Introduction

Welcome to the Routing In Fat Trees (RIFT) hackathon at the IETF 104 in Prague!

At this hackathon we will do "chaos monkey" testing of two RIFT implementations: the RIFT-Python open source implementation and the Juniper Networks RIFT implementation.

This hackathon is inspired by the concept of chaos engineering and in particular by the chaos monkey tool developed by Netflix, which intentionally and randomly shuts down servers in the Netflix production environment. This is intended to build confidence in the system's capability to withstand turbulent and unexpected conditions.

In this hackathon we will do something similar for RIFT networks. We will build a large fat tree network of RIFT routers. Then we will randomly insert failures into the network and make sure that the network behaves correctly. We will make sure the RIFT routers don't crash or otherwise behave badly. And we will make sure that the routers forwarding tables and link-state databases converge correctly.

We will use (and possibly enhance) the chaos testing framework as well as the visualization and verification tools that are already part of the RIFT-Python open source project.

Even if you are not much interested in the RIFT routing protocol, but you are interested in testing large network topologies using container-like technology, or in visualization tools, or in chaos testing, this might be the right hackathon for you!
Goals

The goals for the RIFT hackathon at IETF 104 in Prague are:

1. Allow attendees to learn about the RIFT protocol in a hands-on and fun manner.
2. Gain experience with new tools such as topology generators, visualizers, chaos testing, and Python scripting in the context of routing protocol development and verification.
3. Use chaos testing to expose bugs in the RIFT protocol itself or in the current two RIFT implementations: the RIFT-Python open source implementation, and the Juniper Networks implementation.

Agenda

The agenda for the hackathon is:

1. Get familiar with RIFT-Python and Juniper Networks RIFT.
2. Use a topology generation tool to create a large fat tree topology (also known as Clos topology) containing many RIFT routers.
3. Run the entire network in the Amazon Web Services (AWS) cloud, using container-like technology to run many RIFT routers on a single AWS instance and to interconnect them with simulated Ethernet links.
4. Use a wide variety of tools, including visualization tools and automated checking tools, to make sure that the individual RIFT routers and the RIFT network as a whole operate correctly.
5. Use a "chaos monkey" framework to intentionally introduce failures in the network, and use the verification tools to verify that the routers and the network still operate correctly.
6. Mix multiple RIFT implementations in a single network and verify that they interoperate correctly.
**RIFT Implementations**

The following two RIFT implementations will participate in the hackathon:

- RIFT-Python, an open source RIFT implementation developed by Bruno Rijsman: https://github.com/brunorijsman/rift-python
- A RIFT implementation from Juniper Networks: https://www.juniper.net/us/en/dm/free-rift-trial/

It is trivial to host other RIFT implementations our testing environment, so if anyone else has a RIFT implementation feel free to bring it to the hackathon and we will happily include it.

**Prerequisites**

You will need to bring a laptop with an SSH client. You don't need any other special software installed,

The following skills will help you participate in the hackathon. If you don't have these skills yet and you would like to join to learn these skills, you are still welcome.

Familiarity with the RIFT protocol is the first requisite. Read the IETF draft and possibly view some YouTube videos.

We will use the Amazon Web Services (AWS) cloud to host our testing environment. It is assumed that you are familiar with creating instances (virtual machines) in AWS. If you want to follow these instructions by yourself, you will need your own AWS account. At the hackathon at the IETF itself, we can let you SSH into AWS instances that we create using our account (we will generate temporary keys).

RIFT-Python itself as well as all the associated tools (topology generators, visualizers, chaos testing tools) are written mostly in Python with some use of bash scripts and SVG graphics as well. If you plan to hack the code, knowledge of Python is essential.

If you wish to try out the Juniper RIFT implementation, you need to contact Juniper to get access to the software (for detailed instructions see section "Download the Juniper RIFT Free Trial Software" in the IETF 102 RIFT hackathon report). We will bring a copy to the hackathon.

**Resources**

You can find extra information here:

- The IETF RIFT protocol specification
- The IETF RIFT working group page
• The Git repository for the RIFT-Python open source implementation
• The documentation for RIFT-Python
• The Juniper Networks RIFT implementation
• IETF 102 RIFT hackathon report and presentation video
• IETF 103 RIFT status of open source implementation report and presentation video
• YouTube videos: there are several good videos on RIFT, just search for "RIFT Routing In Fat Trees".
• Tony P's recordings of the meetings of the weekly RIFT core architecture team.

