OPAALS PROJECT

Contract n° IST-034824

WP5: Integration with the Digital Ecosystem platform

Del 5.8 – Complete P2P infrastructure implementation

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: IST-034824

Project Acronym: OPAALS

Deliverable N°: D5.8

Due date: M46

Delivery Date: September 2010

Short Description:

Flypeer is basically a Peer-to-Peer Infrastructure developed using SOA (Service-oriented architecture) concepts where you can easily create, deploy and execute services taking advantage of all P2P benefits. The Flypeer framework tries to provide a fully distributed P2P environment, which executes different type of order service compositions. The proposed framework is considered at the deployment level of SOA, rather than the realisation level, and is targeted to business transactions between collaborating SMEs as it respects the loose-coupling of the underlying services.

Author: Paulo Siqueira, Fábio Serra, Denis Rosa

Partners contributed: Surrey

Made available to: Public

Versioning		
Version	**Date**	**Name, organization**
1	30/09/2010	Paulo Siqueira, Fábio Serra (IPTI)
2	11/10/2010	After review from Thomas Kurs (SUAS)

Quality check

Internal Reviewers: Thomas Kurs (SUAS)

Dependencies:

Achievements*	See http://projectkenai.com/projects/flypeer
Work Packages	WP 5: Integration with the Digital Ecosystem platform
Partners	Surrey
Domains	Computer Science domain: P2P networks, interactions, long-running transactions, distributed systems and networks, redundancy, diversity, consistency, concurrency, design for failure, formal semantics, behaviour patterns, lock mechanisms, formal analysis, distributed identity, SOA.
Targets	Computer, social and natural science researchers, SMEs, business analysts. Computer science communities: database, transactions, P2P networking and applications, formal methods.
Publications*	Not yet. See D5.7 for implementation publications
PhD Students*	None
Outstanding features*	The implementation of the p2p infrastructure has achieved its purposes and became an interesting alternative to webservices, as its has a modelling, transactional and fully distributed features.
Disciplinary domains of authors*	Paulo Siqueira, IPTI, Computer Science Fábio Serra, IPTI, Computer Science Denis Rosa, IPTI, Computer Science

The information marked with an asterisk (*) is provided in order to address Recommendation n. 4 from the Year 2 review report

Table of Contents

Introduction

This deliverable presents the high level documentation and coding for the Flypeer framework for P2P infrastructure. The architecture and details of implementation are deeply described in D5.7 “First increment of P2P infrastructure”.

Flypeer Dynamic P2P Infrastructure

This framework is based on the Java JXTA framework and can be ran in a standalone or in a web environment. In summary, it leverages JXTA with transaction support. Additional features are currently "work-in-progress", like a lock system.

The final goal of this project is to implement the Dynamic Virtual Super Peer (DVSP) model, a concept created by researchers from the University of Surrey, in UK; the transaction support that has already been implemented is the basic stepping stone for the DVSPs.

DVSPs, when ready, will allow us to have completely dynamic peer-to-peer network setups. This means that peers will gain and lose "super peer" status based on data regarding how trust worth they are, how much time they stay online and so on. In a network with lots and lots of peers, it should be easy to have plenty of super peers to maintain the network active.

Platform Requirements

The minimum requirement for running and developing with Flypeer is JDK version 6 or higher. With such an environment configured and running, adding the project's jar file to your project should be enough to starting developing a Flypeer Service. We will go through this process bellow.

How to Create the P2P Network

Before starting developing a service, you have to understand how to start a peer, and how to make it part of the whole network. Flypeer has two kind of peers: Simple peer and Rendezvous peer.

- Simple peers are those that can have services deployed and can start transactions but they don't give support for maintaining the network;
- Rendezvous peers are Simple peers on steroids: they also help propagating messages between peers, and thus help maintaining the whole network together.

Each simple peer must know at least one rendezvous peer to be discovered in the network.

Starting a Rendezvous Peer

You have two options when starting a Rendezvous node. You can do it using a graphical interface (GUI), or using the command line.

