TOTEM Toolbox
Integration of new algorithms

Project Title :	**TOTEM : TOolbox for Traffic Engineering Methods**
Contributor :	**Simon Balon, Olivier Delcourt, Jean Lepropre, Fabian Skivée, Gaël Monfort, Guy Leduc, Bruno Quoitin (UCL)**
Abstract :	**-**

Contents

1 Introduction

This document explains how to integrate a new algorithm into the toolbox. Being able to easily integrate a new algorithm was one of the main requirements of the toolbox architecture.

2 Java

All algorithms integrated in the toolbox implements the interface `TotemAlgorithm`. There are three main interfaces used by the routing algorithms in particular, i.e. `SPF`, `LSPPrimaryRouting` and `LSPBackupRouting`. If the algorithm you plan to integrate does not map to one of these interfaces, the first thing to do will be to define a new one. It should be quite easy just by looking at the ones we already defined. You should also try to manage the parameters as we did.

From then, if your algorithm is written in Java, you are ready to integrate it by developing the methods present in the interface. If your algorithm is written in C or C++, read the next section. If your algorithm is written in Perl, you might want to look at JPL[1]. But we did not investigate JPL. If your algorithm is written in another language, it would still be possible to interoperate with the toolbox using our XML format.

3 C or C++ code

Java Native Interface (JNI)[2] is the key to the integration of algorithms written in C or C++ into the toolbox written in Java.

JNI allows Java code that runs within a Java Virtual Machine to operate with libraries written in other languages such as C and C++. The JNI framework lets native methods utilize Java objects in

[1] See `http://press.oreilly.com/pub/pr/698`
[2] `http://java.sun.com/docs/books/tutorial/native1.1/`

the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them. Native methods can also easily call Java methods. With JNI, we can use the advantages of the Java programming language from the native method. In particular, we can catch and throw exceptions from the native method and have these exceptions handled in the Java application.

All the code examples and commands are given for C code. We will explain how, for example, to integrate a CSPF routing algorithm written in C into the toolbox. All packages names are shortened (and so are class names, method names,... accordingly). You should systematically add the root `be.ac.ulg.montefiore.run.totem.` in front of them.

4 Overview (The CSPF Example)

Here are the steps that must be followed :

1. In the `repository` package, create a new package called `CSPF`, and inside it a new class of the same name. This class will implement `repository.model.LSPPrimaryRouting` and thus the corresponding 4 methods.
2. Create a series of native methods that you intend to call from the four preceeding Java methods, and add the code lines necessary to load the library created in step 5.
3. Recompile the toolbox using ANT. Generate a header file containing the native methods prototypes(formal signature for the native methods).
4. Write the implementation of the native methods in C as part of your CSPF C implementation.
5. Recompile your native CSPF implementation as a shared library file.

We will now detail these five steps.

4.1 New CSPF package and class

So, the code of the new `CSPF` class in the `repository.CSPF` package should look like this.

```
package repository.CSPF;
import repository.model.*;

public class CSPF implements LSPPrimaryRouting{
    public TotemActionList routeLSP(LSPPrimaryRoutingParameter param)
       throws RoutingException, NoRouteToHostException {
        return null;
    }

    public TotemActionList routeNLSP(List param)
       throws RoutingException, NoRouteToHostException {
        return null;
    }

    public void start() {
```

```
    }

    public void stop() {

    }
}
```

4.2 Create the native methods

The native CSPF implementation needs the topology nodes and links information. So, we will explain how to pass the nodes from Java database to the CSPF database. We need to define a native method `jniAddNode(nodeId)`. In the `start` method, we call this native method. We must also not forget to load the C library. We decide to call it `cspf`. This will correspond to the filename `libcspf.so`.

