Assignment 3 – Shortest Path Routing using Link-State Algorithms

CS456/656 Computer Networks

Spring 2020

<u>NOTE: CHECK PIAZZA FOR ANY CHANGES TO THE ASSIGNMENT SPEC, REQUIREMENTS, GRADING AND EXPECTATIONS</u>

Contents

NOTE: CHECK PIAZZA FOR ANY CHANGES TO THE ASSIGNMENT SPEC, REQUIREMENTS, GRADING AND EXPECTATIONS.

Objective

The goal of this assignment is to implement a shortest path routing algorithm, much like the one found in Open Shortest Path First (OSPF). Routers using OSPF construct a topological map, such as a network graph, of the entire autonomous system (AS) by using link-state algorithms. Recall that link-state algorithms are centralized, least-cost path, routing algorithms, so every router in the AS must have complete, global information about connectivity and cost. This is accomplished by flooding link-state information, which contains the identities of the directly connected links and their costs. Then, the router executes Dijkstra's shortest-path algorithm to determine the shortest-path tree to all other routers, with itself as the root node.

Background

You will find detailed discussion on routing, link state algorithms and OSPF protocol in the Internet, in the textbook in Chapter 5 *The Network Layer: Control Plane*, Section 5.1 *Introduction*, Section 5.2.1 *The Link-State Routing Algorithm*, and Section 5.3 *Intra-AS Routing in the Internet: OSPF*.

Specifications

Introduction

You will be writing a program which will behave like a router's OSPF module (or a simplification of it). Much like there are many routers in an AS network, there will be many instances of your program running concurrently. In this assignment, we will have a topology file that describes a network: routers connected by links where each link has a cost.

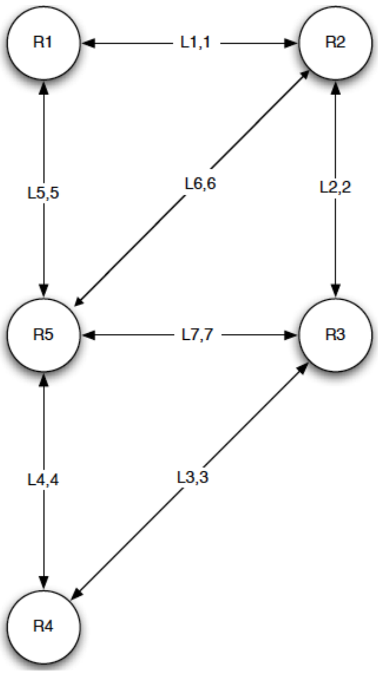


Figure 1. Network Topology

In Figure 1 above you can see a topology with 5 routers (labeled **R1** to **R5**). They links connecting them are full-duplex and have the same weight in both directions; the link labeling is **`L<link id>,<link weight>`** so **`L7,7`** means link ID 7 with a weight of 7. Link IDs and router IDs are not related.

If we were to use the topology in Figure 1, we would have 5 copies of your program running, one for each router. From now on, we'll say 'virtual router' to refer to the program you will be writing **but know that the term 'virtual router' also has other meanings in the networking world** and we're just using it here for convenience. Each virtual router will be given knowledge of its attached links and their weights (e.g. R3 will know its own ID is 3 and know it has the links L2,2, L7,7, and L3,3). It's up to the virtual routers to then communicate with each other, using a link-state protocol, so that each virtual router has a complete and total representation of the topology. Using this knowledge of the topology, each virtual router will then calculate the shortest path from itself to every other router in the topology, using Dijkstra's shortest-path algorithm.

Network Forwarding Emulator (NFE)

To simplify the assignment, we will use a network emulator to replace the links between your virtual routers. In other words, your virtual routers will be talking to the network emulator in order to communicate with their immediate neighbours. The emulator, called the Network Forwarding Emulator (NFE), is responsible for forwarding communication from one virtual router to another virtual router. The NFE does not modify communication packets exchanged between your virtual routers, it simply moves them.

The NFE is also responsible for providing each virtual router with knowledge of its links and their weights. Of course, a real router would get that information by examining its hardware and by consulting a configuration file; however, we will have the virtual router talk to the NFE in an initialisation phase to get this information.

Topology File

The NFE expects a topology file (in a JSON format) which describes the routers and the network which is being emulated. **You will be given a topology file** but can also write your own.

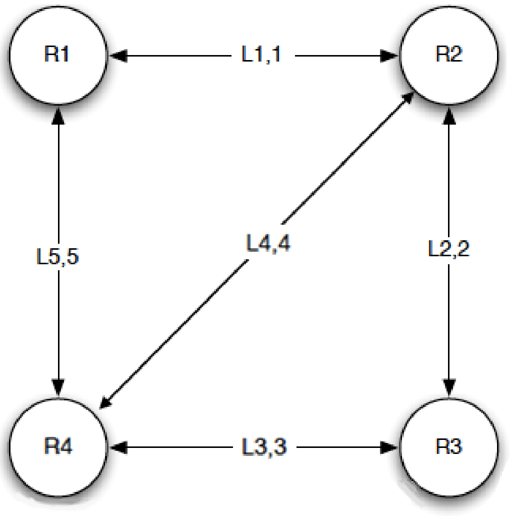
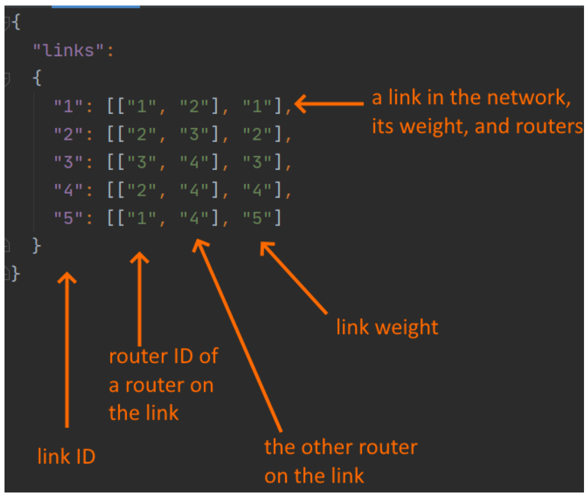


Figure 2.A 4-router, 5-link topology

For this 4-router and 5-link topology, the topology file would look like this:



```
{
  "links":
  {
    "1": [["1", "2"], "1"],
    "2": [["2", "3"], "2"],
    "3": [["3", "4"], "3"],
    "4": [["2", "4"], "4"],
    "5": [["1", "4"], "5"]
  }
}
```

a link in the network, its weight, and routers
link weight
router ID of a router on the link
link ID
the other router on the link

