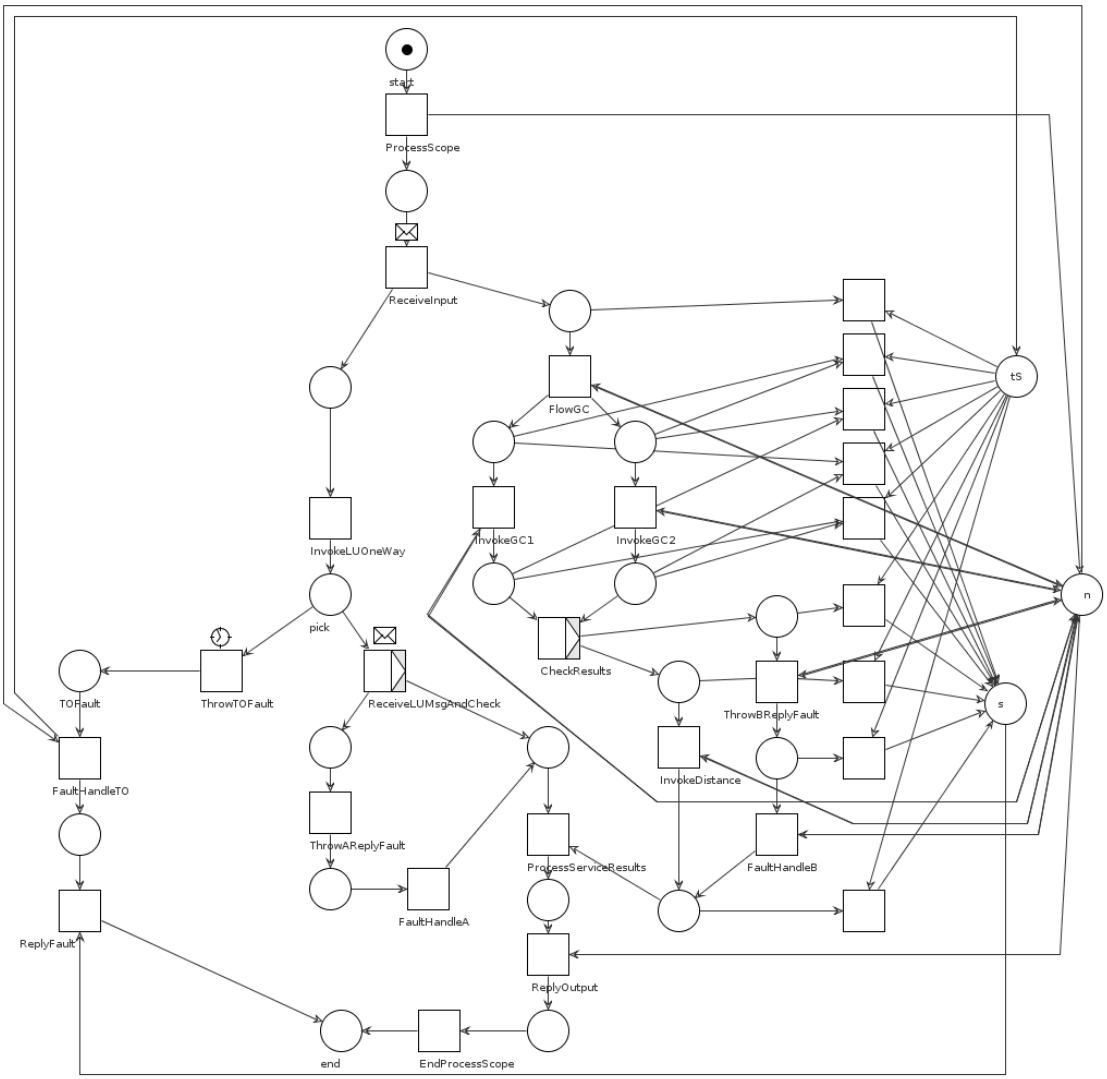


Figure 3.1: Workflow net of the orchestrator