Práctica de Integración de Aplicaciones

Integración de Aplicaciones

Grado en Ingeniería Informática - UDC

Autor:

Oscar Blanco Novoa

Carlos Freire Vilas

Grupo: 1.2

Repositorio: ia012

Índice

1. Introducción

Esta práctica consiste en diseñar un sistema para gestionar y facturar llamadas de clientes de una empresa de telefonía **Telco**. Se divide en dos partes: la primera de ellas consiste en desarrollar un servicio **Web REST** que permita gestionar los clientes y las llamadas. La segunda parte consiste en el desarrollo de un flujo **BPEL** para implementar un servicio de fidelización de clientes integrando el servicio desarrollado en la primera parte con otras tres aplicaciones de terceros.

En la primera parte se implementó un servicio web **REST** en tres capas (modelo, servicio y acceso al servicio) en java utilizando **JAX-RS** y **JAXB** y con una implementación ficticia del modelo que almacena tanto las llamadas como los datos de clientes en memoria en estructuras Map de java. Además se implementó un cliente que permite realizar llamadas a la API REST tanto en **JSON** como en **XML** cambiando un fichero de configuración.

La aplicación permite añadir clientes con sus datos, editarlos, eliminarlos y buscarlos por id, nombre o DNI. Permite añadir llamadas a los clientes, cambiar su estado y buscarlas por intervalos de tiempo. Todas las operaciones que devuelven listas lo hacen de forma paginada añadiendo links para poder navegar por toda la lista.

En la segunda parte de la práctica se utiliza el lenguaje **BPEL** para modelar el proceso de negocio de gestión de facturas para la empresa Telco que ha decidido poner en funcionamiento un sistema que automatice la gestión de facturas mensuales de sus clientes dentro de un programa de fidelización basado en un programa de puntos que les permitirá beneficiarse de descuentos en futuros consumos.

La empresa dispone actualmente de los siguientes servicios:

- **Servicio de Tarificación (RatingService)**. Permite obtener el precio individual de una lista de llamadas realizadas, en función de su duración, su tipo y el momento del día en el que se realiza. Adicionalmente, dispone de una operación que permite comprobar si un cliente se beneficia de algún descuento en el importe total de la factura para un mes de un año.
- **Servicio de Facturación (BillingService)**. Encargado de la generación y envío de la factura mensual para un cliente.
- **Servicio de Fidelización (RewardService)**. Encargado de la gestión de los puntos que acumula un cliente del programa de fidelización. Permite gestionar las cuentas de puntos de los clientes, es decir, proporciona operaciones para añadir y eliminar puntos de ellas.

Además, se integrará el servicio REST implementado en la primera parte de la práctica para gestionar las llamadas y los datos de los clientes.