Figure 3. Topology file example

The emulator will look for the links key and its values. The topology file structure is “link-centered” such that an entry is the link ID, the two virtual routers attached to it and the weight of the link. In the above, the first entry is **`“1”: [[“1”, “2”]: “1”].`**

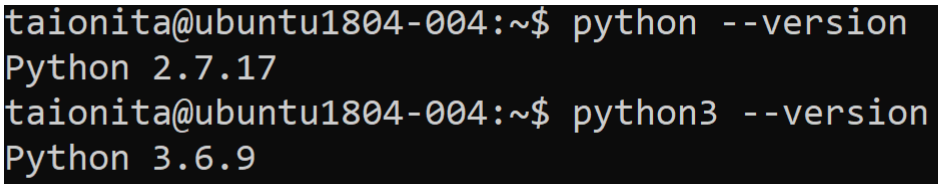
For an entry that looks like **`“1”: [[“4”, “2”]: “60”]`**, we understand that it describes link ID 1, connecting virtual routers 4 and 2 as neighbours and the link has a weight of 60.

Topology restrictions for a topology file, if you're writing your own topology file:

- Link IDs need to be unique
- Every Router ID is mapped to a single virtual router
- All IDs and weights need to be parsable as an integer (keep it under 255)
- Only one link can connect the same 2 routers (e.g. routerID 1 and routerID 2 can only have 1 link between them)
- Virtual routers cannot be connected to themselves (e.g. routerID 1 can't have a link to routerID 1)
- The network cannot be partitioned (i.e. every router must be reachable from any other router)

Starting the NFE

The NFE is written in Python3 so you'll need to make sure you have Python 3.5 or higher installed. The machines you can access at `linux.student.cs.uwaterloo.ca` have Python 3.6.9 as of this writing. Note that Linux distros will often have both Python2 and Python3; Python2 is often just shortened to `python` and Python3 to `python3`. Python2 is deprecated and should not be used in this assignment. The NFE can be run in Linux, MacOS and Windows environments, provided Python 3.5 or higher is installed but it's lightly recommended you stick to Linux.



```
taionita@ubuntu1804-004:~$ python --version
Python 2.7.17
taionita@ubuntu1804-004:~$ python3 --version
Python 3.6.9
```

Figure 4. Python2 and Python3 showing their versions; we want Python3

The NFE will run as a UDP server and listen on an IP and port. It will also want the path to the topology file. We start like so:

`python3 nfe.py <IP> <port> <topo_filepath>`

e.g.

`python3 nfe.py localhost 2000 topo.json`

It's recommended you run both the NFE and your virtual routers on the same machine.

Based on the topology file, the NFE will expect a certain number of virtual routers to connect to it e.g. if the topology file describes a network with 5 routers, 5 virtual routers will need to connect to the NFE.

Starting Your Virtual Router (Getting the Virtual Router ID)

Regardless of the language you write your virtual router in, you will need to accept command-line arguments. For instance, the **virtualrouter** binary here is accepting the 3 needed arguments:

```
./virtualrouter <nfe-ip> <nfe-port> <virtual-router-id>
```

nfe-ip is the IP where NFE is listening, **nfe-port** is the port and **virtual-router-id** will be the router ID assigned to that instance of the virtual router.

The **virtual-router-id** argument passed to the virtual router must match the router ID's in the topology e.g. if the topology mentions router ID 1,2 and 3 then three virtual routers are needed, one with router ID 1, another with 2 and the third with router ID 3.

Interacting with the NFE

Once the NFE is started, it'll have loaded the topology file and is ready to interact with the virtual routers. As it's a UDP server, it expects the virtual routers to send UDP datagrams to it as well as be listening for a UDP reply from the server.

Interacting with the NFE happens in two phases: **init phase** and **forwarding phase**.

init phase:

- The NFE waits for a (valid) **`init`** message from each of the virtual routers
- The NFE replies with an **`init-reply`** message to each virtual router, providing each their attached links and the link weights
- The NFE waits for all virtual routers to contact it before continuing to the next phase
  - i.e. the topology has n routers in it, all n virtual routers will need to contact the NFE before moving on to the next phase

Forwarding phase:

- The NFE will wait for a **`link-state advertisement`** (**LSA**) messages from virtual routers.
- It will do some validation but will not modify the message
- It will forward the **LSA** to the appropriate neighbour

Init Phase (Init, Init-Reply)

The NFE will be waiting for an **`init`** message over UDP from each and every virtual router.

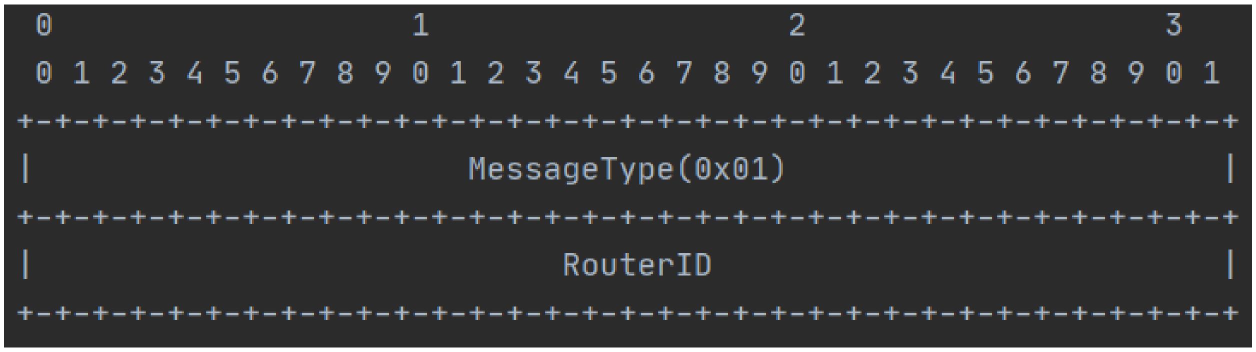
Once it's received all the **`init`** messages, the NFE will then reply to each and every virtual router, individually, with an **`init-reply`** message.

INIT MESSAGE

The init message has 2 fields.

