OPAALS PROJECT

Contract n° IST-034824

WP3: NetworkP2P networks and DE design principles

Del3.14 – Simulation and test of Semantic Search in a DE infrastructure

Project funded by the European Community under the “Information Society Technology” Programme

Contract Number: IST-034824

Project Acronym: OPAALS

Deliverable N°: D3.14

Due date: March 2010

Delivery Date: July 2010

Short Description: This code and report deliverable describes the implementation, deployment and transactional consumption of P2P services using the DE infrastructure Flypeer. It also covers time measurements of the Semantic Search Service in an intercontinental setting performed with the help of the multi-agent simulation framework EvESim.

Author: SUAS (T. Kurz, C. Rücker, T. Heistracher), IITK (A. Jain)

Partners contributed: SUAS, IITK

Made available to: OPAALS Consortium and European Commission Versioning

Versioning		
Version	**Date**	**Name, organization**
0.1	19/02/2010	Service deployment and testing for Flypeer (SUAS)
0.2	28/05/2010	Added: semantic search
0.3	15/07/2010	Finalization
1.0	29/07/2010	Submission

Quality check

Internal Reviewers: Antonio Margarito (T6-ECO), Francisco José Lacueva Pérez (ITA)

Dependencies:

Achievements*	Documentation of the creation of a Flypeer Service. Example of a Flypeer use-case. The semantic search component was modified for using Flypeer communication.
Work Packages	WP3, WP5
Partners	SUAS, IITK
Domains	Computer Science
Targets	Helping with the integration of semantic search with Flypeer P2P and first usage of Flypeer in different versions including feedback on usability, documentation and functionality to IPTI.
Publications*	-
PhD Students*	-
Outstanding features*	-
Disciplinary domains of authors*	T. Kurz (Information Technologies, Software and Systems Engineering, Interpersonal Communication) C. Rücker (Information Technologies, Software and Systems Engineering) T. Heistracher (Information Technologies, Software and Systems Engineering, Biophysical Modelling) A. Jain (Computer Science and Software Engineering)

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

Contents

Executive Summary

This code and report deliverable describes the implementation, deployment and transactional consumption of P2P services using *Flypeer* and also provides time measurement results of the *Semantic Search Service* deployed in that P2P infrastructure.

First, the DE infrastructure *Flypeer* is covered as regards the implementation, deployment and transactional consumption of user-generated services in the context of an example service. The goal of this part of the deliverable is to inform readers with a general P2P understanding – but being not necessarily familiar with technical details regarding *Flypeer* – about steps necessary to configure and use that environment for their own services. A process-oriented description of both a *Flypeer*-based service implementation and a service consumption is given. Topics such as service description and rendezvous peer configuration are covered accompanied by a series of example listings and a guideline for verification for successful instantiation of the related services.

In the second part of this work, the utilization of this infrastructure with the *Semantic Search Service* is covered that in parallel to its search functionality serves as stress test for the underlying infrastructure that is on an intercontinental scale. This semantic search is implemented as infrastructure service that is accessed via the EvESim multi-agent simulation environment for the performance measurements. Distributed timer-based software agents execute (semantic search service) calls over the *Flypeer* infrastructure utilizing the transactional model implemented within the *Flypeer* infrastructure itself. The transaction times for semantic search service calls are discussed for the intercontinental topology setting used.

The source code associated with this deliverable is available at `http://www.evesim.org/D3.14_src.zip`.

1 Introduction

The DE infrastructure *Flypeer*[1] represents a P2P infrastructure based on the *JXTA*[1] P2P standard[2], created by Sun Microsystems. This P2P standard defines set of XML protocols that offer P2P collaboration functionalities for any device and is independent to the underlying network infrastructure as well as firewalls or NAT. The *Flypeer* infrastructure, which uses JXTA as a base, is developed by IPTI (Instituto de Pesquisas em Tecnologia e Inovação) in Brazil and currently in version 0.8 (29/04/2010). The core of *Flypeer* contains the implementation of the transactional model[2] developed at at the University of Surrey [3]. This report describes on the one hand the implementation, deployment and transactional consummation of an example service in Chapter 2. As this guide is written for programmers, it assumes at least a basic understanding in software development.

