GaianDB Opinion Manager

No recommendations | Updated today at 9:25 PM by David A. Wood | Tags: *None*

This tutorial uses an IRuleEngine and a rules file that accepts, maintains and manipulates 'opinions' about neighboring nodes in a network of GaianDB nodes. In this tutorial you will learn how to

1. Create and use an instance of an IRuleEngine in Java
2. Send data into the rule engine for incorporation into local state
3. Extract state information from that maintained by the rules

The GaianDB is a distributed and federated database in which database queries are flooded into its network. Upon response to the query, the intermediate nodes may have a choice about which neighbor node is used to carry the response. It is this decision which we would like to augment with the use of neighbor opinion information. So for example, if two or more neighbors are available to process the response, the forwarding node may select the neighbor with the highest opinion. We would like each node to maintain its own local opinions about its neighbor nodes. The inputs to our opinion system will be a) a node's direct observations of its neighbors and b) shared neighbor opinions or observations of neighbor nodes.
The primary tasks, in the order we will approach them are as follows:

1. Define the database schema that will maintain the opinion information.
2. Implement a GaianOpinionManager Java class using the database schemas and which will allow us to register observations and query the current state of the opinions.
3. Write a set of rules to manipulate opinions based on local observations and shared neighbor opinions or observations.

First, a word about application name spaces, application (or node) identifiers and IApplicationDescriptors and how they will relate to the GaianDB. The ISharedDataPlatform uses IApplicationDescriptors to assign both a application-wide name space and a unique application identifier within the name space. The name space is shared across a set of applications and all applications using the same name space are considered to be part of the same distributed application. The application identifier uniquely identifies a shared platform instance with the name space. We will make extensive use of the application (or node) identifier in what follows to associate opinions, observations and within the shared platform to address and route messages. The GaianDB will define its own values for these application/node ids, and these will in general be the name the GaianDB has assigned to its own local instance.

Database Schema

The database schema needs to hold a single opinion score for each neighboring node. The following schema for a table we'll name *my_opinions:*

Node - String	Opinion - double value

Note that in the rule system this will be defined with the following statement:

```
persistent my_opinions(char[128] node, double opinion);
```

We also need to define the schema for the local observations that we will input into the rules. Again a simple schema for table we'll name *my_observations*:

Node - String	Observation - double value

In the rule system this will be defined as follows:

```
input my_observations(char[128] node, double opinion);
```

GaianOpinionManager

The GaianOpinionManager is a service provided to the GaianDB, which will instantiate one instance per GaianDB node and them make calls to it to a) apply local observations and b) acquire the collaboratively developed opinions about its neighbors. What it does with this information is not part of the GaianOpinionManager implementations. The GaianOpinionManager will use an instance of DSMEngine to allow rule-based manipulation of the opinions - again, based on local observations and shared neighbor opinions or observations. As such it will require a rules file to be provided. These are the primary methods to define:

1. GaianOpinionManager constructor - will create the DSMEngine with the given rules file. Also important is the node id (this is analogous to the instance id of an IApplicationDescriptor).
2. A method to apply local observations
3. A method to query the opinions
4. start and stop method to enable the starting and stopping of the DSMEngine instance.

Note that we did not define a method to send opinions to the neighbors. That function is part of the DSMEngine functionality, with the definition of which information to share with neighbors included in the rules.

The GaianOpinionManager has a class definition that begins as follows

```
public class GaianOpinionManager {
   final public String AppNameSpace = "GaianOpinionManager"
    IRuleEngine dsmEngine;
    ....
```

It declares AppNameSpace as a constant holding the application name space that all applications participating in this application must use. It will be used in the constructor when creating the instance of the rule engine. It then declares dsmEngine which is the IRuleEngine that will be used to processing the opinions and observations using the rules file provided to our constructor, which we look at next.

Constructor

The constructor must create the DSMEngine instance which is fairly straightforward given a rules file. However we would like to restrict the ISharedDataPlatform's topology information to be neighbor-only to better fit with the no-global information paradigm of GaianDB. This is sufficient anyway since we only ever need to share information locally.

```
   public GaianOpinionManager(String nodeID, String rulesFile) throws IOException,
DSMException {

        /** Create a multicast topology monitor that only provides
information about neighboring applications only */
        McastTopologyMonitor mnm = new McastTopologyMonitor(true);

        /** Create the DSM rules engine using the topology monitor,
rules file.  The application descriptor is
         * created so that all instances across the network are in the
same application name space.  The instance id
         * must be unique and is provided by the GaianDB.
         */
        dsmEngine = new DSMEngine(new
ApplicationDescriptor(AppNameSpace , nodeID),new
UnstructuredTopology(), mnm, rulesFile);
     }
```

You may have noticed that the DSMEngine is not an ISharedDataPlatform and instead is an IRuleEngine. The DSMEngine uses an ISharedTuplePlatform to share tuples between engines. If you need the services of the underlying platform you may extend DSMEngine to get access to the platform instance.

setObservation

The setObservation() method must accept an observation score (in the range -1..1) for a node identified by its node identifier. The observation is sent into the engine as a tuple set within the *my_observations* table. This method is not concerned with what happens to the data or how opinions are derived. It only needs to place the observation in the rules engine.

```
   public void setObservation(String nodeID, double observationOpinion) {
      if (observationOpinion < -1 || observationOpinion > 1)
         throw new RuntimeException("opinion value " + observationOpinion + " is not in
range");

      try {
          // Create a row/tuple that will be inserted into the observations table
         ITuple row = new Tuple(nodeID, observationOpinion);
```

```
            // Add the row/tuple to the observations table and trigger a rule evaluation
            dsmEngine.addTuples("my_observations" row);

        } catch (DSMException e) {
            e.printStackTrace();
        }
    }
```

