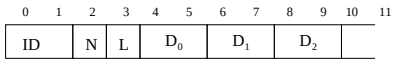
Programming Assignment #3 - HW #6

COMS W4119 - Computer Networks — Due April 5, 2006
Spring 2006 — Prof. Rubenstein

Programming Assignments must be e-mailed to the TA. The e-mail should contain a single compressed file containing all files to be turned in. If desired, answers to questions can be turned in at the beginning of class on the day that the homework is due. CVN students have one additional day. Late assignments will not be accepted.

In this programming assignment, you will build a node that participates in the all-pairs shortest paths Bellman-Ford (Distance Vector) Protocol. You shall implement two versions: one that does not implement poison reverse and the other that does.

Packet Format



Each packet you transmit during the execution of the protocol should contain the following fields (in order, as pictured above):

- ID: a unique ID for your network (to allow multiple students to simultaneously operate in different networks). This is a 2-byte quantity. How it gets sent is described below.
- N: your node's ID, a 1-byte quantity. Your node's ID will always be 0.
- L: the number of distance entries in your vector, a 1-byte quantity.
- D_i: for each $0 \leq i < L$, the current shortest-known distance to node i. The distance is a 2-byte quantity, and should be sent in network-byte order. This distance will always be an integer value in the networks we test with.

Note that $D_0 = 0$, since your distance to yourself is always 0. Upon learning of your node's neighbors, your node should forward a vector to each neighbor initially, and then subsequently any time the vector's value changes. If poison reverse is not used, then the same vector should be forwarded to all neighbors (including the one you may have most recently received a vector from). When poison reverse is used, the vector sent will have the distance to D_i set to 65535 for any node that is reached (via shortest path) through that neighbor.

Your neighbors will send you messages as well using pakcets with the same format.

Code details

Your code should take three input parameters:

- an IP address (in dotted decimal notation)
- a port number
- Poison-reverse indicator: a value of 0 if poison reverse should not be implemented, and a value of 1 if poision reverse should be implemented.

Your executable should then connect to the IP address and port given as input parameters via a TCP (`SOCK_STREAM`) connection and retrieve a stream of data. This stream will have the following format:

- The first two bytes will be your unique ID.

- The third byte will indicate the number of nodes in the network (including yourself).
- The fourth byte will indicate the number of nodes who are your immediate neighbors. Suppose this number is k.
- The $2i + 3$rd byte will give the ID of your ith neighbor, and the $2i + 4$th byte will give you the length of the edge between you and this ith neighbor.

Let S_1 represent the socket used to exchange this information. Before closing S_1, do the following:

- Open a new socket S_2 on a port of your choosing (let p be the number of this port), and prepare to listen/accept for k connections on this socket, one from each of your k neighbors.
- On S_1, send the following 3-byte sequence:
  - a byte whose value is 0 if poison reverse is not being implemented, and 1 if it is being implemented
  - in network byte order, the value of p (this uses two bytes).
- Close S_1.

Once the remote client receives the value of p, it will create the k neighbor nodes, each of whom will attempt to connect to your specified IP/port location. After connecting, each neighbor will immediately send you a packet of their current distance vector in the same format as the packet you are supposed to send them. The packet will reveal their identity (the node ID number). You can wait for all of them to respond at least once before sending your first message if you want, but you do not have to wait (you can send an update to any neighbor who has already identified itself).

Your output

Your code should output your table (of shortest known paths to each destination through each known neighbor), followed by the current vector of shortest distances to each node and the ID of the first node on the path. For instance, the vector:

$((10, 3), (12, 6), (4, 7), \cdots)$ indicates that node 0 is distance 10, and the first hop of this path is through node 3, node 1 is distance 12 and the first hop of this path is through node 6, etc.

Other comments

Your code should not explicitly terminate. However, if the code is written correctly, it should eventually cease transmitting messages (as long as the underlying graph is not changed) because all of its neighbors will eventually cease sending update messages.

At some testing locations, the length of an edge in the graph will be peridically changed, causing the protocol to "come alive" and adapt the shortest path trees to the modified graph.

Testing/Verification

The webpage `http://www.cs.columbia.edu/~danr/4119/pa3-test.html` will provide additional details on how to test your code. We plan to provide different testing locations your node can contact. We will provide descriptions of the configuration networks so that you can observe and check the correctness of your code. In addition, we plan to provide you with executables so you can do the testing yourself locally.

What to turn in

The assignment should be turned in electronically, by e-mail to 4119-submit@cs.columbia.edu. You must turn in **as a single attachment** all work wrapped up together as a single gzipped tarfile with filename `pa2-CUID.tgz` (where `CUID` is your Columbia UNI). The tarfile should decompress to create a directory `pa2-CUID/` with the following files (example given in C):

- `DV.h`: header information for basic distance vector protocol.
- tt DV.c: main code
- If you have additional c code that you wrote that is included by these files, include that code as well.
- `DV`: your distance vector executable.
- `Makefile`: your makefile or `make.txt` that explains the procedures you followed to compile your file.
- A file named `test.txt` that includes some tests on which you ran your code.

Grading

Your grade will be based on *our* execution of your code against some pre-determined test cases (that may be slightly different) from the tests we provide above. So make sure that your code works, or we will not be able to test. We will also randomly spot-check the source code turned in to verify that it matches the object code, and will also compare object code across student assignments.

Again, *the code must execute on Columbia CS Linux boxes to receive credit.* More details on this forthcoming (e.g., a specific machine on which it must work).