Enough introduction, let's get hacking.
Create an Amazon Web Services (AWS) instance

Create an Ubuntu Server 18.04 LTS instance on AWS. A t2.micro instance should be big enough, but if you are going to run very large topologies (more than 20 nodes) you might want to create something bigger. Make sure you use exactly Ubuntu 18.04 and not some other Linux distribution or some other version of Ubuntu. Since IETF 104 will be in Prague, you might want to us an AWS region in Europe for lower latency.

Once the AWS instance is up and running, SSH into it. If multiple IETF attendees are sharing an AWS instance, it would be wise to generate a new AWS key pair just for the hackathon. Remember that the username for AWS Ubuntu instances is ubuntu.

Install RIFT-Python in the AWS instance

Clone the RIFT-Python GitHub repository into the newly created AWS instance:

```
$ git clone https://github.com/brunorijsman/rift-python.git
Cloning into 'rift-python'...
remote: Enumerating objects: 291, done.
remote: Counting objects: 100% (291/291), done.
remote: Compressing objects: 100% (205/205), done.
remote: Total 3614 (delta 200), reused 158 (delta 86), pack-reused 3323
Receiving objects: 100% (3614/3614), 78.82 MiB | 17.35 MiB/s, done.
Resolving deltas: 100% (2538/2538), done.
```

Change directory to the top of the git repository:

```
$ cd rift-python
```

Run the installation script. If anything goes wrong you can running the installation script with the -v command line option for more verbose output. If you want to understand what the installation script does behind the scenes, read the full installation instructions.

```
$ install/install.sh
Updating apt-get... OK
Installing python3-venv... OK
Creating virtual environment... OK
Activating virtual environment... OK
Installing Python module dependencies... OK
```

Activate the Python virtual environment that was created by the installation script:

```
$ source env/bin/activate
(env) $
```
Start a simple RIFT-Python topology in single process mode

There various ways to run multi-router RIFT topologies. The simplest approach is to run all RIFT routers in a single Python process. We will use this method here to do a quick test run of a topology to verify that the installation was successful. Later on, when we do the real chaos testing, we will use a different method where each RIFT router runs as a separate Python process in its own network namespace. But for now we keep it simple to get acquainted with RIFT-Python.

Make sure that you are still logged in to your AWS instance, that the current directory is the root directory of the Git repository, and that your Python virtual environment has been activated (see last steps in previous section).

We are going to start a small topology that contains 2 leaf nodes, 2 spine nodes, and 2 super-spine nodes. We do this by starting the Python rift package and passing the topology file two_by_two_by_two.yaml in the topology directory as an argument:

```bash
(env) $ python rift topology/two_by_two_by_two.yaml
Command Line Interface (CLI) available on port 49890
```

If you get an error message "Command 'python' not found", it means you forgot to activate the virtual environment.

You won't have to create these configuration files (also known as topology files) yourself. Examples are provided, and we will be using a generator to generate the configuration file later on. But if you are curious what they look like, you can open the file topology/two_by_two_by_two.yaml and have a look. Note that RIFT-Python and RIFT-Juniper use (mostly) compatible configuration files.

RIFT-Python reports a port number (49890 in this example), which you use to telnet into the RIFT router and access the Command Line Interface (CLI). Multiple telnet sessions can connect to the same RIFT-Python process, so multiple participants at the hackathon can interact with the same RIFT router at the same time.

