TÉCNICO LISBOA

Computer Networks — 2021/22	**Assignment:**	Project 2
Understanding Routing Protocols	**Issued:**	2022-12-30
Routing Simulation	**Submission Due:**	2023-01-06

1 Overview

In this project, you will learn the basics of network routing protocols by implementing simplified versions of the distance-vector (with and without reverse path poisoning), path-vector, and link-state algorithms. You will also learn about simple event based simulators which are commonly used to simulate networks. Though the simulator code itself is given, you will implement the code that handles link changes and messaging at each router to allow them to react to the changing network and exchange messages with neighbors. Your code will use these mechanisms to negotiate routes among neighboring routers to find the shortest routes to each destination.

2 Using the Simulator

This assignment comes with a code distribution which includes the following files:

- **routing-simulator.cpp** - The code for the event based routing simulator itself (written in C++). Do not submit changes to this file, as it will be overwritten during the grading process.
- **routing-simulator.h** - C header file with the API used to implement routing protocols. Includes the function signatures for the handlers you must implement, as well as the functions you will use to interact with the simulator. Do not submit changes to this file, as it will be overwritten during the grading process.
- **dv.c** - Skeleton C file to fill in with the implementation of the *Distance-Vector without Reverse Path Poisoning* routing algorithm. Edit this file as needed for submission.
- **dvrpp.c** - Skeleton C file to fill in with the implementation of the *Distance-Vector with Reverse Path Poisoning* routing algorithm. Edit this file as needed for submission.
- **pv.c** - Skeleton C file to fill in with the implementation of the *Path-Vector* routing algorithm. Edit this file as needed for submission.
- **ls.c** - Skeleton C file to fill in with the implementation of the *Link-State* routing algorithm. Edit this file as needed for submission.
- **Makefile** - GNU Makefile that builds the simulator and routing protocols. Do not submit changes to this file, as it will be overwritten during the grading process.
- **dot-to-pdf.sh** - A shell script to convert the GraphViz DOT files generated by the simulator into PDFs for viewing.

2.1 Building

The assignment distribution comes with a `Makefile` that builds both the simulator and the routing protocols that you will implement. Simply run `make` to build the project. This will create four binaries - one for each routing protocol: `dv-simulator`, `dvrpp-simulator`, `pv-simulator`, and `ls-simulator`.

2.2 Running

The build process compiles and links together the simulator with each of the routing protocols. Though each binary simulates a different protocol, they can all be run in the same way. A typical simulation run looks like this:

```
:~$ make
:~$ ./dv-simulator --steps-dot output.dot topology.net
:~$ ./dot-to-pdf.sh output.dot output.pdf
```

This runs the simulator with `topology.net` as the input network topology file. As a result, the simulator produces `output.dot`, a GraphViz DOT file illustrating the state of the network at each step of the simulation. This DOT file is then converted into a PDF for easier viewing. The flags used here produce a detailed output showing each of the routes and messages. This is useful for debugging, but it can also show an overwhelming amount of information. The simulator has additional options and flags to filter this information down:

```
:~$ ./dv-simulator --help
Usage: ./dv-simulator [--final-dot <dot-file>] \
                      [--help] \
                      [--hide-future-messages] \
                      [--max-events <limit>] \
                      [--show-routes-for <node>] \
                      [--steps-dot <dot-file>] \
                      [--] <topology-file>

 --final-dot <dot-file>    - Generate a dot file showing the final result.
 --help                    - Show this help screen.
 --hide-future-messages    - Declutter dot files by only showing the
                             current message (default: show).
 --max-events <limit>      - Put a limit on the number of simulation events
                             to process (default: no limit).
 --show-routes-for <node>  - Declutter dot files by only showing routes for
                             <node> (default: show all).
 --steps-dot <dot-file>    - Generate a dot file showing each simulation
                             step.
```

2.3 Network Topology Files

The simulator takes as input a network topology file. We include some simple ones with the distribution, but you should definitely create your own during testing. These files are simple text files with a list of link change events, one per line, in the following format: `<time> <first node> <second node> <cost>`. To create a network, simply bring up the

links between the respective nodes (timestamps, nodes, and link costs are all integers):

```
0 0 1 1
0 1 2 1
```

In this example, a simple 3 node network is setup by bringing up links 0-1 and 1-2 (both with cost 1), at the very beginning of the simulation, at $t = 0$.