To do it using the GUI, just execute the *run.sh* (or *run.bat* under windows) file, and a window will appear. Select the *Rendezvous* option in the combo box in the bottom, enter a username (anything, actually) in the proper field and click *Login*.

To start the Rendezvous peer using the command line, type the following (on Linux):

./run.sh -i -r

or (on windows)

run.bat -i -r

After this, the system will ask for a name for the Rendezvous peer, type it and press enter.

Any of those approaches will make the Rendezvous peer run at the address 127.0.0.1 on port 9701, unless you pass *--public_ip IP* argument in the command line or create a properties file named *rendezvous_ip.properties* and set the new address in a property called *public_ip*:

rendezvous_ip.properties

public_ip=192.168.0.254:9701

Note: The argument has higher priority than the properties file.

Rendezvous List

The process of adding a new Rendezvous peer to the network involve configuring the address of the other known Rendezvous peers to the new one's rendezvous list. This can be done by creating a file named *seeds.properties* and putting it in the peer's classpath. Something similar to this should work:

seeds.properties

seed_1=tcp://192.168.0.253:9701
seed_2=tcp://192.168.1.128:9701

Note: In our tests, to make several Rendezvous peers working properly together, we had to put all the other Rendezvous peer's address into each other's seeds.properties files. You might have to do this as well if you find that your Rendezvous peers don't seem to be communicating correctly. It should be investigated in the future.

Example:

rendezvous peer 1

seed_2=tcp://192.168.1.128:9701

rendezvous peer 2

seed_1=tcp://192.168.0.253:9701

Here, the keys *seed_1* and *seed_2* can actually be anything. The value side of the properties is what really matters.

Starting a Simple Peer

In order to work properly, you should define at least one rendezvous's IP in a file named *seeds.properties* and then put this file in the default package of your peer's project. When you start the peer, if this file is not found, the peer will try to find a Rendezvous at the address *tcp://127.0.0.1:9701*.

Example:

seeds.properties

```
seed_1=tcp://192.168.0.253:9701
```

Now, when it comes to start the Simple peer, there are two common scenarios. In one of them, you are just interested in starting a Simple peer, holding some services, and that's all. In this case, life is easy. As with the Rendezvous peer, there is both a GUI and a command line option.

Using a GUI, execute that same *run.sh* (or *run.bat* under windows) file. Choose *Simple Peer* in the combo box in the bottom of the window, type a user name and password (can be anything for now) and click *Login*. Wait a few moments and the peer should be up and running.

The command line option is also simple. Type (on Linux):

./run.sh -i

or (on windows)

run.bat -i

When it asks for the Identity Provider, choose *Default*. Then type anything as the username. The peer will be started in a few moments.

The second scenario is a little bit more complex. When you want a peer that will start a transaction, you will have to write some code to start the peer and the transaction itself. You will show here what you have to do to start a peer programmaticaly, and later in this document you will see how to start a transaction.