On the other hand the report is dealing with the utilization of *Flypeer* infrastructure to test the semantic search service developed by IITK in Chapter 4. The goal was to make the service available over the P2P infrastructure and to do some performance test which should reveal the usability as well as stability of the DE infrastructure. In order to test the given properties a service which encapsulates the server-based semantic search service was created and deployed within the infrastructure. Transaction times that are generated with every service call reveal the performance as well as stability of the infrastructure. During the tests the P2P network has been spanning a continental area connecting Austria with India.

This use-case is an example for the emulation capabilities of the EvESim. The service deployed and tested via the multi-agent simulation framework by emulating the *Flypeer* infrastructure. Further examples to the emulating capabilities of EvESim will be given in Deliverable 10.20. ť

[1] `http://kenai.com/projects/flypeer/`
[2] `https://jxta.dev.java.net/`

2 Creation of a *Flypeer* Based Service

This chapter explains how a *Flypeer* service can be implemented and deployed in the P2P infrastructure.

Environment

The minimal requirements are an installed Java SE Development Kit 6 or higher, an IDE like Netbeans or Eclipse and all the libraries included in the *Flypeer* release available at `http://kenai.com/projects/flypeer/downloads`. The release also includes a seeds.properties file which includes all the public available nodes and configurations in the *Flypeer* network, displayed in 2.1.

IDE service project

In order to create a service, create a project in your chosen IDE, import all the libraries from the release as well as the seed.properties file. These required libraries displayed in Table 2.1.

As *Flypeer* is using this file to persist available online bootstrap peers during runtime, you need to so make sure it is in the root source folder. Currently this file includes three seed peers also called rendezvous peers running at SUAS, WIT and IPTI, IITK given in Listing 2.1.

```
1 seeds_1=tcp://www.opaals.at:9701
2 seeds_2=tcp://193.1.208.240:80
3 seeds_3=tcp://www.ipti.org.br:9701
4 seeds_4=tcp://202.3.77.136:9701
```

Listing 2.1: Seeds.properties

As a result of using the standard Java service loader, *Flypeer* requires meta-information to specify the services that should be loaded during runtime and deployed by the peer. In order to fulfill this prerequisite, create a folder structure '*META-INF/services/*' in the root source folder of your project. In the 'services' folder, create a file named *br.org.ipti.flypeer.services.FlypeerBusinessService*. This

axiom-api-1.2.8.jar	axiom-impl-1.2.8.jar
axis2-adb-1.5.1.jar	axis2-kernel-1.5.1.jar
axis2-transport-http-1.5.1.jar	axis2-transport-local-1.5.1.jar
bcprov-jdk14-141.jar	commons-codec-1.2.jar
commons-fileupload-1.2.jar	commons-httpclient-3.1.jar
commons-logging-1.0.4.jar	flypeer-0.8.jar
httpcore-4.0.jar	IdentityFlowJxtaModule.jar
IdentityModelSAML-1.0-SNAPSHOT.jar	jdom-1.1.jar
jetty-6.1.22.jar	jetty-util-6.1.22.jar
jxta.jar	mail-1.4.jar
neethi-2.0.4.jar	openws-1.0_alpha1.jar
OperationBuilder-1.0-SNAPSHOT.jar	OperationRequestHandler-1.0-SNAPSHOT.jar
saxon9he.jar	servlet-api-2.5-20081211.jar
smack-3.1.0.jar	swing-layout-1.0.3.jar
woden-api-1.0M8.jar	wsdl4j-1.6.2.jar
XmlSchema-1.4.3.jar	xmltooling-1.0_alpha1.jar

Table 2.1: *Flypeer* Libraries

file will persist later the class path of the services that will be deployed on that peer.

Service Implementation

To start the implementation, create a class that implements the *FlypeerBusinessService* interface[1]. This interface includes all the relevant methods for the service to be called in the *Flypeer* network and are described later. Now the class path of service should be added into the *br.org.ipti.flypeer.services.FlypeerBusinessService* to assure that it will be deployed when the peer starts. Here is an example given, how an interface implementing class can look like: *org.suas.test.TestService*. In order to be compatible with existing web standards every service has to have its own WSDL[4] file, which describes the service itself as well as input and output parameters. To assure availability during runtime place the WSDL file into the root source folder of the project. It is recommended to name the WSDL file after the service class that was created beforehand. An example of a WSDL file for the service *TestService* is given below in Listing 2.2. It shows a service with the name *TEST-SERVICE*. The service name *TEST-SERVICE* is essential, because the service is called by specifying its name. In the documentation tag in line 6, a description of the service could be added. The commented group tag in line 7

