Getting Started with Vivaldi project

Nicolas Joseph, Raphaël Gaschignard
Guillaume Blondeau, Cyprien Quilici, Jacob Tardieu

21 février 2014

1 Overview

This document is meant as a "getting started" guide for the `VivaldiGSI` project, which is an implementation of the Vivaldi algorithm in Scala (hosted on Github https://github.com/BeyondTheClouds/VivaldiGSI). For more specific information on architecture decisions , check the project's `README`.

This project uses `sbt` as its dependency management and testing system. To generate the intial project files for IntelliJ (our recommended IDE for this project), simply run `sbt gen-idea` in the project root.

2 Architecture

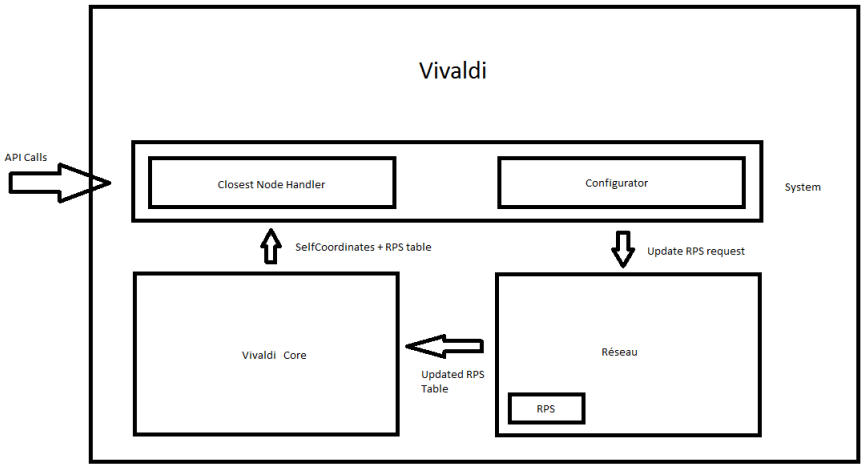

FIGURE 1 – Architecture of the Vivaldi projectl

The project is built on top of the message-passing actor model, and the API itself (described in the project's `README`) works through message passing

(via Akka's `ask` patter) . Here is an overview of the three actors that make our system.

- `VivaldiActor` : This actor is the main overlay for our program, and is the one to which API calls should be sent to. This actor sets up the other actors, sets up timers for which to periodically contact other nodes in our network, and update our information. It contains an array of the nodes which are thought to be closest to this actor.
- `Communication` : This actor maintains a random set of other nodes in our network (called the RPS). This actor, after receiving a signal from the `VivaldiActor`, will contact a random subset of nodes from the RPS to generate a new random set (by "pinging" the nodes and receiving their RPSs, and current positions). After receiving this information, it is forwarded to the `ComputingAlgorithm` for it to apply the Vivaldi algorithm.
- `ComputingAlgorithm` : This node, after receiving information concerning the position of other nodes in the system, recalculates the current node's position, then sends it to `VivaldiActor` to update its table of the closest nodes in the system.

3 Setting up a network

Initializing a node is as simple as creating a new `VivaldiActor`

```
1  val aNode= new VivaldiActor("NodeOne", 1)
```

The node takes a human-readable name and a unique id (used for joining Ping results and used when trying to decide on the equality of actors).

To connect to an existing network, send the `FirstContact` message to your node, with an `ActorRef` to some other node in your decentralized network.

If you are trying to create a new network, you have two choices :

– Make the node connect to itself

```
1  aNode ! firstContact(aNode.self)
```

– Create the first two nodes and connect them to each other

```
1  aNode ! firstContact(anotherNodeRef)
2  //in some other program running elsewhere
3  anotherNode ! firstContact(aNodeRef)
```

In any case, you need to send a `firstContact` message to all nodes, because this is what triggers the schedulers that run all the update code : without this, while nodes can receive information, they will not broadcast their existence in the network.

4 Defining another distance metric

By default the metric used for defining distance between two nodes in the overlay is the time it takes for a `Ping` message to make a round-trip on the network. However, sometimes, we want to modify this metric according to other parameters in the system (such as the bandwidth available in a system).

Doing this is relatively easy in our system. One just needs to inherit our `Communications` actor and override the `CalculatePing` function.