You can also change the network over time to see how the routing protocols reacts to change. Simply add new link change events, at different point in time, changing the cost of each link. Set the link cost to 255 to disable a link altogether.

2.4 Internals

The simulator is built around a simple event processing engine. The engine keeps a list of currently scheduled events, ordered by the time at which they are programmed to occur, and processes them one by one until none are left. Time is virtual and represented as a simple integer, simulation time is not tied to an actual clock time. The current time is simply the time of the event currently being processed.

There are two types of events in this simulator, *link change* events which are loaded from the topology file, and *message* events which are used to simulate communication among neighboring nodes. The link change events are all loaded from the input file once, whereas message events are continuously added throughout the simulation. As the simulation goes on, the routing protocols implementations can send messages, which schedule an event to deliver the message one time epoch into the future.

3 Specification

In this project, you will implement four different routing protocol variants by implementing handlers for each node's link change and message delivery events. The functions you will need to implement for each protocol are:

```
void *init_state();
void notify_link_change(node_t neighbor, cost_t new_cost);
void notify_receive_message(node_t sender, message_t message);
```

`init_state()` is used to initialize the router. In this function, use `malloc()` to create a new buffer to hold the router's state and initialize it as appropriate. You'll later be able to access this state using the `get_state()`, as described below.

The `notify_link_change()` function notifies the router that its link with neighboring node `neighbor` has changed cost to `new_cost`. The `notify_receive_message()` function delivers a message from `sender`.

You will also need to interact with the simulator by calling functions in the simulator API. These functions are:

```
node_t get_current_node();
node_t get_first_node();
node_t get_next_node(node_t node);
node_t get_last_node();
cost_t get_link_cost(node_t neighbor);
cost_t COST_ADD(cost_t a, cost_t b)
void set_route(node_t destination, node_t next_hop, cost_t cost);
void send_message(node_t neighbor, message_t message);
void *get_state();
```

`get_current_node()` returns the (integer) ID of the current node. `get_first_node()`, `get_next_node(node_t node)`, and `get_last_node()` make it easy to iterate over each node in the network, *e.g.*:

```
for (node_t node = get_first_node();
     node <= get_last_node();
     node = get_next_node(node)) {
  // some code
}
```

You can also use the `MAX_NODES` macro for the largest possible node ID that is supported.

`get_link_cost()` allows the router to get the status of the links to its neighbors, returning the respective links cost. You can also use this to detect if a particular node is currently a neighbor or not, by checking if the cost of the link to it is less than `COST_INFINITY`. As you compute routes, you will need to add together link costs, the `COST_ADD` macro helps with this by accounting for wrap-around and properly handling `COST_INFINITY`.

`set_route()` is used to setup a route to `destination`, via `next_hop`, with cost `cost`, in the data plane. You can also delete a route by assigning the cost of `COST_INFINITY` to it. These routes will show up in the output DOT files, indicating the protocol's progress.

You can use the `send_message()` function to send a message to `neighbor`.

Finally, `get_state()` is used to allow each router to access its own local state. This state is stored in a struct that you allocate and initialize in `init_state()`. You can then use the pointer returned by `get_state()` to access that struct and update it in place, as needed. All state *must* be managed using this mechanism, do not create static or global variables in the router code.

3.1 Distance-Vector (without Reverse Path Poisoning)

Distance-Vector is a logically distributed protocol where each router announces its routes (a distance-vector) to its neighbors, indicating which nodes it knows how to reach and at what cost. As each node discovers new routes via link changes and neighbor messages, it uses the Bellman-Ford algorithm to determine the best routes and announce any changes to its neighbors. Make sure to only announce routes if they have changed, as otherwise the simulation may not converge. Implement this protocol in `dv.c`.