1. MessageType field (32 bits): informs the receiver of the type of message the rest of the message is. Set to **1**
2. RouterID field (32 bits): holds

Here are two other presentations of the message:



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        MessageType(0x01)                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             RouterID                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5. Init Message. RFC-style.

```
init {

int32 MessageType (0x01)

int32 RouterID

}
```

Figure 6. Init Message. Struct-style

INIT-REPLY MESSAGE

The init-reply message is more complicated, we can think of it as having 2 conceptual parts: a header and a payload.

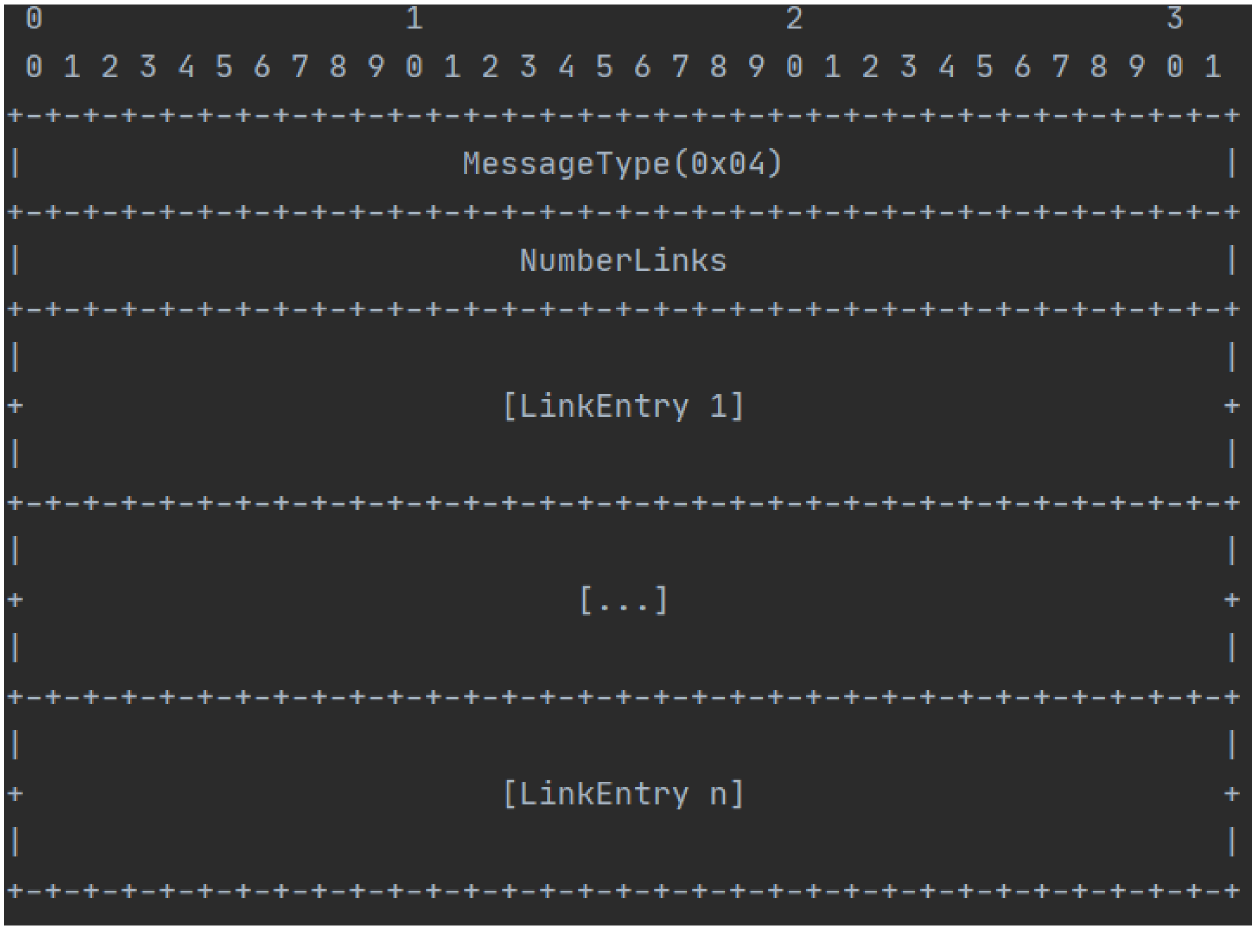
The header part of the message has 2 fields:

1. MessageType (32 bits): Set to **4**
2. Number of links (32 bits): number of link entries the router will find in the rest of the init-reply message i.e. how many links this router has.

The payload part is composed of **Link Entries**, one entry for every link the virtual router.

A single link entry has 2 fields:

1. LinkID (32 bits)
2. LinkCost (32 bits)



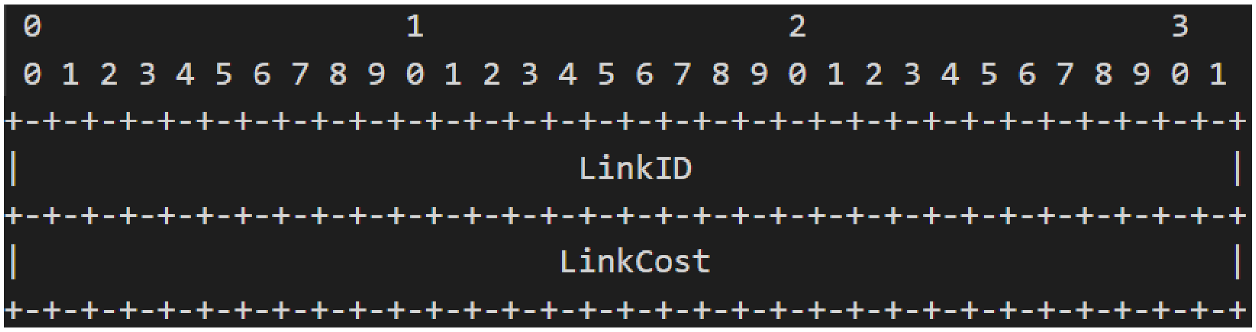
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       MessageType(0x04)                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          NumberLinks                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         [LinkEntry 1]                         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                             [...]                             +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                         [LinkEntry n]                         +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7. Init-Reply, RFC-style

```
init-reply {
int32 MessageType (0x04)
int32 NumberLinks
int32 LinkEntry[NumberLinks]
}
```

Figure 8. Init-Reply, struct-style

Now, let’s see two representations of the Link Entry



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             LinkID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            LinkCost                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 9. Link-Entry, part of Init-Reply. RFC-style

```
link-entry {
int32 LinkID(0x04)
int32 LinkCost
}
```