We suppose that the CSPF implementation only uses integers as node identifiers. So, we have written a method that gives an integer representation from a `Node` object, this method is called `getCorrespondingInt`. From there, we call the native method `jniAddNode` and we catch exceptions that could be thrown by this native method (see section 5.3).

```
package be.ac.ulg.montefiore.run.totem.repository.CSPF;

import be.ac.ulg.montefiore.run.totem.repository.model.*;
import be.ac.ulg.montefiore.run.totem.topology.model.Topology;
import be.ac.ulg.montefiore.run.totem.topology.model.Node;
import be.ac.ulg.montefiore.run.totem.topology.facade.TopologyManager;

public class CSPF implements LSPPrimaryRouting{

    private native static void jniAddNode(int nodeId);

    static {
            System.loadLibrary("cspf");
    }

    public void start() {
       Topology topo = TopologyManager.getInstance().getTopology();

        int nbNodes = topo.getNodes().getNode().size();

        for (int i=0;i<nbNodes;i++){
            Node node = (Node)topo.getNodes().getNode().get(i);

            try{
                jniAddNode(node.getCorrespondingInt);
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
    }
    ...
}
```

4.3 Generate the C header file

To generate the C header file that comprises all native methods declarations, first recompile the toolbox using ANT.

Then, proceed as follows: Add `build/classes/` to your classpath (on linux, just go to that directory and type `export CLASSPATH=$CLASSPATH:.`) From there, run the command `javah -jni PACKAGE.repository.CSPF.CSPF` where `PACKAGE` stands for `be.ac.ulg.montefio` This will generate the header file `be_ac_ulg_montefiore_run_totem_repository_CSPF_CSPF.h` in `build/classes/` which looks like this

```
#include <jni.h>

 * Class:     be_ac_ulg_montefiore_run_totem_repository_CSPF_CSPF
 * Method:    jniAddNode
 * Signature: (I)V
 */
JNIEXPORT void JNICALL
  Java_be_ac_ulg_montefiore_run_totem_repository_CSPF_CSPF_jniAddNode
  (JNIEnv *, jclass, jint);
```

This header file provides a C function signature for the implementation of the native method `jniAddNode` defined in the `CSPF` class. Notice that the method that we will have to implement in C has a more complex name and different arguments. The third argument, of `jint` type, corresponds to the argument that `jniAddNode` had in its Java definition. We will see that this `jint` type can be used exactly as C integer. But generally, it's not the case, i.e. arguments must be accessed through C functions found in the JNI library `jni.h`. The other two parameters are required for every native method. The first parameter is a `JNIEnv` interface pointer. It is through this pointer that the native code accesses parameters and objects passed to it from the Java application. The second parameter is `jclass`, which references the method's Java class [3].

4.4 Writing the native method implementation

We will now write the implemention of the native method. We will suppose that the existing CSPF implementation already has some kind of interface (see section 6) functions. For example, a function like `addNodeTo DB(DataBase,nodeId)` is already accessible from your C code. Create a new file called `jniCSPFInterfa ce.c`. Firstly, include the JNI library and the header file previously created. You should probably also include the header file that contains the CSPF implementation interface functions (like `addNodeToDB`). Just fill in the implementation of the native method. `jniCSPFInterface.c` should look like this

```
#include "be_ac_ulg_montefiore_run_totem_repository_CSPF_CSPF.h"
#include <jni.h> // the JNI library
#include cspf_api.h // interface functions declarations

/*
 * Class:     be_ac_ulg_montefiore_run_totem_repository_CSPF_CSPF
 * Method:    jniAddNode
 * Signature: (I)V
 */

JNIEXPORT void JNICALL
Java_be_ac_ulg_montefiore_run_totem_repository_CSPF_CSPF_jniAddNode
```

[3] Note that if we had not declared jniAddNode as a static method, this second parameter would have been a reference to the current instance of the object

```
(JNIEnv *env, jclass class, jint nodeId)
{
  addNodeToDB(DataBase, nodeId);
}
```

The pointer to the database must be made available to the native function by another native function called before.

4.5 Compiling the CSPF C implementation as a C library

Now that "everything" is done, let's generate the library file. Here is the command line to compile the library in C.

```
ld -Bshareable -fPIC -o libcspf.so jniCSPFInterface.o OTHER_OBJ_FILES -lz -l
```

Where jniCSPFInterface.o has been compiled with `-I\$JAVA_HOME/include/ -I\$JAVA_HOME/include/linux/` (replace `\$JAVA_HOME` by the corresponding value).