As an example, for our tests running on just one machine, instead of using the round-trip time, we use a predefined map to determine what value to use for each pair of machines (to be able to manually define a topology).

```
1  class FakePing(id:Long, core:ActorRef, main:ActorRef)
       extends Communication(id, core, main) {
2
3    /**
4     * Gives the ping from the ping table created in the
          function createTable
5     * @param sendTime
6     * @param otherInfo
7     * @return
8     */
9     override def calculatePing(sendTime:Long,otherInfo:
          RPSInfo):Long={
10
11      var ping = FakePing.pingTable(this.id.toInt)(
            otherInfo.id.toInt)
12      ping += Random.nextDouble()/10*Math.pow(-1, Random.
            nextInt(10))*ping
13      ping.toLong
14    }
15 }
```

In order to actually run this, we also needed to override our `VivaldiActor` class to use this derived communication class

```
1  class FakeMain(name : String, id : Long) extends
       VivaldiActor(name, id) {
2
3    override val vivaldiCore = context.actorOf(Props(
         classOf[ComputingAlgorithm], self, deltaConf), "
         VivaldiCore"+id)
4    override val network = context.actorOf(Props(classOf[
         FakePing], id, vivaldiCore, self), "Network"+id)
```

```
5
6 }
```

(All of this can be found in `FakePing.scala`, in our project files)

From there, we can initialize instances of `FakeMain`, and they will use the updated metrics.

A simple way to build this is to do the following :

- Build a service that lets you calculate your metric based off of the ids of nodes
- Override the `Communication` class with a `calculatePing` that calls this service
- Override the `VivaldiActor` class to use this updated `Communication` class.

5 Monitoring

We also built a monitoring application alongside this project. The aim of the monitoring app is to aggregate data from all the distributed nodes to make sure that the system behaves properly. The monitoring application can be found at https://github.com/callicles/VivaldiMonitoringPlay.

It is very easy to use and deploy. A deploy script is present in the `PythonDummyDataGenerator` folder.

The following section shows how to configure the application to have the monitoring system activated.

6 Configuration

Various values, such as the log level and links toward the monitoring system for the project are initialized via a setting file in the project. You can find all these settings in the `application.conf` of the Vivaldi project.
Here is an example of the configuration to use when you want to see the monitoring :

```
# configuration file for the project

akka {

    loggers = ["akka.event.slf4j.Slf4jLogger"]

    # Options: OFF, ERROR, WARNING, INFO, DEBUG
```

```
    loglevel = "DEBUG"


}

vivaldi {

    system {

        # Here is the monitoring settings for Vivaldi
        monitoring {
        #Activates the monitoring
            activated = true
        #Url of the monitoring app server
            url = "http://vivaldi-monitoring-demo.herokuapp.com"
        #network name to post data on. You can choose
        #whatever name you would like as long as there is a
        #corresponding network created in the monitoring app.
            network = "Vivaldi"
        }

        init {

            firstCallTime = 0
            timeBetweenCallsFirst = 1
            timeBetweenCallsThen = 2
            numberOfNodesCalled = 10
            changeTime = 6

        }

        closeNodes {

           size = 100

        }

        vivaldi {

            delta = 0.5

        }
```

```
        communication {

            rpssize = 100

        }

    }

}
```

As far as the monitoring is concerned, as soon as it is deployed, you just have to create the network you want to use. To do so, you can either post a request on the application API or create it manually from the interface. There is a complete API documentation in the README of the GitHub Project.

7 Where to go from here

Here are a couple things we noted that we would have loved to setup, but didn't have the time to do :

- *Create a mechanism to determine configurations*
  A big issue we have currently is that some of the configuration values used for the Vivaldi algorithm are not necessarily optimized for most use cases. Building an automated testing mechanism that could try to find good results (or that could make these parameters vary dynamically) would make the system give better results
- *Use Dependency Injection*
  Some aspects of the system(notably the metrics definitions) should be highly configurable, to suit the needs of different users. Currently, the way to do that is through sub-classing, but (as seen in the previous section) that can quickly become unwieldy. Setting up proper dependency injection would solve this.
- *Create better automated testing tools*
  We have some tests in place, and we have a monitoring tool to allow for live analysis of a network, but our tools for post-mortem analysis are lacking. While we have logging tools available, some visualization tools for automated post-mortem analysis could be useful.
- *Monitor Network Usage*
  The system works. However, we don't have any experimental data of the network use for our system. It could be useful to know how much bandwidth it needs, could it still be optimized ?