2. Parte 1: Servicio web Rest

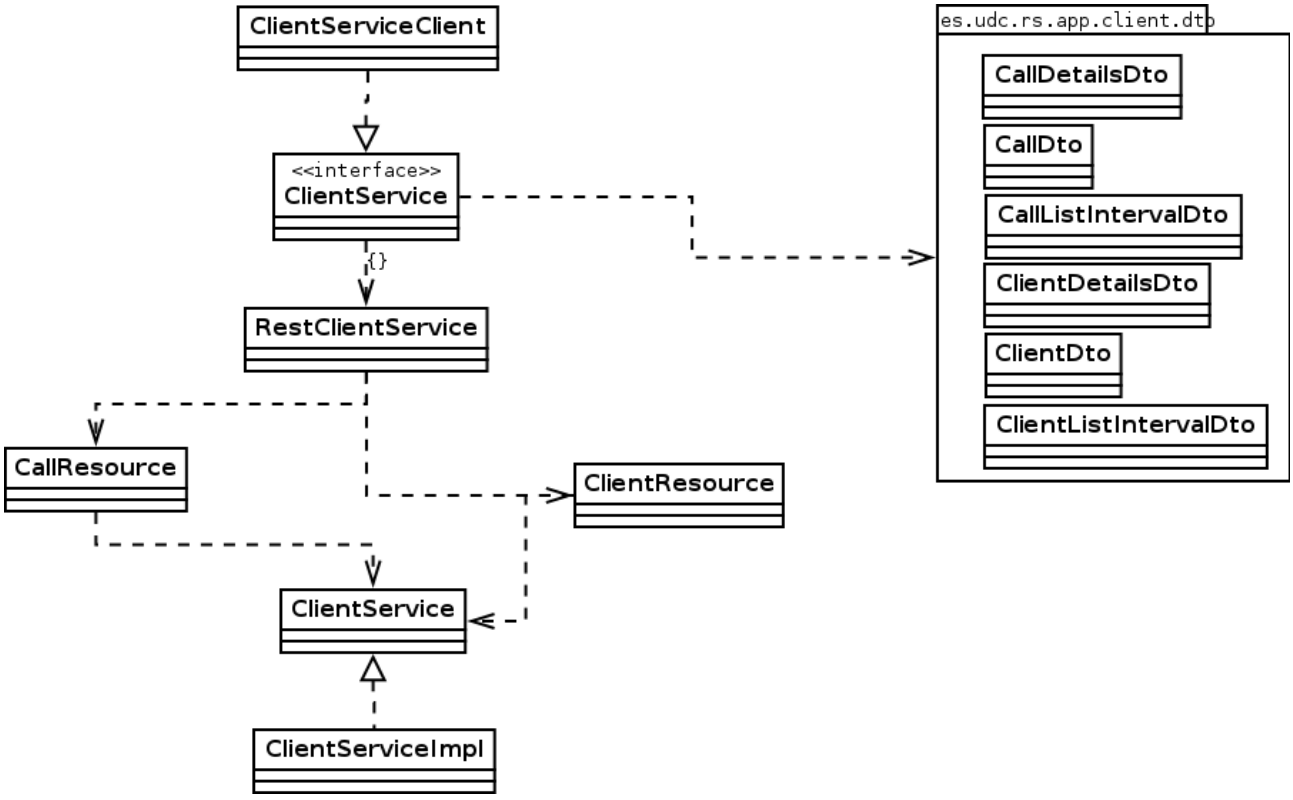
2.1. Diseño

2.1.1. Arquitectura Global

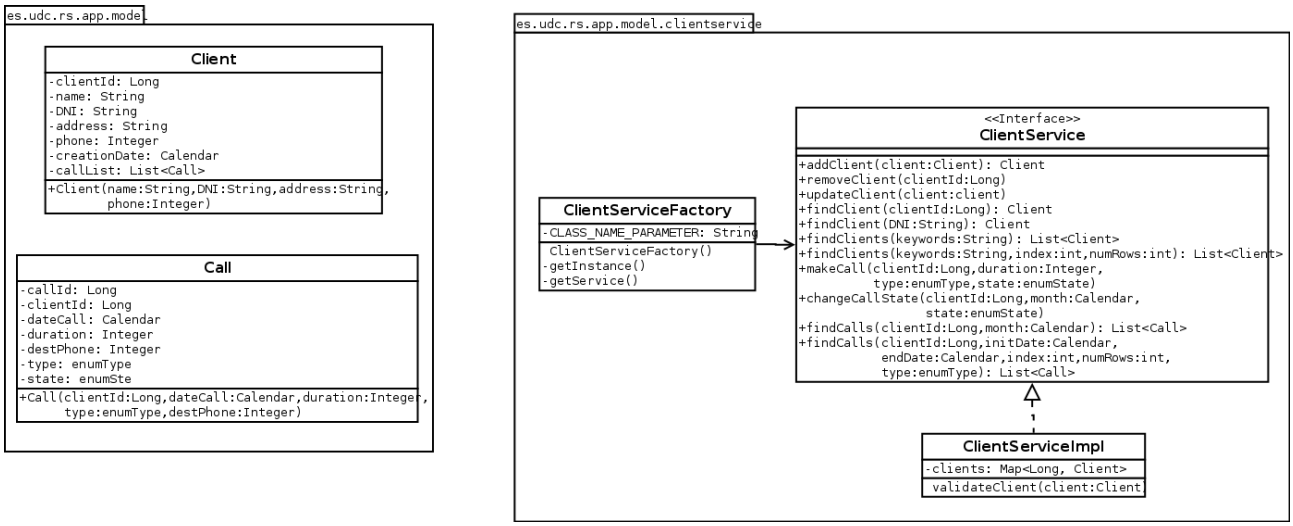
En este apartado se explica brevemente la arquitectura global de la aplicación propuesta. La aplicación consta de tres capas principales con la que el cliente va a hacer uso. Disponemos de una interfaz **ClientService** que proporciona el acceso al servicio mediante la tecnología REST.

La capa servicios se integra con la capa modelo a través de la interfaz **ClientService** (haciendo uso de ClientServiceFactory) y en este caso aportamos una solución ficticia para simular el uso de una base de datos real.

Para la comunicación entre dichas capas se han creado conversores que hacen la conversión entre las entidades de nuestra aplicación y los **DTOs** con anotaciones **JAXB** y en sentido contrario para ofrecerle la información al cliente.