This will trigger an evaluation of the rules and possibly modify opinions - the effect of this observation is defined within the rules, not this class.

getOpinions

The getOpinions() method will return a map of node identifiers mapped to opinion scores. The node identifier values will come from the values passed into the constructors at the various GaianDB node instances. The opinion will be a Double object holding the opinion as developed by application of the rules to the local observations and shared opinions or observations of neighboring nodes. This method is not concerned with how the opinions are derived, only how to get them out of the DSMEngine and provide them back as return values.

```
    public Map<String, Double> getOpinions() {
        Map<String, Double> retMap = new HashMap<String, Double>();
        try {
            // Get the opinions from the opinions table into an ITupleSet
            ITupleSet opinions = dsmEngine.getTuples("my_opinions");

            // Return this in the format requested by GaianDB.
            for (ITuple t : opinions) {
                // Get the entry for the 1st column - the node id.
                ITupleEntry nodeEntry = t.get(0);
                // Get the string value from the entry.  This will be the nodeID as provided to this
class's constructor.
                String nodeID = nodeEntry.getValue();

                // Get the entry for the 2nd column - the opinion
                ITupleEntry opinionEntry = t.get(1);
                double opinion = Double.parseDouble(opinionEntry.getValue());

                // Put the data into the map as expected.
                retMap.put(nodeID, new Double(opinion));
            }
        } catch (DSMException e) {
            e.printStackTrace();
            return null;
        }

        return retMap;
    }
```

start/stop

The DSMEngine instance must be started using it's start() method. Similarly it must be stopped when no longer needed. The easiest way to make this happen is to add start() and stop() methods to GaianOpinionManager and require users to call these methods.   It is also useful to have the start() method validate the rules file to be sure it has defined the tables (my_opinions, my_observations) that are used by our setObservations() and getOpinions() methods.

```
    public void start() throws DSMException {
        dsmEngine.start();

        // Make sure the rules file defined the tables that we expect to use.
        validateTable("my_observations", TupleEntryType.String, TupleEntryType.Double);
        validateTable("my_opinions",    TupleEntryType.String, TupleEntryType.Double);
    }
```

The validateTable() method takes a table name and list of column types and makes sure that the schema inside the database of the engine contains the specified table. This demonstrates how to access the engines tuple storage and manipulate the tuple-related data structures.

```
    private void validateTable(String tableName, TupleEntryType...colTypes) throws
DSMException {
        // Get the tuple storage (db) that the engine is using
        IReadOnlyTupleStorage storage = dsmEngine.getReadOnlyTupleStorage();

        // See if the requested table is present in the storage
        ITupleSetDescriptor desc =
storage.getDescriptor(dsmEngine.getApplicationDescriptor(), tableName);
        if (desc == null)
            throw new DSMException("Rules file does not contain table " + tableName);

        // Get the list of columns for this table
        List<IColumnDescriptor> cols = desc.getColumns();

        // Make sure the number of columns matches
        if (cols.size() != colTypes.length)
            throw new DSMException("Expected " + colTypes.length + " columns in table " +
tableName);
```

```
        // Look at each column, and make sure it is of the expected type.
        for (int i=0 ; i<colTypes.length ; i++) {
            IColumnDescriptor cd  = cols.get(i);
            if (cd.getType().isAssignableFrom(colTypes[i]))
                throw new DSMException("Column " + cd.getName() + " in table " + tableName
+ " is not type compatible with " + colTypes[i]);
        }
    }
```

The last method in GaianOpinionManager is stop() which simply stops the DSMEngine instance.

```
    public void stop() throws DSMException {
        dsmEngine.stop();
    }
```

Rules File

This needs more explanation, but in the mean time, here are the rules...

```
system peers(char[128] node, char[64] relationship);
input my_observations(char[128] node, double opinion);
transient new_opinions(char[128] source, char[128] node, double
opinion);
persistent my_opinions(char[128] node, double opinion);
persistent individual_opinions( char[128] source, char[128] node,
double opinion);
transport shared_observations(char[128] node, double opinion);


// My locally determined observation trumps others
new_opinions("",n,o) := my_observations(n,o);

// If i have an opinion of the sending node, then only accept it if my
opinion of them meets a threshold.
new_opinions(src,n,o) := shared_observations(n,o)@src,
my_opinions(src,src_opinion) : src_opinion > .5;

// Accept other opinions if I don't have one.
new_opinions(src,n,o) := shared_observations(n,o)@src, not
my_opinions(src,*);

// Clear my opinion table.
my_opinions(n, o) -= my_opinions(n,o);
```

```
// Replace existing opinions with new ones.
individual_opinions(src,n,o) -= new_opinions(src,n,o),
individual_opinions(src,n,*);
individual_opinions(src,n,o) += new_opinions(src,n,o);

block;
new_opinions(src, n, o) := individual_opinions(src, n, o), not
new_opinions(src, n, *);
my_opinions(n, avg(o)) += new_opinions(src, n, o), not peers(n,
"Self");

// Send my observations to my neighbor nodes.
shared_observations(n,o)@neighbor += my_observations(n,o),
peers(neighbor, "Neighbor");
```

Comments

There are no comments.