Figure 10. Link-Entry, part of Init-Reply. struct-style

INIT-REPLY EXAMPLE

Given the following router from a topology:

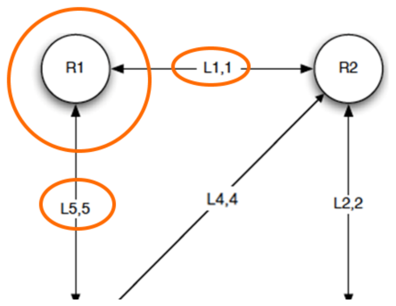
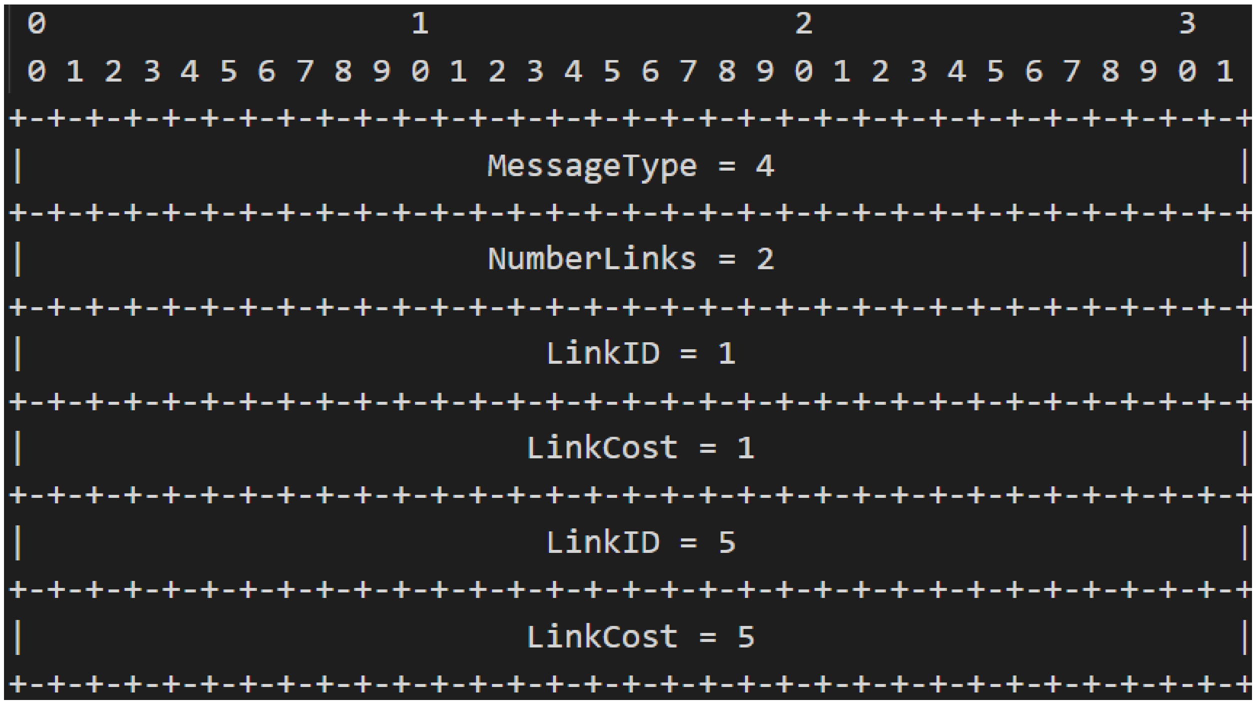


Figure 11. Init-Reply Topology Example

It has 2 links: **link ID 1, cost 1** and **link ID 5, cost 5**

An init-reply for this **specific** virtual router and topology would look like this:



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        MessageType = 4                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        NumberLinks = 2                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           LinkID = 1                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          LinkCost = 1                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           LinkID = 5                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          LinkCost = 5                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 12. Init-Reply Example, RFC-style

```
init-reply-specific-example {

int32 MessageType = 4

int32 NumberLinks = 2

int32 LinkID = 1

int32 LinkCost = 1

int32 LinkID = 5

int32 LinkCost = 5

}
```

Figure 13. Init-Reply Example, struct-style

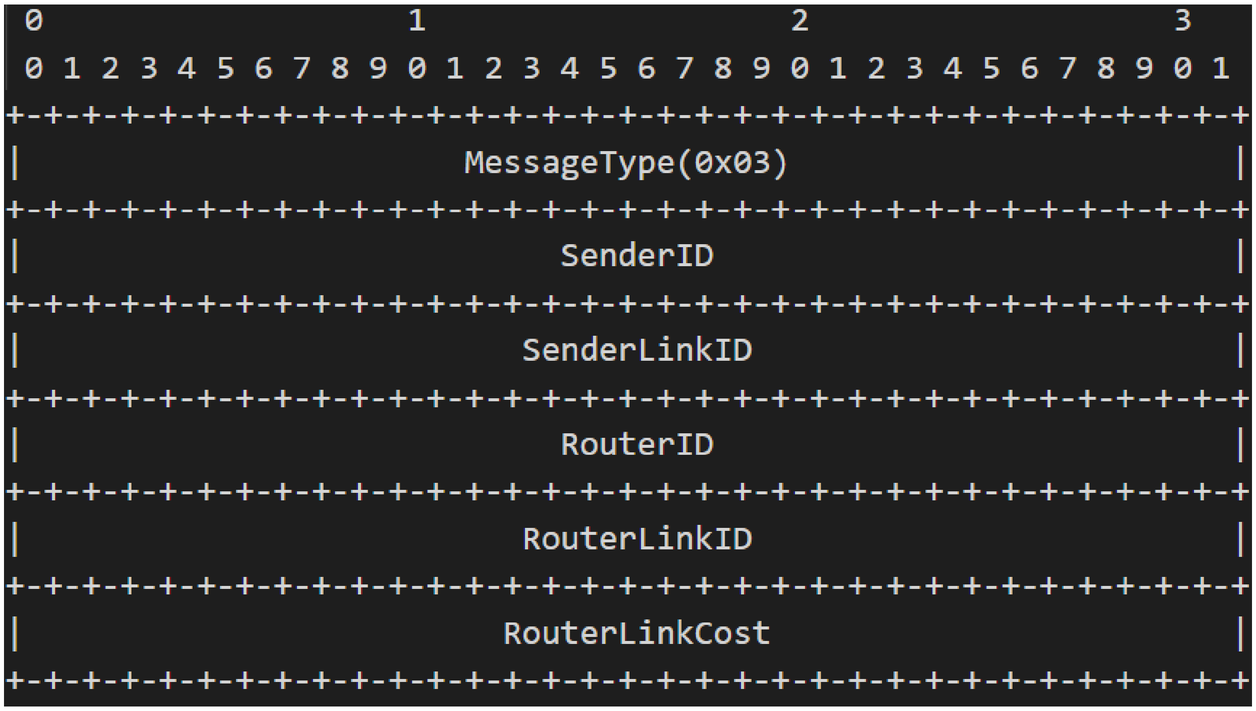
Forwarding Phase (LSA)