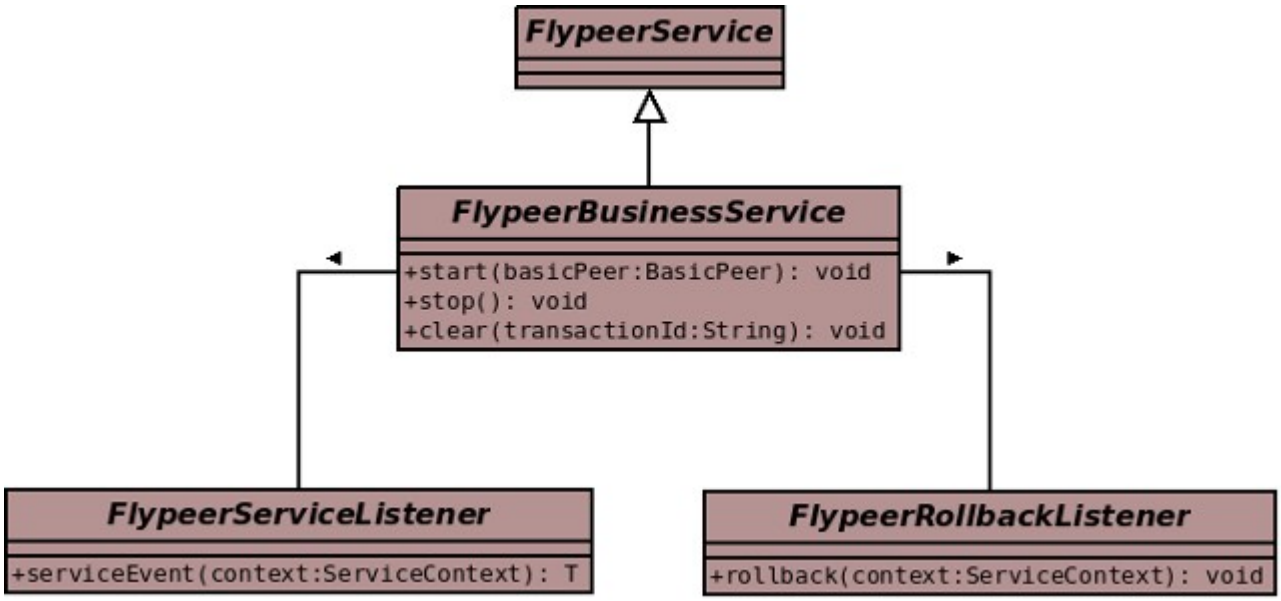
The code bellow start a peer from Java source code:

```
1.CommandSimplePeer commandSimplePeer = new CommandSimplePeer();
2.commandSimplePeer.start(peerName, peerStatusNotifier);
```

A few moments after this, the peer will be set and running, and you will be able to start transactions. This bootstrap for a simple peer will be more explained in details following.

Creating Services

The diagram bellow shows the relationship between the service related classes:



The main component you will be concerned with is the *FlypeerBusinessService* interface. This is the interface you have to implement when creating your own services. Lets go through the methods declared in this interface:

- **start(BasicPeer basicPeer):** This method is called when the service is deployed, which happens when the peer is started. A BasicPeer object will be passed in and can be used by the service to get some simple information about the host peer;
- **stop:** This method is called when the service is going to be undeployed, and can be used to clean up resources that the service might have opened, like a database connection for example;
- **clear:** Despite its name, this method is called after each transaction finishes successfully, and can be used to close / clear transaction specific resources;
- **getServiceListener:** This method must be implemented to return an implementation of the FlypeerServiceListener interface. This interface is where the actual logic of the service is executed. We will talk about this interface a little more shortly;
- **getRollbackListener:** Similar to FlypeerServiceListener; but this method is called when a transaction fails, giving you the opportunity to implement some action to respond to the failure of a service invocation. See bellow for more details.

FlypeerServiceListener

When implementing this interface, you will have to implement only one method: *serviceEvent*. This method receives a single parameter of the type *ServiceContext*, which contains information regarding the call, including the service parameters.

FlypeerRollbackListener

This interface is very similar to the last one. Again you have to implement only one method; this time, the method *rollback*. This method also receives a parameter of the type ServiceContext, which is equivalent to the one passed to the *serviceEvent* method of the *FlypeerServiceListener* interface.

The *rollback* method will be called when a transaction fails (or at least the part of the transaction which this service participated) and serves as a hook to allow the service developer to take the appropriate actions.

Deploying Services

Now that you finished writing your service, its time to deploy it. To do it, you have to create a jar package, with one configuration file in it, as described bellow.

The service deployment system of Flypeer uses the ServiceLoader class, available in Java 6 and above. If you want to understand more how it works, you can see its javadoc here: http://java.sun.com/javase/6/docs/api/java/util/ServiceLoader.html. Otherwise, you can just follow the instructions in this page.

First, create a file named *br.org.ipti.flypeer.services.FlypeerBusinessService*, inside a directory named *META-INF/services*. This directory must be placed in the root of the same jar file where service will be packaged in.