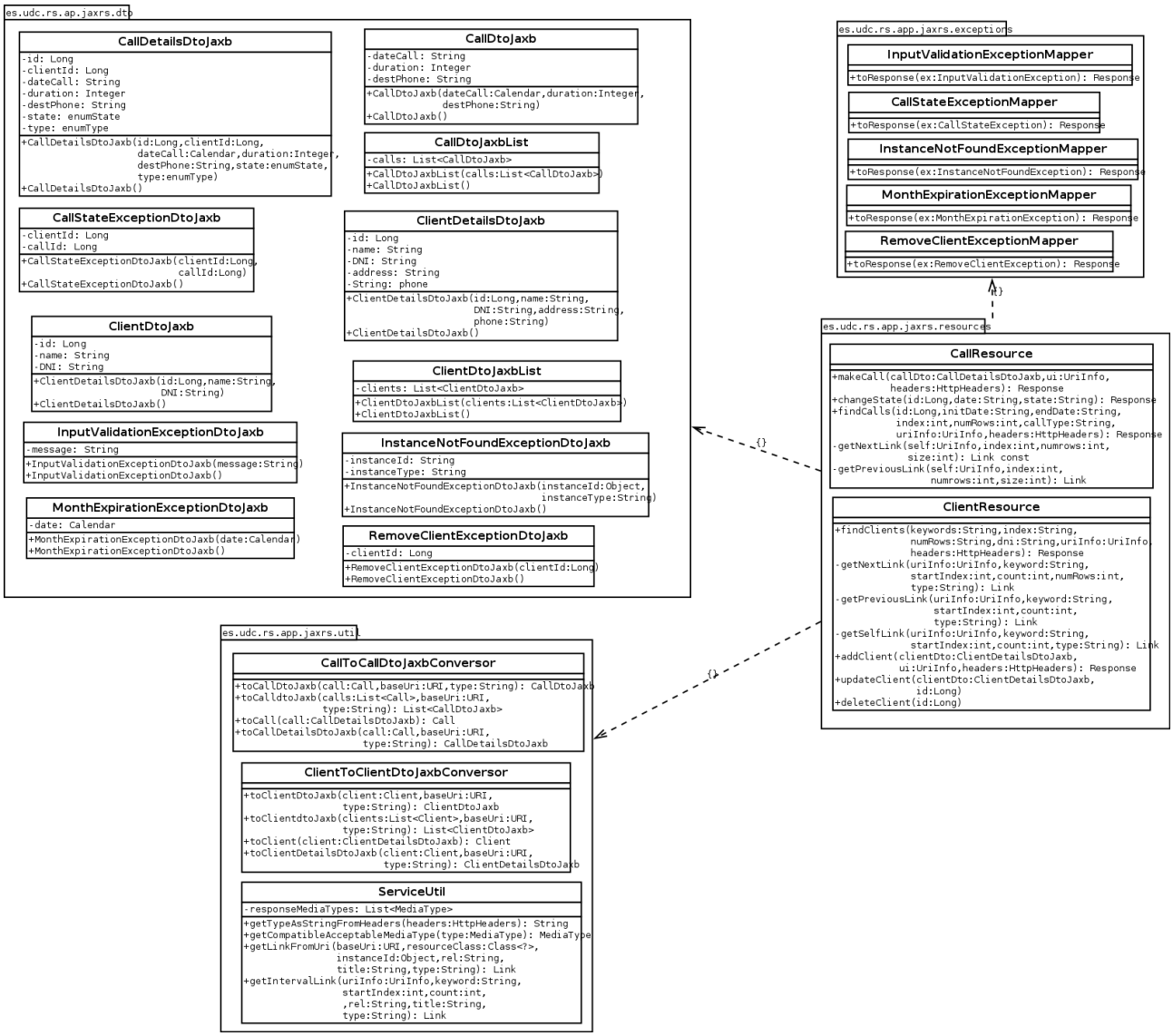


- **Capa Modelo**
  La implementación de esta capa viene dada de la interfaz ClientService que nos ofrece todos los métodos necesarios para el correcto funcionamiento. Estos métodos están implementados en **ClientServiceImpl**. Como ya hemos dicho, usamos una configuración ficticia usando la estructura Map para así conseguir simular una base de datos real.
  Al hacer uso del patrón Factoría podríamos implementar esta interfaz de muchas maneras distintas y seleccionar la opción que queramos en los ficheros de configuración.



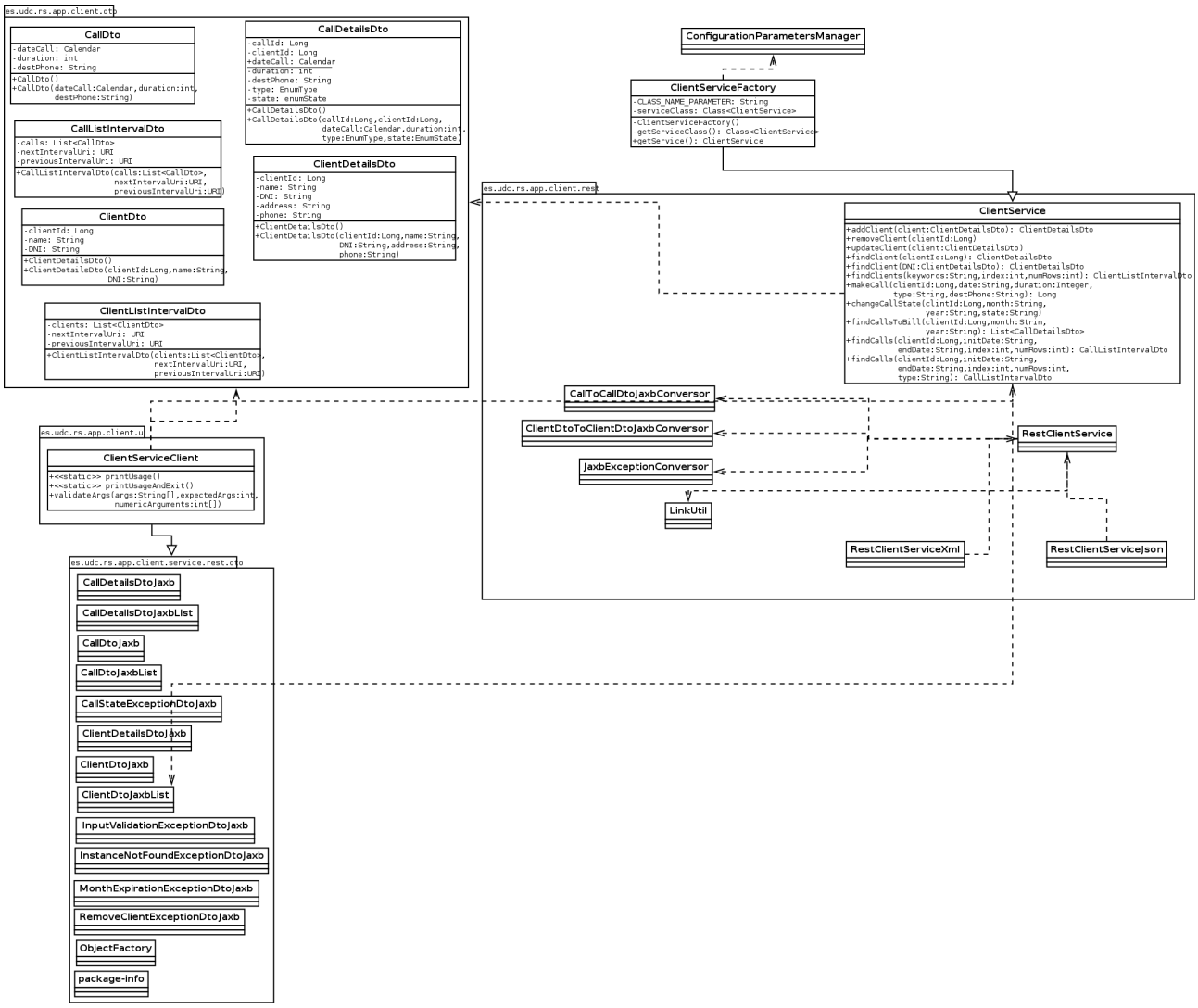
- **Capa Servicios**
  En esta capa se encuentran todas las clases dtoJaxb, que son clases que aportan la notación JAXB y a partir de las cuales se generarán los esquemas necesarios para la capa de acceso al servicio.
  Se incluyen los **Mappers** usados para devolver las excepciones del servicio, anotados como providers(@Provider).
  En esta capa se encuentran los Resources, son los encargados de llamar a la capa Modelo para que se aplique la lógica de negocio de las operaciones a realizar. Hemos implementado dos Resources, uno para Client y otro para Call(ClientResource y CallResource).
  También se encuentra en esta capa los conversores necesarios para convertir los tipos de la capa Modelo a clases anotadas con JAXB y una clase específica con la que podremos crear los links necesarios para ofrecer listas paginadas con enlaces rápidos a los elementos siguiente y posterior con el fin de ofrecer una nueva funcionalidad mas cómoda para el cliente de esta app.