3.2 Distance-Vector with Reverse Path Poisoning

Distance-Vector with Reverse Path Poisoning is a variant of the above protocol where each router omits routes (sets their cost to `COST_INFINITY`) that go through a neighbor, when advertising routes to that neighbor. This avoids the simplest form of the count-to-infinity problem, as we learned in class. Implement this protocol in `dvrpp.c`. Feel free to copy `dv.c` into `dvrpp.c` and then make this small change.

3.3 Path Vector

Path-Vector is an even more advanced version of the distance-vector protocols from above. In this protocol, each node not only advertises which other nodes it knows how to get to and at what cost, but also the entire path it will take to get there. As each node learns new routes from its neighbors it checks for routing loops before picking which route to use. This avoids the count-to-infinity problem in its entirety and allows the network to converge more quickly. Implement this protocol in `pv.c`.

3.4 Link State

Unlike the previous protocols, link-state is logically centralized, in that nodes do not cooperate to pick the best routes, just to learn the network topology. Nodes here interact only to communicate their local network topology (the list of neighboring link costs). As each node builds its understanding of the global network topology, it uses Dijkstra's algorithm to find the best routes for each destination.

To share the network topology use a message like this:

```
typedef struct {
  cost_t link_cost[MAX_NODES];
  int version;
} link_state_t;

typedef struct {
```

```
  link_state_t ls[MAX_NODES];
} data_t;
```

With this message format, each node advertises the most recent *link states* it has heard from throughout the network. Each *link state* represents the costs of all of the links from one node to its neighbors, as well as a version number to allow updates to propagate without regressing. In effect, `data.ls[1].link_cost[2]` is the latest known cost of the link from node 1 to node 2. As nodes receive these messages from their neighbor they check the version number of each *link state* and only update their local state if the version is more recent. Whenever this local state changes, the node should broadcast the state to all of its neighbors to propagate the knowledge across the network. Eventually, the entire network converges on the same set of versions, at which point everyone knows the global topology. As the node's understanding of the global topology evolves, run Dijkstra's algorithm to set the routes accordingly. Implement this protocol in `ls.c`.

4 Submission

Project development and submission will be handled via an online git repository that will be assigned to you on gitlab.rnl.tecnico.ulisboa.pt. As you develop the project, maintain the project C code in your git repository and update the files in place, preserving the original file structure. Please do not commit any build artifacts including any object files or binaries that you built yourself. Once you are ready to submit the final revision of your code, tag it as `project2-submission` and push the tag to the remote repo. This can typically be done as follows:

```
:~$ git tag project2-submission
:~$ git push origin project2-submission
```

Downloading and building the submission from the command line should look something like the following:

```
:~$ mkdir test
:~$ cd test/
:~/test$ git clone <repo URL> .
:~/test$ git checkout project2-submission
:~/test$ ls
routing-simulator.cpp  Makefile  dvrpp.c  ls.c
routing-simulator.h    dv.c      pv.c
:~/test$ make
:~/test$ ls
Makefile      dvrpp-simulator  ls.c          pv.o
dv-simulator  dvrpp.c          ls.o          routing-simulator.cpp
dv.c          dvrpp.o          pv-simulator  routing-simulator.h
dv.o          ls-simulator     pv.c          routing-simulator.o
```

5 Advice

5.1 Debugging

The simulator already provides some useful outputs for debugging. Use the `--steps-dot`, as shown earlier, to generate a visual graph showing how the network evolves over time. Bold black lines show the network topology and colored arrows show the routes each router has established through `set_route()`. Dashed arrows show messages queued up for delivery. These arrows are gray when the message is staged to be delivered in the future and turn black when being delivered. Link change notifications are also shown as a dot on the corresponding link. If the graph gets too complex, filter out unneeded information with the additional options offered by the simulator.

The easiest way to debug a system is to output its state at key places in the code. When adding debug prints to the code, consider using `get_current_node()` to indicate which node is being run and then use `grep` to filter the outputs for a specific node that is giving you trouble. This function can also be used to create conditional breakpoints in `gdb`. e.g. `break notify_receive_message if get_current_node() == 1` will stop the code whenever node 1 receives a message.