After the NFE has sent all the init-reply messages to all virtual routers, it enters the forwarding phase. In this phase, the NFE is only forwarding **LSA** messages from one router to another.

In this phase, the virtual routers are forwarding information about their own links to their immediate neighbours as well as forwarding information they have received about distant neighbours (e.g. the neighbour of the neighbour). This will be discussed in the Convergence section.

The LSA message has 6 fields; the names of the fields are from the perspective of the virtual router receiving the message:

1. Message Type (32 bits): set to **3**
2. Sender ID: the router ID of the virtual router sending this very message; this can only be an immediate neighbour
3. Sender Link ID: the link ID of the link on which the message is sent; this can only be a link attached to both the receiving and sending virtual router
4. Router ID: the router ID about which the update is about (e.g. this might be the neighbour's neighbour's neighbour)
5. Router Link ID: the link ID that the previous Router ID field has
6. Router Link Cost: the cost of that link



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       MessageType(0x03)                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            SenderID                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          SenderLinkID                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            RouterID                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          RouterLinkID                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         RouterLinkCost                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 14. LSA, RFC-style

```
LSA {

int32 MessageType(0x03)

int32 SenderID

int32 SenderLinkID

int32 RouterID

int32 RouterLinkID

int32 RouterLinkCost

}
```

Figure 15. LSA, struct-style

Topology Database

At all times, virtual routers should maintain a view of the network.

At first, that internal view will tell the virtual router it’s alone in the network. As this virtual router receives LSAs from neighbours, it adds information to its internal topology database. Eventually the internal topology and the ‘real’ topology will look the same.

You will need to grow the topology database; you can represent it internally however you wish, but the easiest way is generally using a graph, with vertices/nodes and edges.

Routing Information Base (RIB)

At all times, virtual routers should maintain a shortest-path routing table. This routing table should contain the shortest distance to every other router as well as the first hop needed on that shortest path e.g. if a virtual router 1 has 2 links, some shortest paths might use one link, others another; the routing table must contain this information.

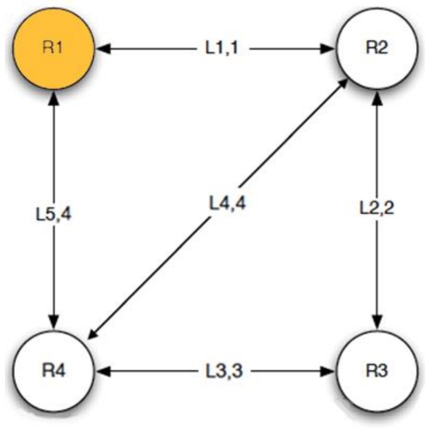


Figure 16. RIB Topology, source router has Router ID 1

Routing Table for **Router ID 1**		
Destination	Cost	Next Hop
R2	1	R2
R3	3	R2
R4	4	R4

Figure 17. Final Routing Table Example for Router ID 1

As the internal topology grows due to LSA updates, the routing table grows and changes. Some new destinations are added, some existing costs are updated and some next-hops are changed.

In order to fill the routing table, you need to use the internal topology you are also maintaining. You need to run a shortest-path algorithm, like Dijkstra's, to traverse your topology and find the shortest paths.

Convergence

Initially, all virtual routers exchange LSAs with their immediate neighbours. After this first exchange, all routers know about their immediate neighbours and their links. Let’s imagine a virtual router with 3 links that receives an LSA on link 1. The router updates its topology database and its RIB. Because the router has other links (link 2 and link 3), it forwards the LSA it has received to both those links. The routers on the other end of link 2 and link 3 will also forward the LSA to their other links.

Note that there would be a danger that LSAs will bounce around indefinitely on the network, forever forwarded. Because we know our topology is static, we use a simple heuristic to stop LSAs from bouncing around indefinitely: if the virtual router receives an LSA whose information (RouterID, RouterLinkID, LinkCost) it’s seen before, it does not forward it and drops it (ignores it) the LSA instead. This means that the system should stabilize eventually and no more LSAs should be forwarded.

Also note that eventually, all routers will converge on the same internal topology and have an identical topology database.

Changing the Sender ID and Sender Link ID

Recall that the LSA has 6 fields, two of which are Sender ID and Sender Link ID. These IDs are related to the identity of the neighbouring router that is forwarding an LSA.

As such, every time a router receives and LSA and right before forwarding it, it needs to change the Sender ID and Sender Link ID field. So if Router 1 receives an LSA from Router 11 and it needs to forward it Router 2, it’ll change the Sender ID from 11 to 1. It will also change the Sender Link ID to the Link ID between itself and Router 2.

Example 1

Let's follow the first LSA emitted initially by Router 1 in the topology below:

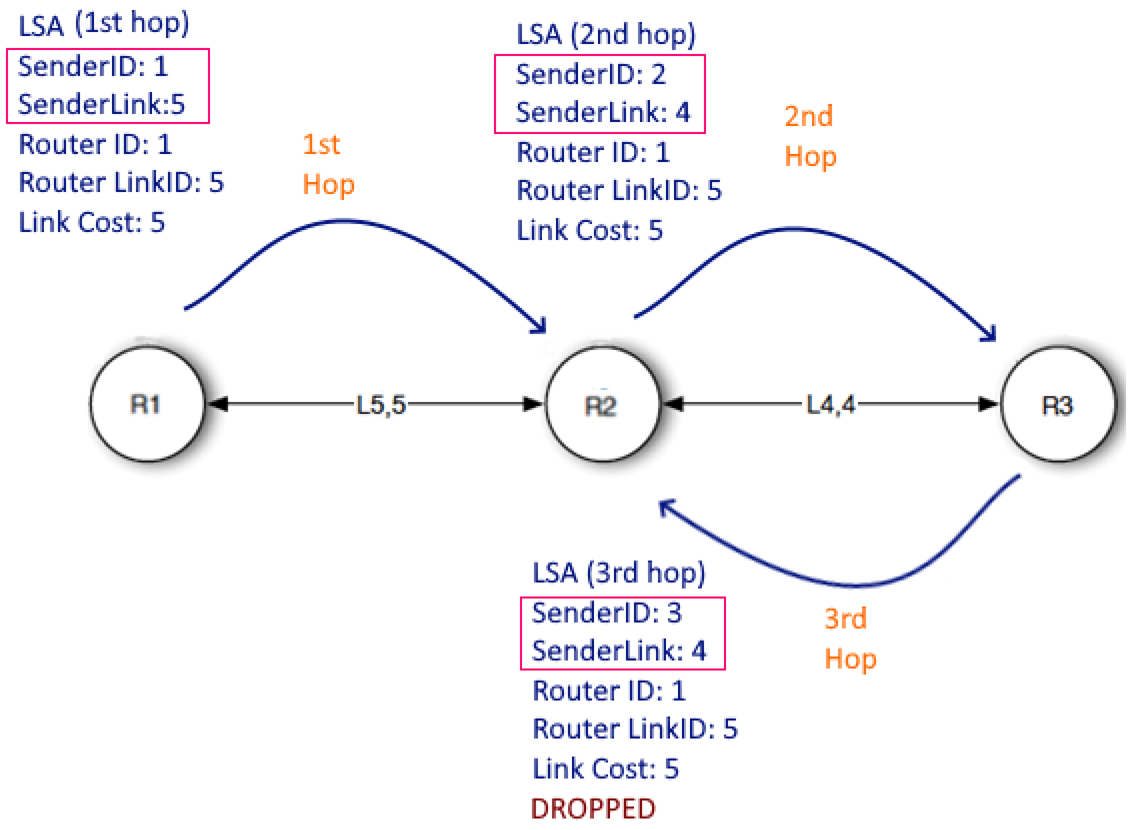


Figure 18. Convergence Example 1

As we can see in Figure 18, R1 emits the LSA with information about itself and one of its links (it just happens to only have 1 link). This information "payload" is the fields **`Router ID: 1, Router LinkID: 5, Link Cost:5`**; these are the fields we wish every router in the network to be aware of. We also notice that **`SenderID`** and **`SenderLink`** is changed at each hop to reflect the router and link over which the forwarding is about to happen. Finally, we notice that after the 3rd hop, R2 knows it's seen this LSA before and drops it: there is no 4th hop from R2 to R1.

While we were following a single LSA in this example, there would be several LSAs being received and sent concurrently between all 3 routers.

Example 2

Let's follow R1 as it receives LSAs from its neighbours; this is only one of the possible timelines, certain LSAs could arrive in different orders. Below the timeline is a table describing how the topology database and routing table would evolve.

1. R1 would receive 3 LSAs from R2 about R2's 3 links
   - this is just the immediate neighbour emitting LSAs about itself
2. R1 would receive 3 LSAs from R4 about R4's 3 links
   - this is just the immediate neighbour emitting LSAs about itself
3. R1 would receive 2 LSAs from R2 about R3's 2 links
   - This is R2 forwarding R3's links
4. R1 would receive 2 LSAs from R4 about R3's 2 links
   - This is R4 forwarding R3's links, unbeknownst to it, R1 already had them

From R1's Perspective

Events Timeline	**Topology Database**	**Routing Table**

0
(before any broadcast)
	R1 <--L1-->?
	R1 <--L5-->?
		Empty

1	R1 <--L1--> **R2** R1 <--L5-->? **R2 <--L1--> R1** **R2 <--L2--> ?** **R2 <--L4--> ?**	Destin. / Cost / Next R2 / 1 / R2

2	R1 <--L1--> R2
	R1 <--L5--> **R4**
	R2 <--L1--> R1
	R2 <--L2--> ?
	R2 <--L4--> **R4**
	R4 <--L5--> R1
	R4 <--L4--> R2
	R4 <--L3--> ?
		Destin. / Cost / Next
		R2 / 1 / R2
		R4 / 5 / R4

3	R1 <--L1--> R2
	R1 <--L5--> R4
	R2 <--L2--> R1
	R2 <--L2--> **R3**
	R2 <--L4--> R4
	R4 <--L5--> R1
	R4 <--L4--> R2
	R4 <--L3--> **R3**
	R3 <--L2--> R2
	R3 <--L3--> R4
		Destin. / Cost / Next
		R2 / 1 / R2
		R4 / 5 / R4
		R3 / 3 / R2

4	(same as event 3)	(same as event 3)

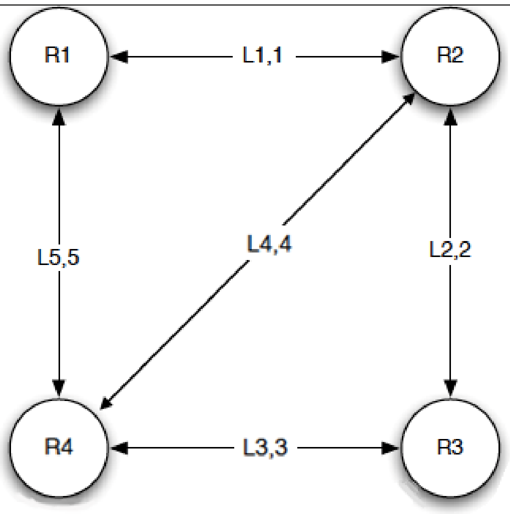


Figure 19. Example 2 Excruciating Details

Note on Network Endianness

Don't forget that x86-AMD64 architecture is little-endian while network transmission of bytes is big-endian. The libraries in your language of choice might already take care of that for you or it might need you to handle it yourself somehow. For example, C would need you to use htons() functions to convert between host and network endianness. Check for your specific language.

Note On Our Using of UDP

While UDP is usually not reliable like TCP is, we'll be using UDP through localhost which means the kernel doesn't actually send the datagram out through the network card. Instead, the kernel will move the data from one internal queue to another internal queue. For our intents and purposes, UDP is reliable in these circumstances.