Inside this file, you have to add the full name of your service implementations, one in each line. For the Flight Booking example we have in the source code, this is how the file looks like:

```
br.org.ipti.flight.service.server.BAFlightBookingService
br.org.ipti.flight.service.server.LufthansaFlightBookingService
```

Its interesting to note that you can have as many services per package as you want - just add each service in this file and it will be deployed. Also, keep in mind that we don't have support for hot deploying yet, so if you decide to change / update your service package, you will have to restart the peer.

To be sure that the service was deployed correctly, you can check the peer logs. You should see something like the line bellow for each service deployed in that peer.

```
INFO: Service flypeer://LUFTHANSA-FLIGHT-BOOKING has been deployed in the group FLYPEER_DEFAULT
```

Of course, the service and the group name will change depending on your service.

Creating Transactions

Flypeer offers two different approaches to compose transactions: using an XML file, or programmatically. The main difference is that later is more flexible, so that you can be "fancy" when composing transactions; but you have to use Java code for the composition in this case.

Here is a sample code, composing a parallel transaction:

```
01. Service service01 = new Service("PARALLEL_TEST", "service01");
02. service01.addDependency("serviceName", "serviceParams01");
03. Service service02 = new Service("PARALLEL_TEST", "service02");
04. service02.addDependency("serviceName", "serviceParams02");
05. Service service03 = new Service("PARALLEL_TEST", "service03");
06. service03.addDependency("serviceName", "serviceParams03");
07.
08. ParallelFlow parFlow = new ParallelFlow();
09. parFlow.addService(service01);
10. parFlow.addService(service02);
11. parFlow.addService(service03);
12.
13. TransactionFlow mainFlow = new TransactionFlow();
14. mainFlow.addTransactionResult("result01", service01);
15. mainFlow.addTransactionResult("result02", service02);
16. mainFlow.addTransactionResult("result03", service03);
17. mainFlow.setGenericFlow(parFlow);
18.
19. TransactionInitiator transInitiator = new TransactionInitiator(peer);
20. Map<String, Serializable> params = new HashMap<String, Serializable>();
21. params.put("serviceParams01", "service01");
22. params.put("serviceParams02", "service02");
23. params.put("serviceParams03", "service03");
24. transInitiator.runTransaction(mainFlow, params, 15000L);
```

The code above is separated in four blocks. In the first one, we create references to all the services that will be called during this transaction. In this case, we will call the same service three times. The service receives one parameter, called *serviceName*. Since we are calling the same service more than once, we have to define aliases to the parameter, so that we can correctly define the values for each call.

In the second block, we create the parallel transaction: in this case, a flow of the three services configured previously. Next, in the third block, we configure the transaction and the expected results from it. To do that, we add each service's result to the internal *transaction results* list, using the *addTransactionResult* method. Then we set the main execution flow of the transaction to be the parallel flow we created in the second block.

The last step, showed in the fourth block, is to create a *TransactionInitiator*, and add the parameters to sent to the services. With this configured, a simple call to *runTransaction* starts the transaction and blocks the execution until it completes, or the timeout is reached. If you want to make this call asynchronously, you can use the method *startTransaction* instead. We will look into the *TransactionInitiator* class in the next section.

This code was extract from the *ParallelCompositionTest* test class. There are more complex examples available in the *junit* tests of the project that can be explored (http://projectkenai.com/projects/flypeer).

Starting a Transaction: Class TransactionInitiator

As you saw above, the *TransactionInitiator* class is the class used to execute transactions. After you design and create your transaction, you can use the methods provided by this class to execute transactions both synchronously or asynchronously. In the first case, you use the *runTransaction* method. In the second case, the *startTransaction* method.