  Para entrar un poco mas en detalle, esta app ofrece la posibilidad de poder acceder a los recursos de una manera rápida y sencilla con el uso de links. Con estos links podremos acceder a los recursos restantes que no tenemos actualmente pero coinciden con nuestros patrones de búsqueda. Con esto se consigue agilizar las consultas y descongestionar el tráfico entre cliente y servidor.



- **Capa de Acceso al Servicio y Capa Cliente**
  Esta capa es la encargada de interactuar con el cliente y la capa de acceso al Servicio. La comunicación entre esta capa y el cliente se realizará por linea de comandos, este comando es parseado por *ClientServiceClient*, que lee los parámetros y los formatea para pasarlos a la interfaz RestClientAppService (Capa de acceso al Servicio).

  Esta capa usa conversores para traducir los objetos creados en el cliente para formatearlos a XML y así enviárselos a la capa servicio cuando realizamos la comunicación y que esta capa realice la operación indicada y nos devuelva el resultado esperado.



- **Módulo Útil**

  Esta aplicación se complementa con este módulo en el que se incluyen las clases necesarias para el correcto funcionamiento. Se incluyen los enumerados usados para determinar que valores pueden atribuirse tanto a **state** como a **type** en las llamadas y también se incluyen las excepciones adicionales y la configuración **MOXy** necesaria para Json.

2.2. Compilación y instalación de la aplicación

Para la correcta ejecución de nuestra aplicación debemos determinar en los archivos de configuración del cliente lo siguiente:

- El lenguaje de intercambio a utilizar (XML/JSON)
- Indicar que servidor usar (jetty/tomcat)
- El numero de puerto

Una vez establecidos estos valores debemos compilar nuestra aplicación con *mvn install* para así generar los esquemas necesarios y la capa de acceso al servicio.

Iniciamos el servidor con un *mvn jetty:run* o *mvn:tomcat:run* desde el directorio del servidor. En nuestro caso en concreto vamos a hacer todas las pruebas necesarias desde el propio eclipse, tanto iniciar el servidor como ejecutar los comandos de la lista.
A continuación mostramos los comandos para cada función disponible:

- **addClient**

```
-a <name> <DNI> <address> <phone>
```

- **removeClient**

```
-r <clientId>
```

- **updateClient**

```
-u <clientId> <name> <DNI> <address> <phone>
```

- **findClientById**

```
-fid <clientId>
```

- **findClientByDNI**

```
-fdni <DNI>
```

- **findClients**

```
-fcs <keywords> <index> <numRows>
```

- **makeCall**

```
-mc <clientId> <date> <duration> <phoneDest> <type>
```

- **changeState**

```
-cs <clientId> <month> <year> <state>
```

- **findCalls**

```
-fcll <clientId> <dateInit> <dateEnd> <index> <numRows> <type>(OP)
```

- **findCallsBill**

```
-fcb <clientId> <month> <year>
```

2.3. Problemas Conocidos

Podemos destacar el método **changeState**, inicialmente lo teníamos definido como un método **PUT** pero debido a los problemas que nos encontrábamos y a que además no es idempotente decidimos definirlo como un **POST**.

3. Parte 2: Flujo BPEL

3.1. Diseño

3.1.1. Flujo BPEL

En esta parte de la práctica se trataba de integrar diferentes servicios web para crear un sistema de facturación unificado. Utilizamos el servicio **REST** desarrollado en la primera practica para obtener las llamadas de un cliente y mantener el estado de las mismas; un servicio de tarificación para obtener los precios de las llamadas y los descuentos; un servicio de facturación para generar las facturas y un servicio de recompensas para gestionar los puntos asignados a cada cliente en función de las llamadas realizadas.