[1] `http://kenai.com/projects/flypeer/pages/UserGuide#createService`

specifies the *Flypeer* group where the service is published in. This tag is optional and if it is not there the service will be published in the *FLYPEER_DEFAULT* group. It is recommended to leave the group tag and deploy the services in the default group.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="TestService" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <wsdl:service name="TEST-SERVICE">
        <wsdl:documentation>This is a test service developed by SUAS</wsdl:
            documentation>
        <!-- <group name="test_group"> -->
    </wsdl:service>
</wsdl:definitions>
```

Listing 2.2: TestService.wsdl

Now the basic specifications of the service are shown in the WSDL file. Defining the input and output parameters are the next steps to make the service deployable. Currently *Flypeer* only supports the following Java simple types. These types are 'xs:string', 'xs:int', 'xs:short', 'xs:long', 'xs:float', 'xs:double' and 'xs:boolean'. Further development to support complex parameter types is taking place at the moment. An example for the service parameters could look like the example given below. The example given in Listing 2.3 specifies an input parameter named 'inputParameter' with the type 'xs:string' as well as an output Parameter named 'outputParameter' with the type 'xs:string'. There can be a user-defined number of input parameters but only one output parameter.

```
<wsdl:types>
  <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified"
      targetNamespace="http://ws.services.flypeer.ipti.org.br">
    <xs:element name="input">
      <xs:complexType>
        <xs:sequence>
          <xs:element minOccurs="0" name="inputParameter" nillable="true" type="xs:
              string">
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="output">
      <xs:complexType>
        <xs:sequence>
          <xs:element minOccurs="0" name="outputParameter" nillable="true" type="xs
              :string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
```

```
17   </xs:schema>
18 </wsdl:types>
```

Listing 2.3: Service Parameter Example

In addition to the parameter type definition, the WSDL file specifies the messages that send the parameters. The messages according to the given type declarations are shown in Listing 2.4.

```
1 <wsdl:message name="input">
2   <wsdl:part name="input" element="input"/>
3 </wsdl:message>
4
5 <wsdl:message name="output">
6   <wsdl:part name="output" element="output"/>
7 </wsdl:message>
```

Listing 2.4: Message Declarations

The last step to finish the WSDL file is to specify the process operation to declare what input and output the service requires illustrated in Listing 2.5. The definition for the 'portType' in line and the 'operation' tag in line is not relevant in the current *Flypeer* implementation. In the wsdl:input tag in line 3 the input parameter is specified and in wsdl:output in line 4 the output parameter.

```
1 <wsdl:portType name="TestService">
2   <wsdl:operation name="input">
3     <wsdl:input message="input"/>
4     <wsdl:output message="output"/>
5   </wsdl:operation>
6 </wsdl:portType>
```

Listing 2.5: Process Operation Example

The WSDL is now finished and the implementation of the *FlypeerBusinessService* can be started. First of all, implement the *getServiceListener* method by returning a class that implements *FlypeerServiceListener*. This class is responsible for the service call handling as it will receive the input parameter and has to return the output parameter. Whether, create a new class that implements the *FlypeerServiceListener* interface or let the service itself implement the interface. The listener has the *serviceEvent* method which handles the requests that are sent to the service. In the example the *serviceEvent* could be implemented like in Listing 2.6. Make sure that the output parameter has the specified type and that the variable in the code is named after the WSDL definition.

```
1 public Serializable serviceEvent(ServiceContext params) throws Exception {
2         String parameter = (String) params.getParameter("input");
3         System.out.println(parameter);
```

```
4         String output = new String("Service Finished");
5         return output;
6 }
```

Listing 2.6: *serviceEvent* Method Implementation

The *FlypeerBusinessService* contains some other methods which are described in the following and displayed in Listing 2.7. First, the clear method, shown in line 2, is called every time the service has finished a request. Next the *getRollbackListener* method, shown in line 6, is called when an error occurred during a service call and the transaction needs a rollback. As this functionality is not implemented yet, the method is not needed. The start method, shown in line 10, is called when a service is deployed and therefore should be used to initialize the service. The stop method, shown in line 13, will be called when a service stops but this is also not implemented yet.

