**CIS 427 – Project 3**

Ravi Ramasami

**Usage**

This project has 4 main files:

* routemsg.py, which contains the code that allows routers and the link daemon to send messages between each other. This is included in all the other files, and does not execute anything. The message format is listed there.
* link.py, which is the link daemon that allows routers to connect to each other. It takes 5 command line arguments and uses them to send messages to each router, telling them that there is a link between the two routers with the cost given.
* router.py, which uses the distributed Bellman-Ford algorithm to determine the shortest path to each router in the system. This is the naive implementation, and does not use poison reverse.
* routerpr.py, which is a duplicate of router.py except that it uses poison reverse to partially avoid the count to infinity problem.

To run the program, open the command prompt, change directory to where the files are stored, and type:

* python router.py <router name> <router port> to start a regular router,
* python routerpr.py <router name> <router port> to start a poison reverse router,
* python link.py <router 1 name> <router 1 port> <router 2 name> <router 2 port> <cost> to link two routers together.

If the name of your python interpreter is different, use that instead.

If you do not want to run it as a command line program, change the prompt\_for\_args variable to True in the files, and run them normally. It will prompt you for command line arguments, and then use them to run the program.

If you give the program invalid arguments, they will be handled by the program and an error message will be displayed.

If you give the program 0 arguments, usage will be displayed.

**Implementation**

routemsg.py

The message format is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Router or link message?  0x00: router  0xFF: link | Source router name  single character, 1 byte | Source router port  4 bytes | Data  variable length |

This implements make\_routemsg() and get\_routemsg(), which encode 4-tuples into a byte array ready to send over a socket and decode a byte array into a 4-tuple, respectively, using the format above. The data section is a Python object that is converted into a string with the pickle module.

link.py

Error checking is done on the number and type of the arguments before anything is done. After error checking is passed, we create a socket and construct two link messages.

The first is sent to the first router, from the second router, where the data is the cost of the link. The second is sent to the second router, from the first router, where the data is the same as before.

After this, we are done, so we close the socket and terminate the program.

router.py

Error checking is done on the number and type of the arguments before anything is done. After error checking is passed, we initialize our variables: neighbors (dictionary) to store which neighbors we have, and dist\_vecs (dictionary of ditionaries) to store the distance vectors for itself and its neighbors.

Start a loop: we will get a message, examine its contents, and branch based on its source. We also have a variable send\_dvs (boolean) to determine if dvs should be sent this round.

* If we get a link message, we are getting a new link cost. So add the pair (source port, link cost) to neighbors under the key of the name of the source router. Since a link cost changed, we want to set send\_dvs to True.  
  If the key is not present in dist\_vecs, we add it to dist\_vecs, setting the shortest path to (link cost, 1 hop, next router in path is where it came from).
* If we get a router message, we take the message data and replace their old distance vector with the new one. We then apply the Bellman-Ford equation:  
  Iterate through all destinations. For each destination, compute min{c(self,v) + d(v,dest)} where v is all the neighbors of self. We compare the new distance vectors we get to the old distance vectors: if they are not the same, then set send\_dvs to True. If nothing was changed, we don't want to send new messages.

After this is done, we use the function print\_distvecs to print the distance vector information.  
Now check send\_dvs. If it is true, we create a message with the data being our dv (which will be pickled and sent), and send it to all our neighbors. We know their addresses since we store them in neighbors.  
The loop ends here, and repeats forever until the program is terminated. This is all we need for this program to work.

routerpr.py

The program is the same as before, but with a slight modification in the router message branch. After the Bellman-Ford equation has been applied for a particular destination, we check the paths through all the neighbors and see if a routing 2-cycle was formed: This would be where the next router on the path from self to destination is the neighbor, and the next router on the path from the neighbor to destination is self. If it is, we set that particular entry to infinity. It will be corrected later when the neighbor sends a new route back to us.

**Tests**

Instead of taking screenshots, I made a video instead. This is easier to view and harder to fake. See here: <https://www.youtube.com/watch?v=O64YJ4muPdg>