A mayores se han implementado dos servicios SOAP ficticios para simular las confirmaciones de envío y cobro de factura desde **SoapUI**.

El flujo se inicia con la operación *gestionarFactura* que recibe el id de cliente, el mes y el año a facturar y devuelve las llamadas con su coste, el identificador de factura, la cantidad total a pagar y si la factura esta ya pagada o no.

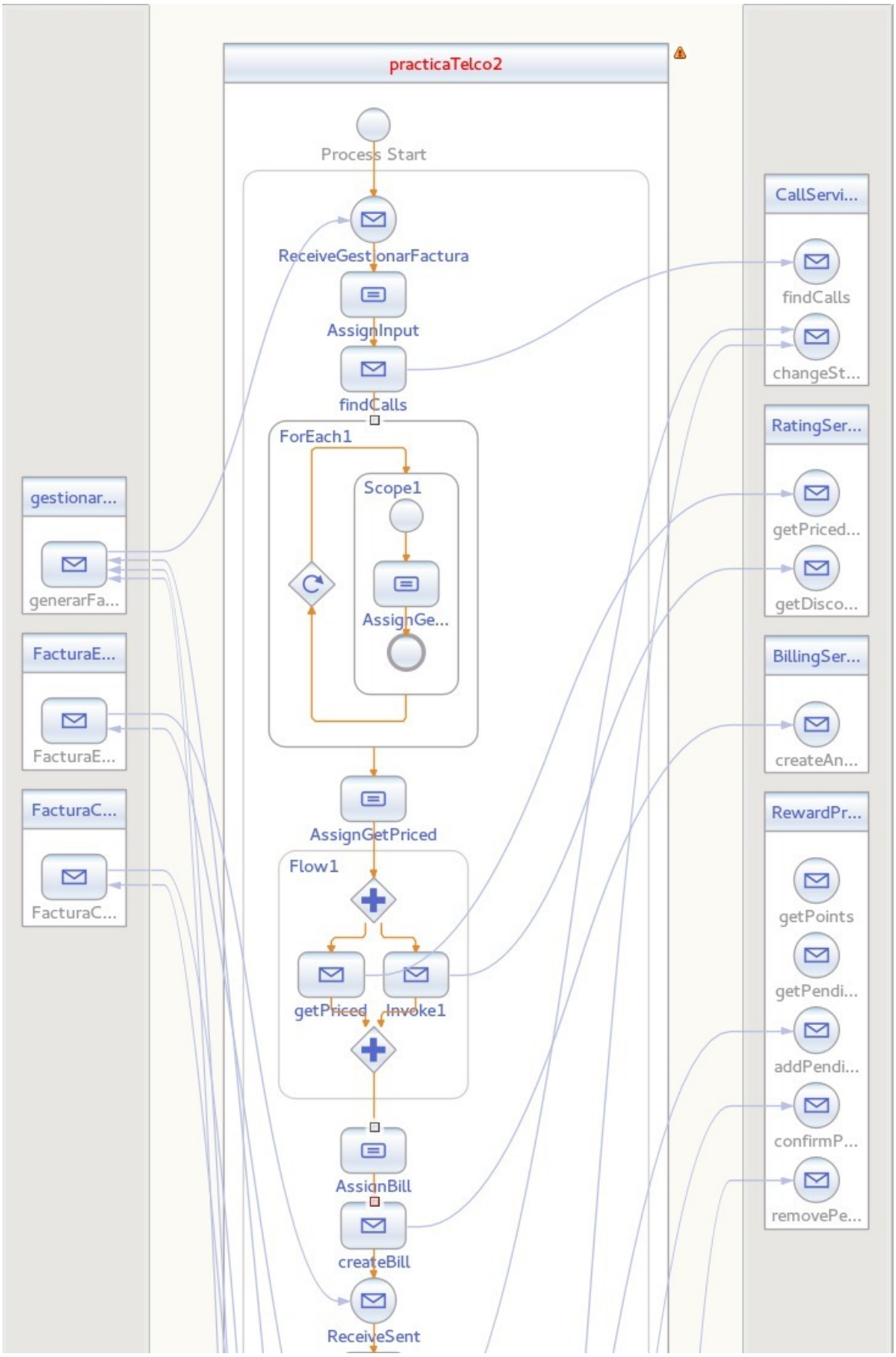
Más en detalle el flujo inicia invocando a la operación *findCalls* del servicio implementado en la primera parte de la practica, esta operación devuelve una lista de llamadas con detalles, posteriormente hacemos un bucle que recorre todas las llamadas y convierte la estructura para poder pasarla a la operación *getPricedCalls* que devuelve una nueva lista de llamadas con el coste correspondiente asignado. De forma concurrente a través del modificador del flujo "flow"se invoca la operación *getDiscount* que obtiene el descuento que que debe aplicarse al cliente.