The methods receive different parameters, the most important ones being:

- **Map<String, Serializable> parameters**: a map of name / value pairs with the parameters that will be passed to the services in the transaction; the name of the parameter also determines who receives the value, as demonstrated in the example in the last section;
- **FlypeerResponseListener**: you have to pass in an implementation of this interface when executing transactions asynchronously. The methods to be implemented in this case are:
    - **processResponse**: called when the transaction ends and is successful. The keys in the map are the ones mapped in the *addTransactionResult* call, as demonstrated in the example in the last section;
    - **processFail**: This method is called when an error occurs during the execution of the transaction. You can check the errors threw by the services using the map of exceptions passed in. Please note that flypeer will call the rollback hooks of the transaction before calling this method.

Putting everything together

To finish, lets understand how to actually create the TransactionInitiator. This is a small catch in this process because you need a *peer* instance to do that. This is how you do it:

```
1.  public class MyPeer implements PeerStatusNotifier {
2.          private CommandSimplePeer peer;
3.
4.              public MyPeer() {
5.              peer = new CommandSimplePeer();
6.
7.               AuthnData authnData =
8.               new AuthnData(AuthnData.TYPE.DEFAULT, "username", "password");
9.               peer.start(authnData, this);
10.                             }
11.
12.         public void peerStarted() {
13.         TransactionInitiator initiator = peer.createTransactionInitiator();
14. // create, compose and execute transaction here, as demonstrated previously
15.                                     }
16.                                                         }
```

In this example, our class is implementing *PeerStatusNotifier*. This a good practice (actually, almost mandatory), to avoid trying to execute transactions before the peer is fully initialized. In this case, we create a peer reference in our constructor and start this peer. To the *start* method, we pass in login information,which can be dummy for now, and *this*, saying that our class is to be notified when the peer is ready.

The *peerStatusNotifier* is called when it is ready, and inside its implementation we create a *TransactionInitiator*. This is the object we need to execute transactions, as seen before.

Identity Flow in Flypeer

Identity flow, a project lead by WIT, was fully integrated to Flypeer to give it support to authentication and identification. The Identity Flow supports new identities solutions into the network with no centralized authority.

Identity Provider

To support authentication, the network must have at least one peer with Identity Provider set up. The Identity Provider is the component from Identity Flow which does the authentication process and generates a credential to the peer when the authentication process is successfully completed.

For now, there is a technical limitation that could not be addressed due to time constraints. We can have only one identity provider running in the Flypeer network at a time. To get a peer with the Identity Provider running, the peer must be started with the peername '*default'*. After that, any peer can be executed, be authenticated in the network and receive credentials.

Authenticating

A peer to be authenticated and receive a valid credential it must choose the authentication type. For now, Flypeer supports two types of authentication: Default and Guigoh's authentication.

If the Default authentication is chosen, a very simple authentication will be executed and no valid credential will be obtained. Because of that, any service which calls for a valid credential will not accept anything from this peer, then any transaction will not can be executed completely.

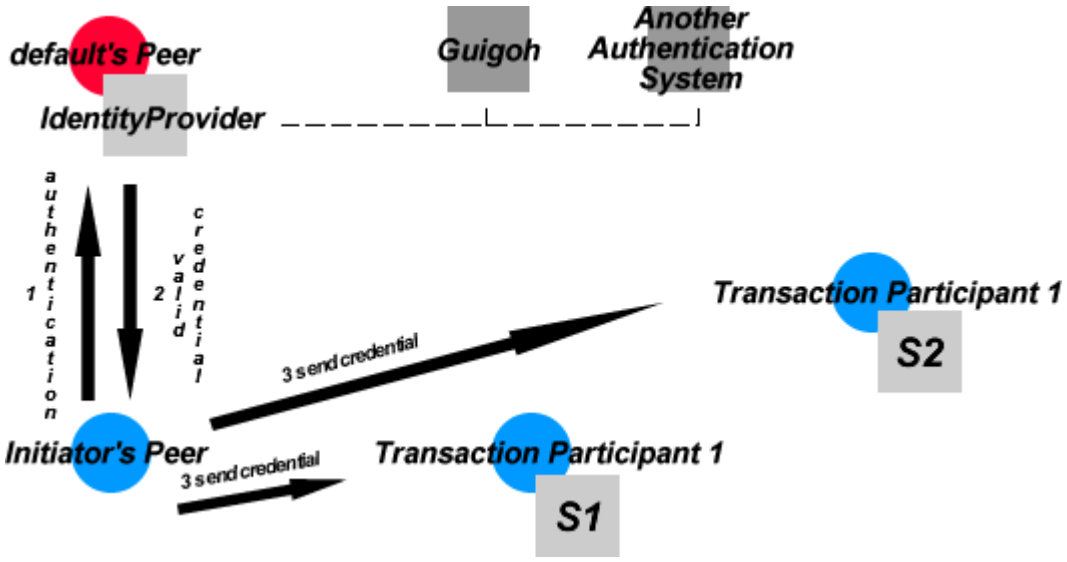
Otherwise, if the Guigoh's authentication is chosen the peer will receive a valid credential and any type of transaction and service can be reached. But, a peer to be authenticated by Guigoh must have a Guigoh's account (http://www.opaals.org.br).