Open another terminal window and SSH into the same AWS instance. Once in the AWS instance, use telnet to connect to the reported CLI port (i.e. replace 49890 with the port number that RIFT-Python provided when you started it).

```
$ telnet localhost 49890
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

Page 6 of 39
To exit the CLI, use the `exit` or `stop` command. If you use the `exit` command, you exit the CLI but the RIFT router keeps running. If you use the `stop` command, you exit the CLI and the RIFT router also stops running. For now, use the `stop` command:

```
agg_101> stop
Connection closed by foreign host.
$ 
```

This will cause the RIFT-Python process in the other terminal to terminate:

```
$ python rift topology/two_by_two_by_two.yaml
Command Line Interface (CLI) available on port 39773
$    # We are back at the shell prompt in the original terminal where RIFT-Python was started
```

This whole business of having to open a separate terminal window to connect to the CLI is a bit of a nuisance. There is a simpler method to avoid this: when you start RIFT-Python, you can use the `--interactive` (or simply `-i`) command-line option to start RIFT in interactive mode. In interactive mode, you can access the CLI directly through standard input and output instead of using a separate telnet session. The disadvantage of this method is that RIFT-Python does not yet support command-line editing using the cursor keys. To get out of the CLI and terminate the RIFT router, enter the `exit` or `stop` command (in interactive mode the RIFT router always terminates when you exit the CLI, regardless of whether you use the `exit` or `stop` command).

```
$ python rift --interactive topology/two_by_two_by_two.yaml
agg_101> exit
$ 
```

In addition to `--interactive`, the rift package also provides some other command-line options. For details see the documentation or use the `--help` command-line option:
$ python rift --help
usage: rift [-h] [-p | -n] [-l LOG_LEVEL]
       [-i | --telnet-port-file TELNET_PORT_FILE]
       [--ipv4-multicast-loopback-disable]
       [--ipv6-multicast-loopback-disable]
       [configfile]
Routing In Fat Trees (RIFT) protocol engine
positional arguments:
  configfile            Configuration filename
optional arguments:
  -h, --help            show this help message and exit
  -p, --passive         Run only the nodes marked as passive
  -n, --non-passive     Run all nodes except those marked as passive
  -l LOG_LEVEL, --log-level LOG_LEVEL
                        Log level (debug, info, warning, error, critical)
  -i, --interactive     Start Command Line Interface (CLI) on stdin/stdout
                        instead of port
  --telnet-port-file TELNET_PORT_FILE
                        Write telnet port to the specified file
  --ipv4-multicast-loopback-disable
                        Disable IPv4 loopback on multicast send sockets
  --ipv6-multicast-loopback-disable
                        Disable IPv6 loopback on multicast send sockets

A quick test drive of RIFT-Python

Now that we know how to start and stop a RIFT topology, let's try out some CLI commands to get an idea of what RIFT-Python looks like. We are not going to give an exhaustive description of every available command; for that see the CLI documentation.

I recommend you follow along with the following examples by typing them into the CLI yourself. Make sure that you restart RIFT-Python with the two_by_two_by_two.yaml topology first:

$ python rift --interactive topology/two_by_two_by_two.yaml
agg_101>

Let's start with the help command. Unsurprisingly, it prints the complete list of available commands.

spine-1> help
agg_101> help
clear engine statistics
clear interface <interface> statistics
clear node statistics
...SNIP...
show spf direction <direction> destination <destination>
show tie-db
stop
We mentioned previously that RIFT-Python runs multiple RIFT routers in a single process. For consistency with the IETF draft and other RIFT implementations, these routers are called "nodes". The single RIFT-Python process that contains multiple nodes is called the "engine". In this document we use the terms router and node interchangeably, and we use the terms process and engine interchangeably.

You can use the show engine command to view information about the engine:

```
agg_101> show engine
+-----------------------------------------------------+
| Stand-alone                          | False        |
| Interactive                         | True         |
| Simulated Interfaces                | True         |
| Physical Interface                  | eth0         |
| Telnet Port File                    | None         |
| IPv4 Multicast Loopback             | True         |
| IPv6 Multicast Loopback             | True         |
| Number of Nodes                     | 10           |
| Transmit Source Address             | 127.0.0.1    |
| Flooding Reduction Enabled          | True         |
| Flooding Reduction Redundancy       | 2            |
| Flooding Reduction Similarity       | 2            |
| Flooding Reduction System Random    | 11092270224597309270 |
+-----------------------------------------------------+
```

You can use the show nodes (plural, with an s) command to view a list of nodes (i.e. routers) which are running in the engine (i.e. in the RIFT-Python process):

