Challenge 5: Implementing distance-vector routing

1 Challenge

Design and implement a distance vector routing protocol. Multiple instances of your protocol will have to run and automatically find the least-cost paths between them, as quickly (in terms of iterations of the protocol) as possible, and even when connectivity and link costs change.

2 Details

The implementation must be done in a Java/C++ program. You run multiple instances of this, which will be connected to each other via our server. The server allows the instances to exchange packets, and checks the forwarding tables computed by the instances to calculate a score. The server also gives you feedback on what goes wrong.

As usual, you are provided with a skeleton Java/C++ program, in which only a single method (Java) / function (C++) needs to be implemented by you. This method/function is called `tick()`, and it will be executed periodically. This function gets a list (array) of packets that have been received from the neighbouring nodes since the previous invocation of tick(). You'll have to write code to interpret the received packets, update the forwarding table, and send out packets to the neighbouring nodes.

The `client` package contains a number of classes and interfaces, which must not be modified, but can be used by your code. Examples of their use are in the skeleton program, but for completeness here's an overview:

- `client/LinkLayer` which represents a virtual network interface. It supports querying for your own address, querying the cost to a certain neighbour, and sending a packet. See the examples in the skeleton code.
- `client/Packet` is used by your protocol to create packets, and send them over the LinkLayer.
- `client/DataTable` is sent as part of a Packet, and can contain data in table format (with rows and columns):
  - Instantiate a DataTable with `new DataTable(x)` where x is the number of columns in the table.
  - Use `set(row, column value)` to set individual values in the table, or `setRow(row, values[])` to set an entire row in one go.
  - Use `get(row, column)` or `getRow(row)` in a similar manner to retrieve data from the table.

Once you start the program, the framework will initiate a series of tests. Your implementation of `IRoutingProtocol` will be instantiated once for each test. The `IRoutingProtocol` interface specifies a number of methods which you are asked to implement:

- `init(LinkLayer)` is called at the start of each test. It allows you to set up the initial state of your program, and to save a reference to the LinkLayer which you will use to transfer data.
- `tick(packets)` is called subsequently once for every iteration of the protocol, with an array of received packets as its argument.

Again, please note that you are not allowed to change anything in the framework.

Each instance of your protocol will have at least one other instance connected to it through the `LinkLayer`. By sending a Packet with destination address 0, it will be received by all neighbouring instances connected to your `LinkLayer`. You can send a Packet to a specific neighbour by specifying its address as destination. Note however that specifying a destination that is not a direct neighbour is pointless as the packet will simply be lost.

The challenge is to implement a distance vector protocol, which exchanges messages with its neighbours, and fills the entries of the forwarding table. The server runs simulation for a fixed amount time, or rather ticks. Your protocol should converge before the simulation finishes. After each tick the server collects the forwarding

tables of all the nodes for scoring. Each time the server will verify all the routes that your program has entered into the forwarding table and check whether they are the shortest possible.

After you've started some number (say between 3 and 6) of instances of your program, you can go to the server web interface and select one test case out of 6 to quickly only run that single test. Note that it is possible to run a test with less than six nodes to have a simpler network to test with however, not all test cases are build to be run with less than six nodes. All tests are such that after a while, the topology changes, and your algorithm should converge before and after each change.

To avoid arbitrarily long tests, the server only gives your protocol a limited number iterations. Each network state is actually considered as separate tests for the grading.

You do not need to go through all test cases every time you test the program. You can just stop your program after it has done one test case, and see how it went, and if needed fix it, before trying all test cases (which can be time consuming).

For a final score, you have to run with six instances all tests by simply starting the simulation by pressing a key from within the client.

3 Hints for getting started

Start by having a look at the skeleton code in `protocol/DummyRoutingProtocol.java` (or `.cpp`), and running it. As with challenges 2 and 3, you'll need to set your group number and a password in `protocol/Program.java` (or `.cpp`), and you can make a copy of `DummyRoutingProtocol.java` (or `.cpp` and `.h`) under another name and modify `Program.java`) (or `.cpp`) to use your protocol. The skeleton program simply sends an empty packet at every clock tick, and shows the packets it receives from its neighbours.

Next, modify this protocol, such that when it receives a packet from a neighbour, it inserts that neighbour into the forwarding table. Thus, you get a partially filled-in forwarding table, and can already score some points.

Then implement the rest...

At first, you can simply ignore the fact that the topology changes after a while. Just focus on the initial topology and try to get that to work correctly. Only as a later step, detect the topology change (by polling `LinkLayer.getLinkCost` for changes), and process it.

4 Submission and grading

In addition to implementing a protocol, you are required to write a one-page report on your implementation. Explain what you tried to achieve, if, how and why it works.

As usual, grading is based both on the performance of your solution (both the optimality of the found routes and how quickly your system converged), and the report.

Get a teaching assistant to sign off your work, and submit the source code (no binaries) in a zip archive, and your report in PDF format to Blackboard before 18:00.

5 Learning objectives

- Thoroughly understand how distance vector works.
- Get insight in distributed protocols.