```
1 @Override
2 public void clear(String arg0) {
3     System.out.println("All resources were cleared");
4 }
5 @Override
6 public FlypeerRollbackListener getRollbackListener() {
7     return null;
8 }
9 @Override
10 public void start(BasicPeer peer) {
11 }
12 @Override
13 public void stop() {
14     System.out.println("TestService was stopped");
15 }
```

Listing 2.7: *FlypeerBusinessService* Methods Implementation

To finally deploy the service create a class with a main method to create a *Flypeer* peer. This method could look like the example below. The name of the deploying peer is specified in the *AuthnData*. In the example given in Listing 2.9 *Test_Deploying_Peer* is the name of the peer. That is important to assure that the service deploying and the service calling peer do not have the same names. Starting the main method you should receive a correct initialization process in the console and read that your service has been successfully deployed like given in Listing 2.8.

```
1 INFO: Service flypeer://TEST_SERVICE has been deployed in the group FLYPEER_DEFAULT
```

Listing 2.8: Successful Service Deploy

```
1 CommandSimplePeer simple = new CommandSimplePeer();
2 AuthnData data = new AuthnData(TYPE.DEFAULT, "Test_Deploying_Peer", "pass");
```

```
3 simple.start(data, new PeerStatusNotifier() {
4   @Override
5   public void peerStarted() {
6   System.out.println("PeerStarted");
7   }
8 });
```

Listing 2.9: Service Deployer Main Method Implementation

2.1 Creation of a *Flypeer* Based Service Caller

The calling peer basically looks approximately the same as the service deploying peer. Create a new project in your IDE and import the libraries and the seeds.properties file as specified in the previous Chapter. Create a class with a main method to start your calling peer. An example for this method is shown in Listing 2.10.

```
1 CommandSimplePeer simple = new CommandSimplePeer();
2 AuthnData data = new AuthnData(TYPE.DEFAULT, "Test_Calling_Peer","pass");
3 simple.start(data, new TestStatusNotifier(simple));
```

Listing 2.10: Service Caller Main Method Implementation

The *TestStatusNotifier* is a class that implements the *PeerStatusNotifier* interface and the *FlypeerResponseListener* interface to handle the service responses. In order to communicate you have to pass the *CommandSimplePeer* object to the class. Create the class and implement the *peerStarted* method which will be called after the peer is started and in order to assure that the communication works. The *CommandSimplePeer* object is stored in the class variable peer. The example in Listing 2.11 shows an example implementation of the *peerStarted* method. The first step, to create a transaction is to create the *TransactionInitiator* object. To create the transaction parameters first create a *SequentialFlow* object because you just want to call your service once. Now specify the service you want to call by creating a service object. Make sure that the service has the correct name. Otherwise it will not be found. Add the service to the SequentialFlow. Now create the overall transaction wrapper object the *TransactionFlow*. Add a transaction result to the *TransactionFlow*, shown in line 6, that specifies the output that the service in the transaction has to deliver so that the transaction knows when successfully executed. Now set the *GenericFlow* in the *TransactionFlow* with the created *SequentialFlow*. Next create the parameters map that you want to pass to the service and add the parameter. Finally start the transaction and pass on the parameters to the *TransactionFlow*,

and the *FlypeerResponseListener* in this example it is the class itself.

```
1 TransactionInitiator init = peer.createTransactionInitiator();
2 SequentialFlow seq_flow = new SequentialFlow();
3 Service service = new Service("TEST-SERVICE");
4 seq_flow.addService(service);
5 TransactionFlow gen_flow = new TransactionFlow();
6 gen_flow.addTransactionResult("output", service);
7 gen_flow.setGenericFlow(seq_flow);
8 HashMap<String, Serializable> map = new HashMap<String, Serializable>();
9 map.put("input", "value");
10 init.startTransaction(map, gen_flow, this);
```

Listing 2.11: Transaction Initiation Code

When the transaction was executed successfully the *processResponse* method receives the results. In case of an error during the transaction, the *processFail* method will be called. An example for the *processResponse* method is given in the Listing 2.12.

```
1 public void processResponse(Map<String, Serializable> responses) {
2         System.out.println(responses.get("outputParameter"));
3 }
```

Listing 2.12: *processResponse* Method Implementation

2.2 Start the Test

In order to connect to the rendezvous peers specified in *seed.properties* you need an Internet connection. To test the service offline just start your own rendezvous peer via command line. Navigate to the folder where the *Flypeer* libraries are stored and execute the command given in Listing 2.13.