```
agg_101> show nodes
+----------------+---------+---------+
| Node           | System  | Running |
| Name           | ID      |
|----------------+---------+---------|
| agg_101        | 101     | True    |
| agg_102        | 102     | True    |
| agg_201        | 201     | True    |
| ...SNIP...     |         |         |
| edge_2002      | 2002    | True    |
+----------------+---------+---------+
```

In the CLI you are always in the context of a particular node called the "current node". The name of the current node is reflected in the CLI prompt (agg_101 in this example):

```
agg_101>
```

You can change the current node using the set node <node> command:
If you are following along with the example, make sure to change the node back to agg_101 because there is where the more interesting things are happening.

In addition to set node command, the CLI offers a few other set commands that you can look up in the CLI documentation mentioned above. However, it is currently not possible to do general configuration using the CLI; if you want to change the configuration you have to edit the topology file, stop the RIFT-Python process (using the stop command) and restart it.

Use the show node (singular, no s) command to view the details of the current node.
**Node:**

<table>
<thead>
<tr>
<th>Name</th>
<th>agg_101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passive</td>
<td>False</td>
</tr>
<tr>
<td>Running</td>
<td>True</td>
</tr>
<tr>
<td>System ID</td>
<td>101</td>
</tr>
<tr>
<td>Configured Level</td>
<td>1</td>
</tr>
<tr>
<td>Leaf Only</td>
<td>False</td>
</tr>
<tr>
<td>Leaf 2 Leaf</td>
<td>False</td>
</tr>
<tr>
<td>Top of Fabric Flag</td>
<td>True</td>
</tr>
<tr>
<td>Zero Touch Provisioning (ZTP) Enabled</td>
<td>False</td>
</tr>
<tr>
<td>ZTP FSM State</td>
<td>UPDATING_CLIENTS</td>
</tr>
<tr>
<td>ZTP Hold Down Timer</td>
<td>Stopped</td>
</tr>
<tr>
<td>Highest Available Level (HAL)</td>
<td>2</td>
</tr>
<tr>
<td>Highest Adjacency Three-way (HAT)</td>
<td>2</td>
</tr>
<tr>
<td>Level Value</td>
<td>1</td>
</tr>
<tr>
<td>Receive LIE IPv4 Multicast Address</td>
<td>224.0.0.81</td>
</tr>
<tr>
<td>Transmit LIE IPv4 Multicast Address</td>
<td>224.0.0.120</td>
</tr>
<tr>
<td>Receive LIE IPv6 Multicast Address</td>
<td>FF02::0078</td>
</tr>
<tr>
<td>Transmit LIE IPv6 Multicast Address</td>
<td>FF02::0078</td>
</tr>
<tr>
<td>Receive LIE Port</td>
<td>20102</td>
</tr>
<tr>
<td>Transmit LIE Port</td>
<td>10000</td>
</tr>
<tr>
<td>LIE Send Interval</td>
<td>1.0 secs</td>
</tr>
<tr>
<td>Receive TIE Port</td>
<td>10001</td>
</tr>
<tr>
<td>Kernel Route Table</td>
<td>3</td>
</tr>
<tr>
<td>Flooding Reduction Enabled</td>
<td>True</td>
</tr>
<tr>
<td>Flooding Reduction Redundancy</td>
<td>2</td>
</tr>
<tr>
<td>Flooding Reduction Similarity</td>
<td>2</td>
</tr>
<tr>
<td>Flooding Reduction Node Random</td>
<td>33470</td>
</tr>
</tbody>
</table>

**Sent Offers:**

<table>
<thead>
<tr>
<th>Interface</th>
<th>System ID</th>
<th>Level</th>
<th>Not A ZTP Offer</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>if_101_1</td>
<td>101</td>
<td>1</td>
<td>False</td>
<td>THREE_WAY</td>
</tr>
<tr>
<td>if_101_1001</td>
<td>1001</td>
<td>0</td>
<td>False</td>
<td>THREE_WAY</td>
</tr>
<tr>
<td>if_101_1002</td>
<td>1002</td>
<td>0</td>
<td>False</td>
<td>THREE_WAY</td>
</tr>
<tr>
<td>if_101_2</td>
<td>101</td>
<td>1</td>
<td>False</td>
<td>THREE_WAY</td>
</tr>
</tbody>
</table>

**Received Offers:**

<table>
<thead>
<tr>
<th>Interface</th>
<th>System ID</th>
<th>Level</th>
<th>Not A ZTP Offer</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>if_101_1</td>
<td>101</td>
<td>1</td>
<td>False</td>
<td>THREE_WAY</td>
</tr>
<tr>
<td>if_101_1001</td>
<td>1001</td>
<td>0</td>
<td>False</td>
<td>THREE_WAY</td>
</tr>
<tr>
<td>if_101_1002</td>
<td>1002</td>
<td>0</td>
<td>False</td>
<td>THREE_WAY</td>
</tr>
<tr>
<td>if_101_2</td>
<td>101</td>
<td>1</td>
<td>False</td>
<td>THREE_WAY</td>
</tr>
</tbody>
</table>

**Use the show interfaces (plural, with an s) to see a summary of all interfaces of the current node:**
Use the show interface `<interface>` (singular, no s) to see the details of one particular interface of the current node:

```
agg_101> show interfaces
+-----------------+-----------------+--------+--------+
| Interface Name   | Neighbor Name   | Neighbor System ID | Neighbor State |
+-----------------+-----------------+--------+--------+
| if_101_1        | core_1:if_1_101| 1      | THREE_WAY |
| if_101_1001     | edge_1001:if_1001_101| 1001 | THREE_WAY |
| if_101_1002     | edge_1002:if_1002_101| 1002 | THREE_WAY |
| if_101_2        | core_2:if_2_101| 2      | THREE_WAY |
```
Use the `show routes` command to see a summary of all routes in the Routing Information Base (RIB) of the current node:
There are many more commands available, but we are going to leave it at this for now. Later on, when we discuss chaos testing, we will provide some more hints about which commands are useful to debug or trouble-shoot issues.

Now would be a good time to experiment a little bit and try out some other commands, or see how the output of a command changes from one node to another node.

If you are a very curious and observant reader, and you happen to play around with the `show kernel routes table main` command, you may notice that the RIFT routes are not actually installed in the kernel. This is because we need elevated privileges (which we don't have) to install routes into the kernel. We will deal with this issue later; for now ignore it.

Before you move on to the next section, but sure to exit the CLI and to stop the RIFT-Python engine using the `stop` command:

```
agg_101> stop
```
**Generate the first fat tree topology**

Up until now, we have been running all RIFT routers in a single Python process. We will now switch over to a different method, which is called network namespace per node mode (netns-per-node mode).

In the netns-per-node mode, each node (i.e. each RIFT router) runs in a separate Python process in its own Linux network namespace. The links between the node are implemented using Linux virtual Ethernet (veth) pairs.

This method allows us to test larger topologies, allows us to easily mix different RIFT router implementations in the same topology (e.g. RIFT-Python and RIFT-Juniper), and gives us more flexibility to introduce perturbations into the network.

The current RIFT-Python open source implementation already includes a config_generator tool that can automatically generate configurations for large Fat Tree topologies (also known as Clos topologies). For example, you can provide the following 4 line "meta configuration" file as input to the config_generator tool and will generate a configuration file that is literally hundreds of lines long.

```
nr-pods: 3
nr-leaf-nodes-per-pod: 3
nr-spine-nodes-per-pod: 3
nr-superspine-nodes: 4
```

The RIFT-Python repository already includes a few example meta topologies in the meta_topology directory.

We will now invoke the config_generator tool to generate the topology for the clos_3pod_3leaf_3spine_4super.yaml example meta topology. Make sure you run the command from the top of our Git repository. It is normal that the command does not produce any output to the terminal (all output is to files).