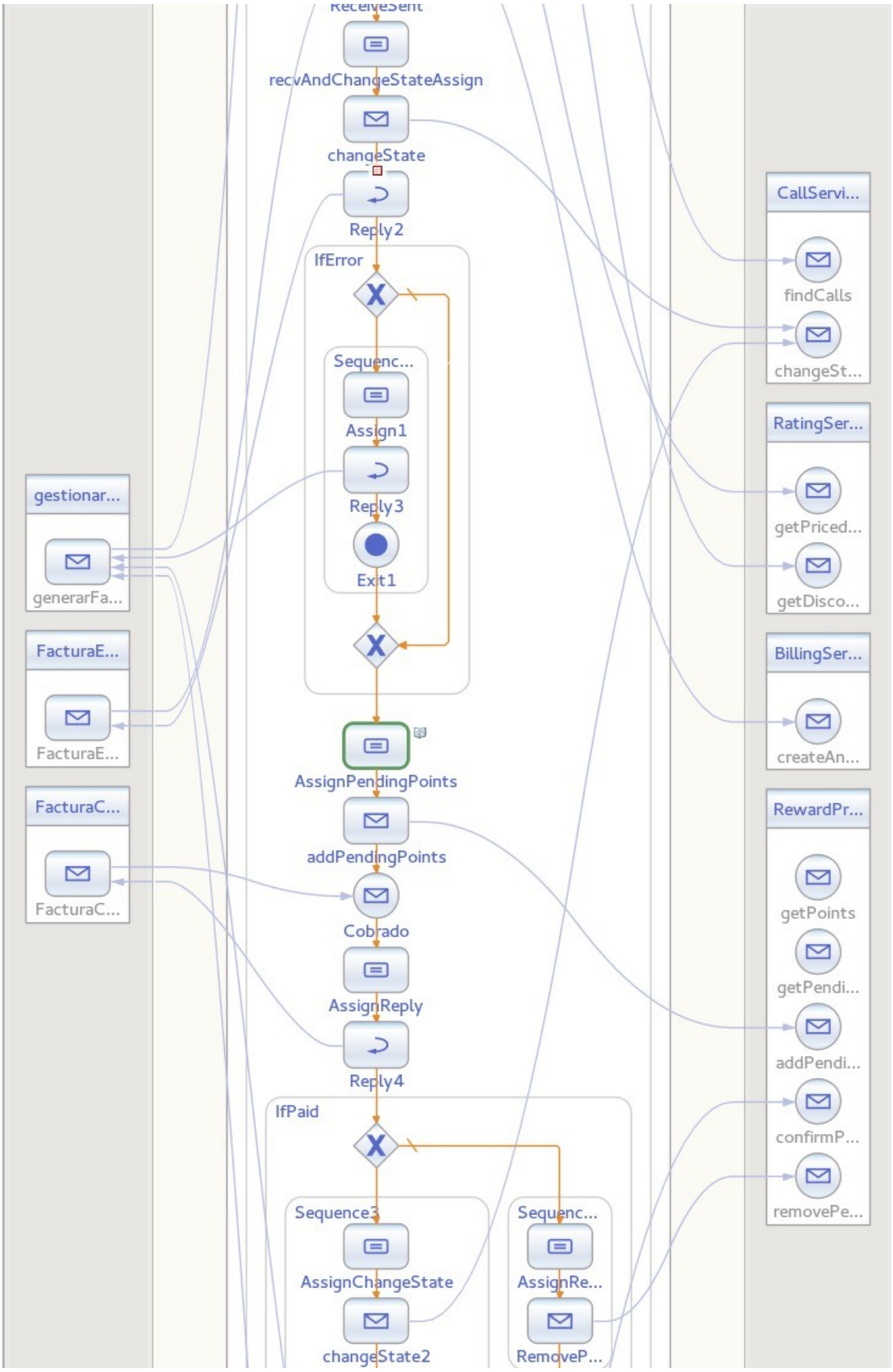
Posteriormente se llama a *createAndSendBill* que genera una factura enviándole los parámetros obtenidos en las dos operaciones anteriores. Una vez realizada esta llamada el flujo se queda esperando a que se confirme el envío de la factura.