```
1 java -cp axiom-api-1.2.8.jar;axiom-impl-1.2.8.jar;axis2-adb-1.5.1.jar;axis2-kernel
    -1.5.1.jar;axis2-transport-http-1.5.1.jar;axis2-transport-local-1.5.1.jar;
    bcprov-jdk14-141.jar;commons-codec-1.2.jar;commons-fileupload-1.2.jar;commons-
    httpclient-3.1.jar;commons-logging-1.0.4.jar;flypeer-0.8.jar;httpcore-4.0.jar;
    IdentityFlowJxtaModule.jar;IdentityModelSAML-1.0-SNAPSHOT.jar;jdom-1.1.jar;
    jetty-6.1.22.jar;jetty-util-6.1.22.jar;jxta.jar;mail-1.4.jar;neethi-2.0.4.jar;
    openws-1.0\_alpha1.jar;OperationBuilder-1.0-SNAPSHOT.jar;
    OperationRequestHandler-1.0-SNAPSHOT.jar;saxon9he.jar;servlet-api-2.5-20081211.
    jar;smack-3.1.0.jar;swing-layout-1.0.3.jar;woden-api-1.0M8.jar;wsdl4j-1.6.2.jar
    ;XmlSchema-1.4.3.jar;xmltooling-1.0\_alpha1.jar;. br.org.ipti.flypeer.main.
    FlypeerMain -i -r
```

Listing 2.13: Rendezvous Peer Starting Command

Now the rendezvous peer will be started and you can start your service test by first, executing the service deploying class and second, executing the service calling

class. The transaction may need some seconds to execute. You should be able to see the transaction output shown in Listing 2.14 in the log window of your calling peer.