Leveraging both Flypeer and IdentityFlow, it is also possible to create new authentication methods besides the ones mentioned here, which gives some extra flexibility.

Credential

Once a peer is successfully authenticated, it receives a credential from the Identity Provider and then when any transaction is started the credential is sent to all peers into the transaction.

The Credential can be accessed by the *getInitiatorCredential* method of the *ServiceContext*'s object received as a parameter in the *serviceEvent* method of the implemented service. In figure X, the diagram represents these classes.



In summary, what happens is that, since each participant in the transaction has access to the transaction's initiator's credential, it can then validate if the messages they are receiving for each transaction come from the correct peer, instead of a fake one, for example.

Besides transaction validation, there is at least one more use for credentials. They can be used by the services to implement user related things, like file storage with files belonging to the correct users. It is important to note that the trust of this behaviour has not being fully investigated though.

Consuming Flypeer Services Using SOAP

An alternative option for consuming flypeer services is using a standard SOAP client. Flypeer offers what we call a *Gateway Service*. This service exposes services deployed in the flypeer network to the "external" world through a SOAP interface.

Each service has an WSDL file used to deploy it internally. When accessing externally, this WSDL file is transformed, so that it is a valid one, in terms of the WSDL standard - our internal version is not usable from outside the flypeer network, since it is a simplified version, if compared to the standard.

To access this external version of the WSDL file for the service you are trying to call, use an URL in the following format:

http://{flypeer-node-host}:7070/{group}/{service-name}?wsdl

Calling this URL will return a WSDL file that can be used with any standard SOAP client. In our tests, we use *Netbeans*. The option is in the *Services* tab, and is named *Web Services*. Using this option, the IDE generates code that calls our flypeer service through a SOAP interface, without having to do any additional programming. Also, some people (thank you Jaakko) reported being able make a call using *python*.

In the URL above:

- **{flypeer-node-host}** is the domain or IP address of the peer that is acting as a gateway
- **{group}** is the peer group where the service is deployed (ignore this if the service was deployed in the default group)
- **{service-name}** is the name of the service

Having this URL should be enough to call a flypeer service that is already deployed, using SOAP. If you need additional support, please feel free to contact us using any of our mailing lists.

Note: in the current version, all nodes try to act as a Gateway service node. In the future, this will be configurable.

Example

One of the example services we have in the source code of Flypeer is a flight booking service. The one we are going to use in this example is called **LUFTHANSA-FLIGHT-BOOKING**. This service is configured to be deployed in the default peer group. This is what the URL for reading the service's external WSDL could look like:

http://localhost:7070/LUFTHANSA-FLIGHT-BOOKING?wsdl

Here we are using localhost, because we are accessing a service in our own machine. Also, note the port **7070** being used. Currently, all SOAP access are done using this port. Further develpment is needed to make it.