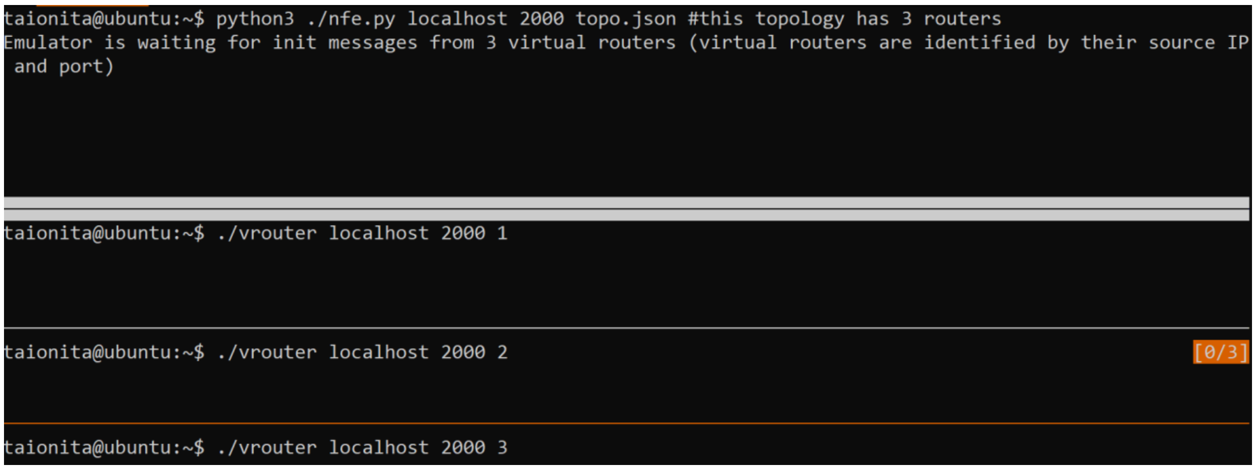
Note On Emulator Identifying Virtual Routers

Recall that when a client sends a UDP datagram to a server, it sends data from source IP and source port to a destination IP and destination port. As such, when the server gets the data from the client, it also knows which source port and source IP was used; when the server replies to the UDP client, it sends messages to that source IP and source port.

As you can imagine, the NFE uses the source port and source IP to identify which virtual router is which. Therefore, if your virtual router tears down its socket, the virtual router and the NFE will not be able to communicate anymore.

Running It All

This is an example of how you'd run the emulator and the virtual routers. **This is not necessarily the best way to do it, it's just demonstrative.**



```
taionita@ubuntu:~$ python3 ./nfe.py localhost 2000 topo.json #this topology has 3 routers
Emulator is waiting for init messages from 3 virtual routers (virtual routers are identified by their source IP
 and port)
```

```
taionita@ubuntu:~$ ./vrouter localhost 2000 1
```

```
taionita@ubuntu:~$ ./vrouter localhost 2000 2                                                            [0/3]
```

```
taionita@ubuntu:~$ ./vrouter localhost 2000 3
```

Figure 20. Running It All

The top terminal window/pane starts the NFE, which will bind as a UDP server on localhost:2000, and read in topo.json. Because the topo.json lists 3 routers, the NFE waits for 3 virtual routers to contact it (and says so, as you can see in the terminal)

The last 3 terminal windows/panes are the different virtual routers started with 3 command-line arguments: the IP and port where the NFE is listening as well as their own router ID.

Recall that the virtual router’s router ID must match up with the topology’s routerIDs.

Deliverables

If you have questions, reach out to the course staff earlier rather than later.

Program Stdout Output

On stdout, the virtual router should output:

- (1) The initial LSA being emitted
- (2) The LSA it's forwarding
- (3) The LSA it's received
- (4) If it's dropping an LSA

Here are the output formats where:

DROPPING means the LSA is **going to be dropped**

SENDING(F) means the LSA is being **forwarded**

SENDING(E) means the LSA is being **emitted initially**

SID means **SenderID**,

SLID means **SenderLinkID**

RID means **RouterID**

RLID means **RouterLinkID**

LC means **LinkCost**

(1) **`Sending(E):SID(<value>),SLID(<value>),RID(<value>),RLID(<value>),LC(<value>)`**

(2) **`Sending(F):SID(<value>),SLID(<value>),RID(<value>),RLID(<value>),LC(<value>)`**

(3) **`Received:SID(<value>),SLID(<value>),RID(<value>),RLID(<value>),LC(<value>)`**

(4) **`Dropping:SID(<value>),SLID(<value>),RID(<value>),RLID(<value>),LC(<value>)`**

Program File Output

Every virtual router will need to produce 2 files:

- topology_<routerID>.out
- routingtable_<routerID>.out

where <routerID> is the ID of that particular virtual router; if there are 3 virtual routers running (with routerIDs 1,2 and 3), there should be 6 files: *topology_1.out*, *topology_2.out*, *topology_3.out*, *routingtable_1.out*, *routingtable_2.out* and *routingtable_3.out*.

TOPOLOGY OUTPUT FILE

This should contain the topology database of the virtual router as it gets updated. That is, every time an LSA triggers an update to the topology database, it should be output to this file. This will mean the file will grow in size as the topology database changes, the later topology entries larger than the earlier ones.

The format of a single line in a topology entry is as follows:

```
router:<routerid>,router:<routerid>,linkid:<linkid>,cost:<cost>
```

Before the topology entry, the word **`TOPOLOGY`** should appear alone on its own line.
Here is the last topology entry for a network with 2 routers only, and 1 link between them:

```
TOPOLOGY
router:1,router:2,linkid:5,cost:12
router:2,router:1,linkid:5,cost:12
```

Note the output contains both directions, router 1 to router 2 over link 5 and router 2 to router 1 over link 5.

Also note the topology entry should not display a link where one of the routers is unknown e.g. we don't want to include something like **`router:1,router:?,linkid:5,cost:12`**

Finally, in the appendix of this document, you'll find a full topology output of 1 router in a small network as an example.

ROUTING TABLE OUTPUT FILE

This file is similar to the topology output file; it should contain the growing routing table. Every time the routing table changes, it should be appended to this file.

The format of a single line in a routing entry is as follows:

```
<destination ID>:<next hop ID>,<total cost>
```

Before the routing entry, the word **`ROUTING`** should appear alone on its own line.

In the appendix of this document, you'll find a full routing output of 1 router in a small network as an example.