Now, you just have to call the `start()` method of your CSPF algorithm somewhere in the toolbox code (:-)) and you're ready.

5 Going into more details

So, after this brief example, we'll detail some more things that we think are interesting. The objective of this documentation is not to replace existing JNI documentation.

5.1 Java types and C types

In the simple CSPF example, we only had a basic integer argument `NodeId` which the C environment gets as a new `jint` type. This `jint` can basically be used as C `int` type.

Basically, all Java primitive types can be accessed directly by the native method in C [4].

But non-primitive Java types can never be accessed directly but well through functions defined in the JNI C library (`jni.h`) [5].

For example, suppose we declare in Java a new float array, and then pass it to a native method as argument

```
float[] floatArray = new float[SIZE];
//... store information in this array
jnimethod(floatArray);
```

We will then get this array as a `jfloatArray` type element. To access information stored in this array, we will use:

```
jsize len = (*env)->GetArrayLength(env,PACKAGE_floatArray); // get the length
of the array
jfloat *body = (*env)->GetFloatArrayElements(env, PACKAGE_floatArray, 0); //
env is the environment pointer

for (int j=0;j<len;j++){
  printf(''Content at position \%d: \%f\n'',j,body[j]);
```

[4] See http://java.sun.com/docs/books/tutorial/native1.1/integrating/types.html

[5] See http://java.sun.com/docs/books/tutorial/native1.1/implementing/string.html and http://java.sun.com/docs/books/tutorial/native1.1/implementing/array.html

```
}
(*env)->ReleaseFloatArrayElements(env, PACKAGE_floatArray, body, 0);
```

We invite you to look at the code of `jni_interface.c` in `src/C/jniDAMOTE/jni` which shows interesting examples on how we manage to access the information stored in multi-dimensional arrays.

5.2 Returning information and accessing Java member variables

`jni_interface.c` code also shows how we manage to create new arrays (and arrays of arrays) in the native world that are then returned to the Java environment.

When we had to return other information, we used the possibility offered by JNI to access Java member variables. That is, we define in the calling Java class some variables that we then access from the native world using the environment pointer and modify from the native world. The modified value is then available in the Java environment[6]. You will also find interesting examples in `jni_interface.c`.

5.3 Throwing Exceptions from native world

When an error occurs in the native code, it's interesting to be able to throw an exception from the native code that is then caught in the Java environment.

Let's see what our native method implementation would become with exception handling.

```
JNIEXPORT void JNICALL
Java_be_ac_ulg_montefiore_run_totem_repository_CSPF_CSPF_jniAddNode
(JNIEnv *env, jclass class, jint nodeId)
{
  if (addNodeToDB(DataBase, nodeId) < 0) // error occured
  {
    jclass newExcCls;

    // find the class corresponding to RoutingException in the Java environment
    newExcCls =
    (*env)->FindClass(env,"be/ac/ulg/montefiore/run/totem/repository/model
/RoutingException");

    if (newExcCls == NULL){
      fprintf(stderr,"Unable to find the exception class, giving up\n");
      return; // come back to Java
    }

    (*env)->ThrowNew(env,newExcCls,NULL);
    return;
  }
}
```

5.4 Callbacks

Sometimes, if you have no other choice, you may want to call Java methods from C native code[7]. This is feasible, but slow (and we met some bugs when playing with call-backs).

[6] See http://java.sun.com/docs/books/tutorial/native1.1/implementing/field.html
[7] See http://java.sun.com/docs/books/tutorial/native1.1/implementing/method.html

6 Recommendations

The integration of a native algorithm should be facilitated by the definition of a clear interface. Let's take the concrete case of our CSPF algorithm. This CSPF algorithm computes routes based on the topology information it finds in its proprietary database. This database must be filled at one moment. If it is filled using well defined "interface" functions like `addNode`, `addLink`,... rather than by directly accessing the database structure, it will facilitate JNI integration. Because, in the `start()` method, we will only have to "call"[8] these methods.

[8] i.e. to define a native method like jniAddNode as we did in our example, the corresponding method implementation being only a call to the C function addNode