```
1 INFO: CLIENTCONNECT event has been received
2 19.07.2010 10:11:39 net.jxta.impl.rendezvous.edge.EdgePeerRdvService addRdv
3 INFO: Renewed RDV lease from SUAS C : 299917 / -83
4 19.07.2010 10:11:58 net.jxta.impl.pipe.NonBlockingOutputPipe <init>
5 INFO: Constructing for urn:jxta:uuid-
      A01BE8FBAEC740BDB2B8B925CABF9226EB8A434DEFE34C8FBB25CCFB446106CB04
6 19.07.2010 10:11:58 net.jxta.impl.pipe.NonBlockingOutputPipe startServiceThread
7 INFO: Thread start : Worker Thread for NonBlockingOutputPipe : urn:jxta:uuid-
      A01BE8FBAEC740BDB2B8B925CABF9226EB8A434DEFE34C8FBB25CCFB446106CB04
8   worker state : ACQUIREMESSENGER queue closed : false  number in queue : 0 number
        queued : 0 number dequeued : 0
9 19.07.2010 10:11:58 net.jxta.impl.pipe.NonBlockingOutputPipe close
10 INFO: Closing for urn:jxta:uuid-
      A01BE8FBAEC740BDB2B8B925CABF9226EB8A434DEFE34C8FBB25CCFB446106CB04
11 19.07.2010 10:11:58 net.jxta.impl.pipe.NonBlockingOutputPipe run
12 INFO: Thread exit : Worker Thread for NonBlockingOutputPipe : urn:jxta:uuid-
      A01BE8FBAEC740BDB2B8B925CABF9226EB8A434DEFE34C8FBB25CCFB446106CB04
13   worker state : CLOSED queue closed : true number in queue : 0 number queued : 1
        number dequeued : 1
14 19.07.2010 10:12:00 org.evesim.usecase.sna.service.FlypeerTestService info
15 INFO: value
```

Listing 2.14: Log Output

3 Distributed Semantic Search

The Distributed Semantic Search is an application that facilitates search over local as well as peer repositories. At its core, it uses *Lucene*[1] indexes and JXTA protocol. *Lucene* is defined as a syntactic indexer and that allows third party tools to use their results of indexation to perform semantic search. This work involves extending *Lucene* in peer-to-peer network. Each node is an independent document repository. Documents are indexed locally and searched using *Lucene*. User can specify multiple directories for indexing. This application runs on individual peers in a network and uses JXTA library to automatically discover other peers within its subnet. While searching, the application searches local index, broadcasts query to all connected peers and then collects results back. Next it merges all results and represents to user. Further descriptions are given in Deliverable 6.10.

3.1 Application Architecture

Figure 3.1 represents the architecture of the application.

Indexer: This is responsible for building index from the shared documents. It is composed of following subcomponents:

- ***Text extractor*** - uses *Lucene*-Tika package to retrieve file contents. The various file formats supported are txt, pdf, html, doc, docx, ppt, pptx, mp3, xml, odp, rtf, etc.
- ***Index Builder*** - uses *Lucene* package. It builds a reverse-index for fast searching. The indexer uses local file system to store index.

Peer Communication (Network Layer): This component facilities communication with peer nodes. When a search is performed, the search query is broadcasted to all peer nodes connected in the network and later results are collected

[1] `http://lucene.apache.org/java/docs/index.html`

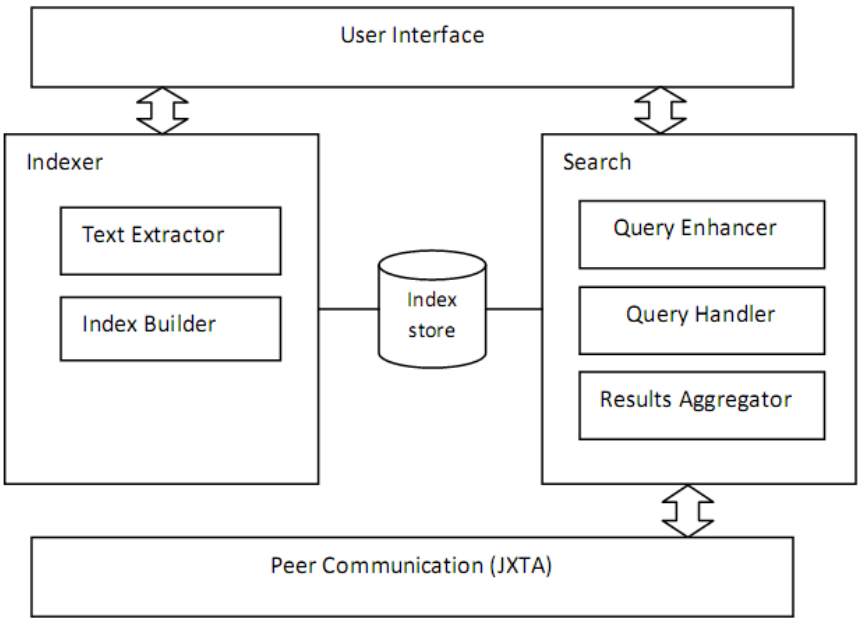

Figure 3.1: Architecture of a single node

back. This component uses JXTA peer-to-peer protocol for communicating with peers. Therefore there are easy to integrate in the Flypeer network using the same base. The JXTA protocol uses XML based messages and it is platform independent. This component is also responsible for peer discovery. Currently the application supports peers in its subnet only. To test the service via the Flypeer network a wrapper class was implemented.

Search: This component is responsible for local repository search. It interacts with local index and returns results. It is composed of following subcomponents:

- ***Query enhancer*** - enhances search query semantically. Currently this component uses Wordnet ontology to find synset (synonyms) and does query expansion.
- ***Results Aggregator*** - merges results collected from peers and produces an aggregated list. It uses match-score (returned by *Lucene*) of documents to build a merged list of results.
- ***Query Handler*** - performs search on local index. It also handles remote queries and sends results back to the requesting peer.

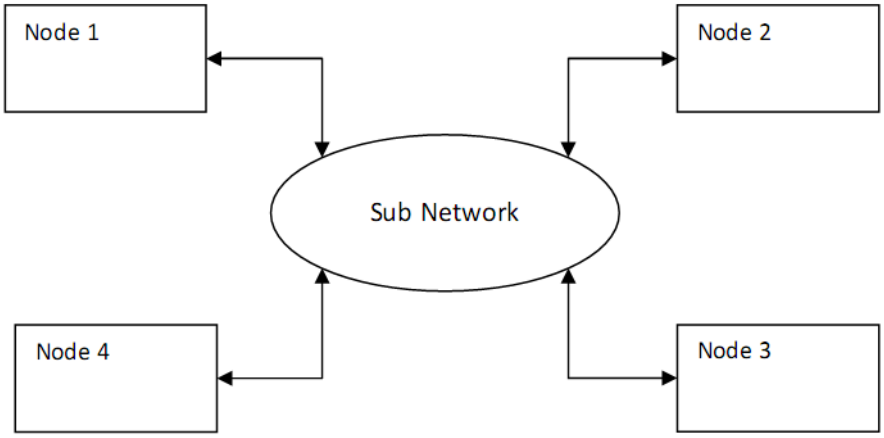

Figure 3.2: Network architecture for distributed search