Just for reference, this is the WSDL file returned when accessing the URL mentioned above:

```
01.<?xml version="1.0" encoding="UTF-8"?>
02.<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                    03.         xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
                    04.         xmlns:xs="http://www.w3.org/2001/XMLSchema"
                    05.         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                    06.         xmlns:ns1="http://org.apache.axis2/xsd"
                    07.         xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                    08.         xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
                    09.         xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
                    10.         xmlns:ns="http://LUFTHANSA-FLIGHT-
                    BOOKING.ws.services.flypeer.ipti.org.br"
                    11.         targetNamespace="http://LUFTHANSA-FLIGHT-
                    BOOKING.ws.services.flypeer.ipti.org.br">
12.
13.   <wsdl:documentation>LUFTHANSA-FLIGHT-BOOKING</wsdl:documentation>
14.   <wsdl:types>
15.     <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified"
16.                 targetNamespace="http://LUFTHANSA-FLIGHT-
BOOKING.ws.services.flypeer.ipti.org.br">
17.         <xs:element name="bookFlight">
 18.           <xs:complexType>
19.             <xs:sequence>
20.               <xs:element minOccurs="0" name="flightTarget" nillable="true"
type="xs:string"/>
21.             </xs:sequence>
22.           </xs:complexType>
23.         </xs:element>
24.         <xs:element name="bookFlightResponse">
25.           <xs:complexType>
26.             <xs:sequence>
27.               <xs:element minOccurs="0" name="return" nillable="true"
type="xs:string"/>
28.             </xs:sequence>
29.           </xs:complexType>
30.         </xs:element>
31.     </xs:schema>
32.   </wsdl:types>
33.
34.   <wsdl:message name="bookFlightRequest">
35.     <wsdl:part name="parameters" element="ns:bookFlight"/>
36.   </wsdl:message>
37.   <wsdl:message name="bookFlightResponse">
38.     <wsdl:part name="parameters" element="ns:bookFlightResponse"/>
39.   </wsdl:message>
40.
41.   <wsdl:portType name="LUFTHANSA-FLIGHT-BOOKINGPortType">
```

```
42.     <wsdl:operation name="bookFlight">
43.       <wsdl:input message="ns:bookFlightRequest" wsaw:Action="urn:bookFlight"/>
44.       <wsdl:output message="ns:bookFlightResponse"
wsaw:Action="urn:bookFlightResponse"/>
45.     </wsdl:operation>
46.   </wsdl:portType>
47.
48.   <wsdl:binding name="LUFTHANSA-FLIGHT-BOOKINGSoap12Binding"
49.               type="ns:LUFTHANSA-FLIGHT-BOOKINGPortType">
50.     <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
51.     <wsdl:operation name="bookFlight">
52.       <soap12:operation style="document" soapAction="urn:bookFlight"/>
53.       <wsdl:input>
54.         <soap12:body use="literal"/>
55.       </wsdl:input>
56.       <wsdl:output>
57.         <soap12:body use="literal"/>
58.       </wsdl:output>
59.     </wsdl:operation>
60.   </wsdl:binding>
61.
62.   <wsdl:service name="LUFTHANSA-FLIGHT-BOOKING">
63.     <wsdl:port name="LUFTHANSA-FLIGHT-BOOKINGHttpSoap12Endpoint"
64.               binding="ns:LUFTHANSA-FLIGHT-BOOKINGSoap12Binding">
65.       <soap12:address location="http://192.168.0.193:7070/LUFTHANSA-FLIGHT-
BOOKING.LUFTHANSA-FLIGHT-BOOKINGHttpSoap12Endpoint/"/>
66.     </wsdl:port>
67.   </wsdl:service>
68.</wsdl:definitions>
```

Using the WSDL above, your tool should be able to generate client code to call the flight service.

Drawbacks

There is a catch when using this strategy for calling flypeer services. Your call will be a simple, direct call. Unless you are calling a service that is itself composing a transaction, this call is not part of any transaction. Actually, internally a single service transaction (a transaction with only one service) is generated and executed for each SOAP call received.