Inleiding Telematica, 2007
0440949 Andreas van Cranenburgh
Mon May 14 11:29:20 CEST 2007

Abstract

Using SOAP this program connects to virtual routers, thereby forming a connectivity graph with associated costs. Afterwards, a Floyd-Warshall implementation constructs a table of optimal routes in $O(V^3)$ time (where V is the number of vertices).

1 Virtual Router Service

This paper shall describe a SOAP client for polling virtual routers to discover interconnections and further nodes to poll. The set of nodes already visited is recorded, while every node that it comes across not yet in that set is added to a queue of nodes to visit. In the end, when the difference between the connections of the last visited node and the set of visited nodes yields the empty set, the whole mesh has been mapped (islands that are not connected to the mesh in any way will not be discovered, but they are by definition not part of the mesh).

Finally, an implentation of Floyd-Warshall constructs an optimal routing table, describing an optimal route (in terms of minimal total cost) between every pair of nodes.

1.1 SOAP

SOAP[1] is the protocol spoken to the Virtual Routers. It stands for Simple Object Access Protocol. Using SOAP, one can execute procedures remotely, regardless of the programming languages used between client and server; though in this case, both programs happen to have been written in Python.

Python does not have SOAP functionality in its standard library. This program uses SOAPpy[2]. The README file enclosed with the tarball of this program[3] explains its installation.

The assignment defines four procedures that can be called on the Virtual Routers:

- hello() – simple test procedure
- getVRName() – returns alphanumeric callsign for the current router. These names are only used for human-friendly display purposes
- getConnected() – returns a comma seperated string of connected nodes (represented as IP:port tuples)
- getLinkSpeed(node) – where 'node' is an IP (with the port part stripped off). Returns an integer representing the cost of the link between the currently connected node and the node passed as argument. As this is a cost, the lower the better.

Using SOAPpy, very little is necessary to use SOAP. See for example this snippet:

```
server = SOAPProxy(url)
try:
    VRname = server.getVRName()
except:
    print "error connecting to", url
```

It is also possible to see what happens under the hood:

```
>>> [...]
>>> server.config.dumpSOAPOut = 1
>>> server.config.dumpSOAPIn = 1
>>> print server.getVRName()
*** Outgoing SOAP ******************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

[1] http://en.wikipedia.org/wiki/SOAP
[2] See http://pywebscvs.sourceforge.net
[3] See https://unstable.nl/andreas/vr.tar.gz

```
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
>
<SOAP-ENV:Body>
<getVRName SOAP-ENC:root="1">
</getVRName>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
**********************************************************************
*** Incoming SOAP ****************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
>
<SOAP-ENV:Body>
<getVRNameResponse SOAP-ENC:root="1">
<Result xsi:type="xsd:string">AR1</Result>
</getVRNameResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
**********************************************************************
AR1
```

As can be seen, a lot of the data is XML overhead, such as the URLs referring to schemas. The line with the name of function and the one with the result are the most pertinent, the rest is just packaging.

1.2 Connecting to all routers

First of all, the program connects to a hard-coded IP:port tuple, the primer. This is "`http://ow120.science.uva.nl:8080/`", but has been converted to "`146.50.7.30:8080`", so as to conform to the format, as observed, returned by the Virtual Routers using the getConnected() procedure.

After a queue of routers to visit is formed, namely the neighbors of the first router, a loop is started to visit each of these routers, meanwhile recording routers that have already been visited and adding unknown routers to the queue. With each visit, the cost of the link between all of its neighbors is recorded, in order to calculate optimal routers later on.

Once this queue is empty, all of the routers have been visited, and the first phase has been succesfully completed.

1.3 Finding optimal routes

The second part is to form an optimal routing table. The routes are not necessarily the shortest, but they are optimised for minimal cost.

A very famous algorithm for finding optimal routes is the Dijkstra algorithm. This algorithm will, given a graph and a node in this graph, find optimal routes to all other nodes in the graph, with the given node as source. I have chosen not to use this algorithm, because it would have to be run seperately, for every node in the graph. Furthermore, it is an example of a Greedy Algorithm[4], whereas I tend to prefer Dynamic Programming[5] approaches.

One such approach is the Floyd-Warshall algorithm[6]. This algorithm takes a graph as its argument, and goes on to find every optimal route in the graph. So this code will only run once. Imagine finding the optimal route from Rome to Amsterdam. If the optimal route from Basel to Amsterdam has just been found, it's probably going to form a part of the one from

[4] `http://en.wikipedia.org/wiki/Greedy_algorithm`
[5] `http://en.wikipedia.org/wiki/Dynamic_Programming`
[6] See: `http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`

Rome to Amsterdam. With Dijkstra's algorithm, it would be computed all over again (the concept of "overlapping subproblems.")

The algorithm works by maintaining one big table of routes. I have implented this table as a dictionary, indexed by (source, destination) tuples, containing tuples consisting of the cost and route, represented as a list of virtual router addresses. For example:

```
path[ ( 1.2.3.4:80, 9.8.7.6:80 ) ] = ( 101, [1.2.3.4:80, ..., ..., 9.8.7.6:80] )
```

This table is indexed with the links and their costs that have been found in the first phase. Nodes which have no direct link between them are represented has having a cost of 32767 (which is used to represent infinity).

After that the real work begins. In a loop, a variable k iterates over all nodes. With each iteration every possible pair of nodes is examined, to see if the path via k is cheaper than the direct route between the pair of nodes in question.

Because every pair is examined, with every iteration of k, the number of comparisons is V^3, where V is the number of nodes.

1.4 Dumping the table

Finally, the whole routing table is printed to the screen. IPs are converted to router names, for human readability.

1.5 References

The code: `https://unstable.nl/andreas/vr.tar.gz`
Floyd-Warshall algorithm: `http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`