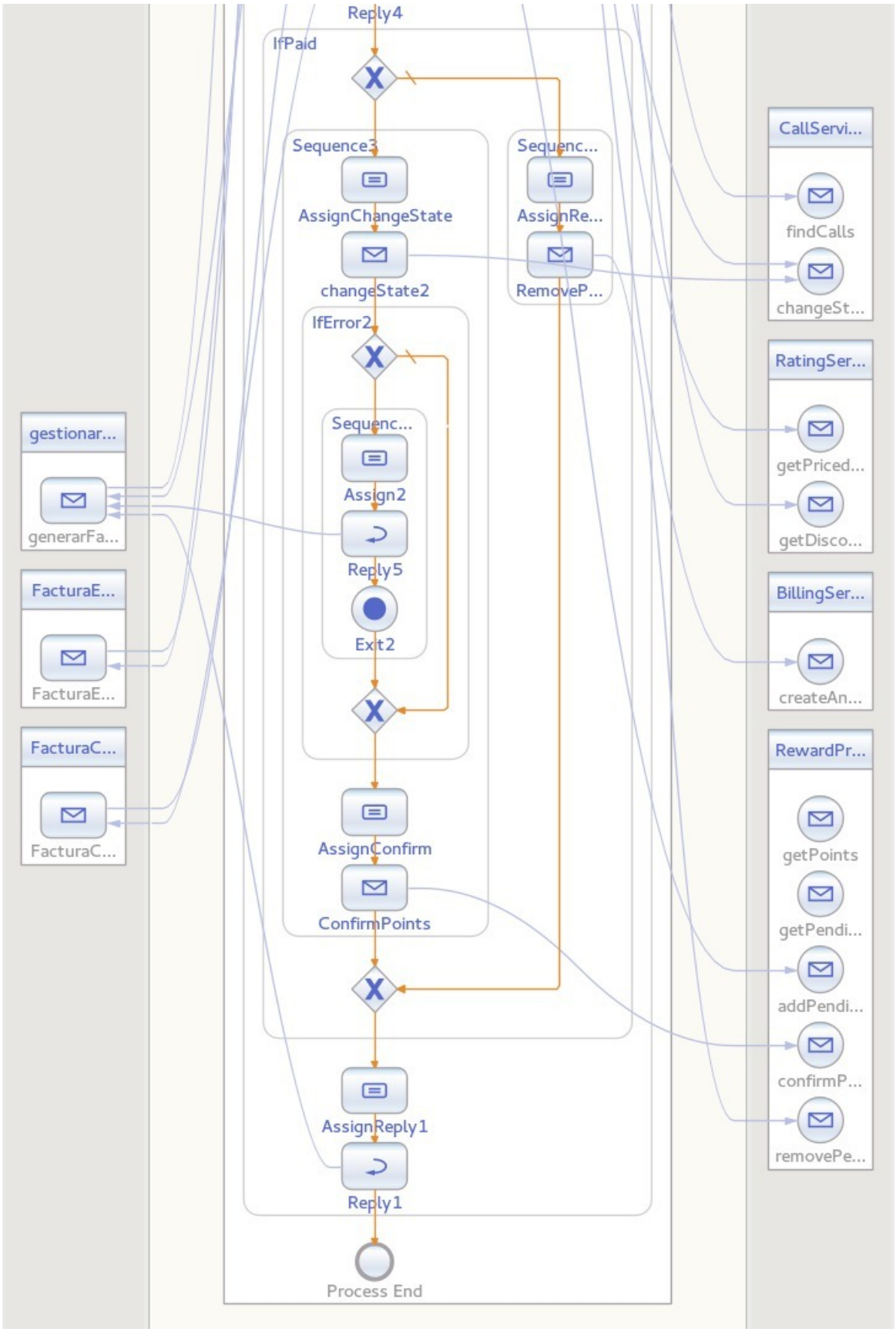
Una vez se recibe la confirmación de envío de la factura se asigna el estado de las llamadas a **BILLED**. Hay que tener en cuenta que esta operación puede devolver error en caso de que las llamadas no estén en el estado correcto, dicha excepción se controla comprobando el código de respuesta *HTTP* de la petición.

Una vez comprobado que no ha habido ningún error al cambiar el estado de las llamadas, se añaden los puntos como pendientes al cliente a razón de 1 punto cada 10 euros gastados (redondeando hacia abajo con la operación *floor*).

Es este momento el flujo se queda de nuevo esperando a que se reciba la confirmación de pago de la factura y, una vez se recibe, se comprueba si el estado es impagado eliminando los puntos que se habían añadido al cliente o pagado en cuyo caso se cambia el estado de las llamadas a **PAID**. Si no hay error en esta operación se confirman los puntos del cliente y el flujo *BPEL* finaliza devolviendo los parámetros arriba indicados.


practicaTelco2
Process Start
ReceiveGestionarFactura
AssignInput
findCalls
ForEach1
Scope1
AssignGe...
AssignGetPriced
Flow1
getPriced
Invoke1
AssignBill
createBill
ReceiveSent
gestionar...
generarFa...
FacturaE...
FacturaE...
FacturaC...
FacturaC...
CallServi...
findCalls
changeSt...
RatingSer...
getPriced...
getDisco...
BillingSer...
createAn...
RewardPr...
getPoints
getPendi...
addPendi...
confirmP...
removePe...