User Interface: Responsible for interacting with user to perform search, displaying results and adding shared files.

3.2 Network architecture

Figure 3.2 shows network architecture for the application. Each node is autonomous with its own documents repository and search service. Nodes communicate following JXTA protocol. When search is performed on a node, it is propagated to all connected nodes in the network.

3.3 Future Enhancements

1. Intelligent query routing: Broadcasting the query to all peers suffers from scalability issue. The application must use intelligent query routing. One of the approaches is using expertise based routing where each peer initially advertises its expertise.

2. Extending the communication layer to be able to connect to peers outside of sub-network.

4 Semantic Search Utilization with *Flypeer* Infrastructure

4.1 Setting and Scope

The ongoing development on the *Flypeer* infrastructure requires as much testing as possible in order to assure high quality feedback and change suggestions. This chapter describes the combination of the infrastructure investigation and the utilization of the semantic search employment provided by IITK. This particular service offers the ability to search for the semantic constructs in the OPAALS file library. Originally, this service is server-based and can be reached via an HTTP call including the semantics to search for. The computed results are encoded as an RSS XML file and provide complete information about the, according to the search parameters, discovered documents. During the test preparation, a *Flypeer* proxy service was created in order to bridge the interspace between the server-based service and the *Flypeer* infrastructure. Besides the infrastructure testing purposes, this bridging makes the semantic search service available for every user of the *Flypeer* network. It was only possible to reach the service via HTTP connection and not within the *Flypeer* network. Furthermore, it is the first infrastructure test that spans the testing network over a continental area, linking Austria with India. The test itself intends to validate the availability and performance that are supplied by the *Flypeer* infrastructure working over continental distances. Mainly the times measured between the start of a service call and the reception of results is used. The setting and conception of the test is outlined in the following.

4.2 Test Case Setting

In order to include another OPAALS component besides the *Flypeer* infrastructure and the Semantic Search Service, the EvESim multi-agent framework has been used to concept, implement and execute the test case. The framework provides generic timer-based software agents which can be adapted to the user's needs. Furthermore, it offers a simulation environment that enables the user to visualize the simulation network and to collect the results computed by the simulation. The test scenario is based on a collection of agents which are calling the semantic search service over the *Flypeer* infrastructure. They use different search strings and measure the time from the moment of starting the transaction until the reception of the results. The agents work autonomously and send the processed results to the framework control. To go a bit into detail, every agent is communicating with the semantic search service using the transactional model implemented within the *Flypeer* infrastructure. A proxy agent is used to provide communication facilities to all underlying agents. This is necessary because only one agent instance per running machine is supported by *Flypeer*. Normally the simulating agents are distributed over a network, but for this particular test this distribution has been no key factor and therefore, was not included. The outcome of this is that the agents have been executed on the same physical machine using the described gateway agent. The communication is accomplished via a transaction prepared and executed in the *Flypeer* network. It uses parameters which are on the one hand a simply query string that is passed from the agent to the service and on the other hand a XML RSS string returned by the service. To get results and measure the time needed, the simulation introduces timestamps which are taken when the transaction is initiated. When a result is detected, the according timestamp is taken and the elapsed time is calculated by subtraction. The results are cumulated to give an impression on the characteristics of the transaction behavior. These results are illustrated in the following