To Be Handed In for Grading

Upload the following, zipped up in one file, to the LEARN Assignment 3 drop box:

- the source code for your virtual router
- a script, Makefile or whatever mechanism that will prepare the program to be run
    - e.g. compilation or downloading dependencies
- README that should contain instructions on how to run program, or any particular things to note
- Among other things, grading will consider the output files of your virtual routers for correctness

Documentation

- Leave reasonable amount of comments in the code; make the code modular and readable. Prefer readability over performance, cleverness or succinctness.

GRADING SCHEME

- Grading will use a single topology, the *grading_topo.json*
- Your virtual router instances will be left to run for up to 10 seconds. They are not expected to terminate; 10 seconds should be ample time for topology convergence
- Do not hesitate to ask questions, earlier rather than later
- Your program will be run on `linux.student.cs.uwaterloo.ca` machines. It doesn't mean you need to develop there, but it means it needs to work there. If there are particular problems, reach out to the course staff.

Grading Elements [out of /100]

- **Documentation** [/5]
    - README [/1]
    - Code comments [/2]
    - Code is readable [/2]
- **Running the code** [/8]
    - Make/preparation script works [/3]
    - Virtual router outputs on stdout [/5]
- **Virtual Routers send init message** [/7]
- **Virtual Routers send their link-state to their neighbours** [/7]
    - This is the initial emission of link-state propagation that gets everything rolling
    - LSAs are properly formed (correct fields)
- **Virtual Routers propagate the LSAs they receive correctly** [/17]
    - This does not cover stopping
- **The LSAs do not get forwarded indefinitely** [/14]
    - The virtual routers correctly drop the LSAs, preventing LSAs indefinitely bouncing around the network
- **Topology Database** [/21]
    - All the virtual routers share the same topology database [/7]
    - The topology database is correct [/14]
- **Router routing table** [/21]
    - Routing table for every router

APPENDIX

Topology and Routing Table File Output

Given the following topology

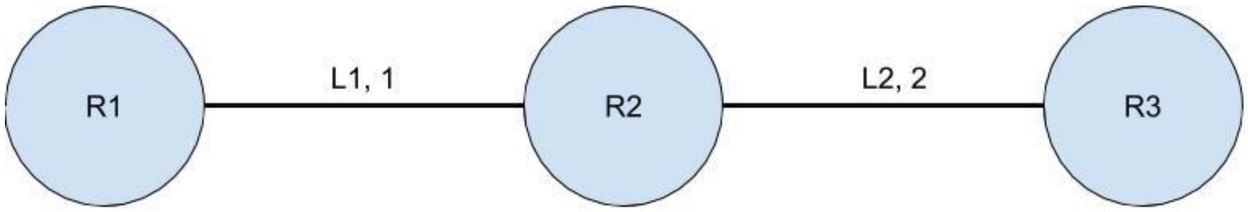


Figure 21. Boring Topology

TOPOLOGY_1.OUT

This is the entire file with 2 topology entries, but you can imagine how a larger network would create many more entries. As the filename implies, this is the topology from router ID 1

```
TOPOLOGY
router:1,router:2,linkid:1,cost:1
router:2,router:1,linkid:1,cost:1

TOPOLOGY
router:1,router:2,linkid:1,cost:1
router:2,router:1,linkid:1,cost:1
router:2,router:3,linkid:2,cost:2
router:3,router:2,linkid:2,cost:2
```

Figure 22. Topology_1.Out

ROUTINGTABLE_1.OUT

```
ROUTING
2:2,1

ROUTING
2:2,1
3:2,3
```

Figure 23. RoutingTable_1.out

Example of Stdout Output for Router 1

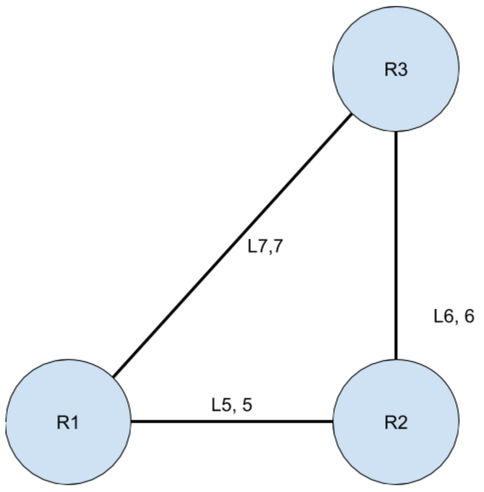


Figure 24. Slightly less boring but ever so slightly

From virtual router 1's perspective (coloring not needed in your output):

```
Sending(E):SID(1),SLID(5),RID(1),RLID(5),LC(5)
Sending(E):SID(1),SLID(5),RID(1),RLID(7),LC(7)
Sending(E):SID(1),SLID(7),RID(1),RLID(5),LC(5)
Sending(E):SID(1),SLID(7),RID(1),RLID(7),LC(7)
Received:SID(2),SLID(5),RID(2),RLID(5),LC(5)
Received:SID(2),SLID(5),RID(2),RLID(6),LC(6)
Sending(F):SID(1),SLID(7),RID(2),RLID(5),LC(5)
Sending(F):SID(1),SLID(7),RID(2),RLID(6),LC(6)
Received:SID(3),SLID(7),RID(3),RLID(7),LC(7)
Received:SID(3),SLID(7),RID(3),RLID(6),LC(6)
Sending(F):SID(1),SLID(5),RID(3),RLID(7),LC(7)
Sending(F):SID(1),SLID(3),RID(3),RLID(6),LC(6)
Received:SID(2),SLID(5),RID(3),RLID(6),LC(6)
Received:SID(2),SLID(5),RID(3),RLID(7),LC(7)
Dropping:SID(2),SLID(5),RID(3),RLID(6),LC(6)
Dropping:SID(2),SLID(5),RID(3),RLID(7),LC(7)
Received:SID(3),SLID(7),RID(2),RLID(5),LC(5)
Received:SID(3),SLID(7),RID(2),RLID(6),LC(6)
Dropping:SID(3),SLID(7),RID(2),RLID(5),LC(5)
Dropping:SID(3),SLID(7),RID(3),RLID(6),LC(6)
```

Telling R2 about R1's 2 links and telling R3 about R1's 2 links

Receiving LSAs about R2's 2 links and forwarding to R3

Dropping LSAs which are advertising R3's 2 links (we already have this knowledge)

Figure 25. Example Stdout, Coloring Added Manually for Demonstration Purposes