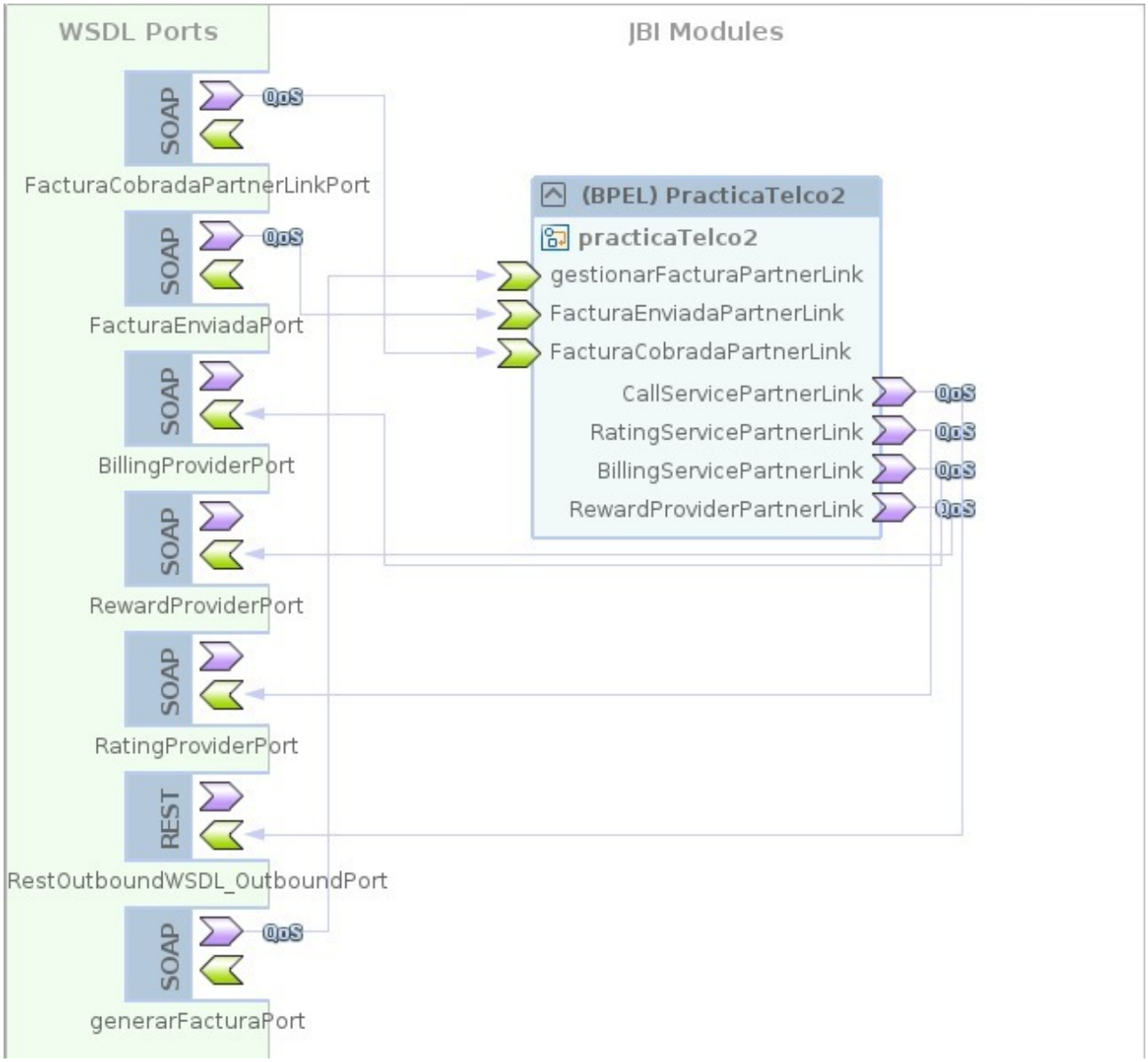

recvAndChangeStateAssign
changeState
Reply2
IfError
Sequenc...
Assign1
Reply3
Exit1
AssignPendingPoints
addPendingPoints
Cobrado
AssignReply
Reply4
IfPaid
Sequence3
AssignChangeState
changeState2
Sequenc...
AssignRe...
RemoveP...
gestionar...
generarFa...
FacturaE...
FacturaE...
FacturaC...
FacturaC...
CallServi...
findCalls
changeSt...
RatingSer...
getPriced...
getDisco...
BillingSer...
createAn...
RewardPr...
getPoints
getPendi...
addPendi...
confirmP...
removePe...


Reply4
IfPaid
Sequence3
AssignChangeState
changeState2
IfError2
Sequenc...
Assign2
Reply5
Exit2
Sequenc...
AssignRe...
RemoveP...
AssignConfirm
ConfirmPoints
AssignReply1
Reply1
Process End
gestionar...
generarFa...
FacturaE...
FacturaE...
FacturaC...
FacturaC...
CallServi...
findCalls
changeSt...
RatingSer...
getPriced...
getDisco...
BillingSer...
createAn...
RewardPr...
getPoints
getPendi...
addPendi...
confirmP...
removePe...

3.1.2. Composite Aplication

Generamos una *composite aplication* a la que añadimos el flujo bpel y los siete puertos para cada uno de los servicios utilizados. Esta aplicación es desplegada y ejecutada en el servidor **OpenESB Standalone**.

Añadimos cinco casos de test, el primero para ejecutar el flujo principal, otros dos para simular la notificación de factura enviada y cobrada y por ultimo otros dos para obtener los puntos pendientes y confirmados con el objetivo de verificar que el proceso funciona correctamente.



3.2. Importación y ejecución en OpenESB

Para importar la práctica de *BPEL* debemos importar desde el *menu File > import project > from ZIP* de OpenESB los archivos comprimidos en zip existentes en la raíz del repositorio.

Para ejecutar la practica hacemos click derecho sobre el proyecto de *BPEL* module y seleccionamos *çlean and build"* posteriormente hacemos lo mismo en la composite aplication.

Ahora desde la pestaña de servicios lanzamos el servidor *OpenESB Estandalone* y hacemos click derecho en la composite aplication y seleccionamos **deploy**.

Por último seleccionamos el caso de test1 y lo ejecutamos.

3.3. Problemas Encontrados

Debido a problemas de compatibilidad con la aplicación de **SoapUI** con el escritorio de GNOME que hacia prácticamente imposible trabajar con el. Hemos usado como alternativa **POSTMAN** para las peticiones *REST* y simulado las peticiones *SOAP* utilizando los test de OpenESB. Esto no supone ningún cambio en la práctica ni en los requisitos de la misma.

OpenESB nos dio verdaderos problemas generando el xml del flujo **BPEL**, especialmente con los concat, ya que de forma aleatoria, por algún error del programa desaparecían las comillas y se cambiaba la estructura del **XML** fallando la compilación lo que nos obligó reiteradas veces a editar manualmente el **XML** y corregirlo continuamente. Desconocemos si es un error general o es producido por el entorno que hemos utilizado.

La generación de los correlation nos resultó bastante costosa debido a no encontrar documentación suficiente para hacerlo, sin embargo creemos que finalmente ha quedado funcionando de forma correcta.