4.3 Test Results

After several simulation executions the results were cumulated and are shown in 4.1. It illustrates the time periods elapsed during transactions executed by the simulating agents. In a whole 300 transactions time periods are displayed in the figure but in order to generate this time data about 350 transactions were conducted. The particular number of not-committed transactions during a simulation is about 15

percent. As shown, about half of the transactions have been executed within a time period of 20 seconds. The approximately next 50 transaction periods are clearly higher and fluctuate up to 60 seconds per transaction. The last 100 transactions show another staircase-shaped increase up to 130 seconds.

The results show that the transaction times are widely spread from below 5 up to 130 seconds. To explain, the intercontinental area spanned within these simulations and the various network boundaries and firewalls, which have to be bridged, are the main factors for this wide spectrum of transaction times. Reasons for the occurring exceptional long transaction times are much more difficult to be explained. The following observations were taken during the simulations. The infrastructure tends to have a starting threshold, meaning the first transaction finishes always after considerable longer time period as the following ones. Also remarkable is that the transactions following are fluctuating around a time level that is specified in the first few ones. It is more likely that the transaction periods circulate around the level that was instantiated by the first few transactions than having time periods that differ very much from that level. Concrete reasons for these phenomena had not been found yet. To conclude, the infrastructure is functional bridging a continental area and provides service access over the *Flypeer* network. The performance of the infrastructure has been proven to be not adequate for industrial use but as the development is ongoing it will be changed in future.

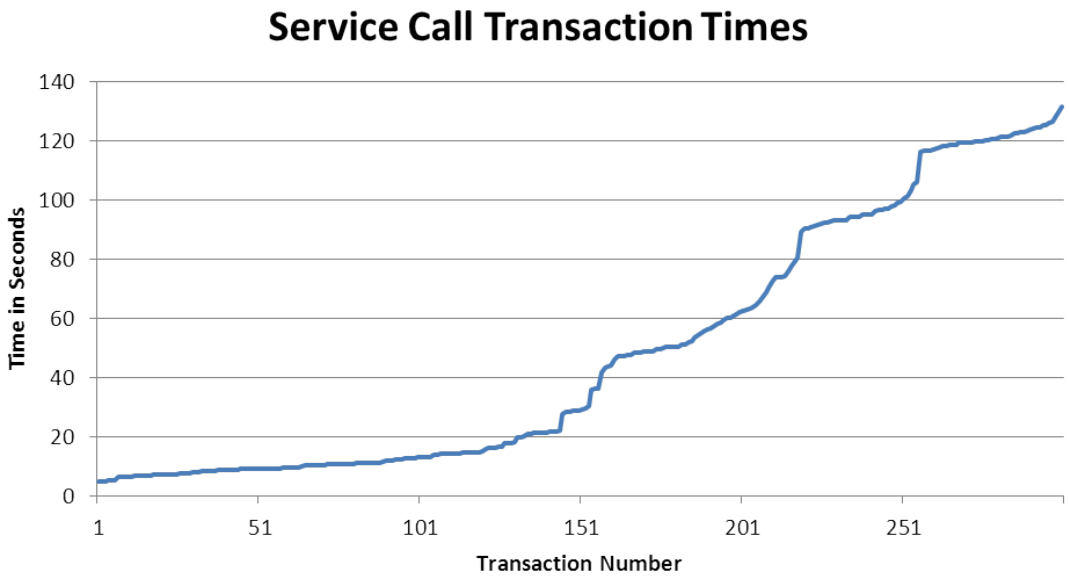

Figure 4.1: Test Case Results

5 Conclusion

As described in Chapter 4 the performance and reliability of the *Flypeer* infrastructure is not yet ready for industrial usage but there has been several improvements when combining it with previous versions. The services deployed can be reached via web service calls and therefore the compatibility with already existing company service infrastructures can be more easily achieved. In addition, in the test case described above, the infrastructure bridged a transcontinental gap bypassing several firewalls and network boundaries. Furthermore, there had been improvements concerning the runtime stability of the infrastructure and the usage complexity meaning it has been made easier to implement a service. Summing up, the semantic search scenario, as a transcontinental test emphasizes the bridging and connecting qualities of the *Flypeer* infrastructure as well as shows some weak spots concerning stability and performance. The source code of the use-case is provided at `http://www.evesim.org/D3.14_src.zip`.

Causes for the performance limitations within the infrastructure mentioned in the report, may be based on the complex and extensive structure that the JXTA and consequently the *Flypeer* architecture is based on. As JXTA is aimed on generic principles it does not aim on performance in the first place. The transactional model that is implemented within *Flypeer* append additional overhead and complexity to the infrastructure. Although the infrastructure has accomplished providing a running platform for transcontinental service usage and deployment.

Bibliography

[1] JXTA Community. JXTA. `http://jxta.dev.java.net`, Last Accessed: 29/07/2010.

[2] S. Moschoyiannis, A. Razavi, Y. Zheng, and P. Krause. Long-Running Transactions: semantics, schemas, implementation. In *IEEE Digital Ecosystems and Technologies (DEST 2008)*. IEEE Computer Society, 2008.

[3] A. Razavi and P. Krause. D3.10 - integrated autopoietic de architecture. OPAALS Project, September 2009.

[4] R. Chinnici, J. Moreau, C. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Technical report, W3C, 2007. Last accessed: 27/07/2010.