```
$ cd ~/rift-python
$ tools/config_generator.py --netns-per-node meta_topology/clos_3pod_3leaf_3spine_4super.yaml generated
```

Let's look at the command-line options one by one:

- The --netns-per-node (or -n) option specifies that we want to generate a configuration for running each node in its own Python process and its own network namespace. If you don't provide this option, the generated configuration will run all nodes in a single Python process.
• The argument `meta_topology/clos_3pod_3leaf_3spine_4super.yaml` is the name of the meta configuration file.

• The argument `generated` is the name of a directory. The configuration generator will create this directory and generate a number of configuration files and bash scripts in this directory (which we will look at in detail below).

If the output directory already exists, you will get an error message:

```bash
$ tools/config_generator.py --netns-per-node meta_topology/clos_3pod_3leaf_3spine_4super.yaml generated
Output directory 'generated' already exists
```

In that case, just delete the directory (or choose a different one) and run the command again:

```bash
$ rm -rf generated  # *** WARNING *** Always be EXTREMELY careful when you run rm -rf
$ tools/config_generator.py --netns-per-node meta_topology/clos_3pod_3leaf_3spine_4super.yaml generated
```

Let's have a look at the files that the configuration generator generated (note the argument for the `ls` command is the number 1 and not the lower case letter L):

```bash
$ ls -l generated
chaos.sh   check.sh   connect-leaf-1-1.sh   connect-leaf-1-1-2.sh
...SNIP... connect-super-3.sh   connect-super-4.sh
leaf-1-1.yaml  leaf-1-2.yaml
...SNIP... spine-3-2.yaml  spine-3-3.yaml
start.sh  stop.sh  super-1.yaml  super-2.yaml  super-3.yaml  super-4.yaml
```

The configuration generator generated the following types of files:

• Configuration files with extension `.yaml`. There is one configuration file for each node in the topology.

• Shell scripts with extension `.sh`:
  - `start.sh`: Start the topology. Create all network namespaces and virtual ethernet interface pairs, and start the RIFT-Python processes.
- **stop.sh**: Stop the topology. Stop the RIFT-Python processes, and delete all virtual ethernet interface pairs and network namespaces.

- **connect-<node-name>.sh**: Start a telnet session to the specified node (it automagically knows which telnet port to use).

- **check.sh**: Check whether there is "full" connectivity in the topology. It does a both a ping and a traceroute from every leaf node to every other leaf node (using all loopback addresses). In RIFT, the leaf nodes are not expected to be able to ping the spine or super-spine nodes; that is why I put "full" in quotes.

- **chaos.sh**: This script unleashes the chaos monkeys! When you run this script it will randomly break and repair thinks (currently only links) in your network. We will discuss this script in much more detail in the chaos testing section below.

### Visualize the fat tree topology

The topologies that are generated by `config_generator` can be complex and huge. It is easy to lose track of what's going on.

Fortunately, the `config_generator` has a `--graphics-file` (or `-g`) command-line option to generate a graphical diagram of the generated network (you may have to delete the generated directory first if it still exists from a previous run):

```bash
$ tools/config_generator.py --graphics-file diagram.html --netns-per-node
meta_topology/clos_3pod_3leaf_3spine_4super.yaml generated
```

This produces the file `diagram.html`, which can be viewed using any web browser. On MacOS you can open the file in your default browser using the `open` command. On other systems you have to open the file manually in your browser. Either way, you will have to copy it from AWS to your laptop first using secure copy (`scp`).

```bash
# Replace <your-key-file> with your AWS private key file
# Replace <aws-instance-ip> with the IP address of the AWS instance that we are running RIFT-Python on
mac$ scp -i ~/.ssh/<your-key-file>.pem ubuntu@<aws-instance-ip>:rift-python/diagram.html .
diagram.html                                              100%   26KB  48.3KB/s   00:00
mac$ open diagram.html
```

Here is what a diagram looks like in the browser. Note that if you hover over a node or an interface or a link it will be highlighted. (In the future we will display useful information about the object such as addresses and statistics when you hover over it.)
Start the fat tree topology

When we start our topology, we will need root privileges (or more accurately, the CAP_NET_ADMIN capability) to create network namespaces, to install routes in the kernel, to enable IPv4 and IPv6 forwarding in each network namespace, etc. etc.

One simple way to get this is to sudo start a new bash shell. (This may not be the best way to do it, but it is simple and it works for the purpose of the hackathon.) If you are running these examples on Ubuntu 18.04 in AWS, this will work without entering any password; your mileage may vary in other environments.

```
(env) ubuntu$ sudo bash
root#
```

Since we are in a new shell, we have to re-activate our virtual environment from the root of the repository:

```
root# cd ~/rift-python
root# source env/bin/activate
(env) root#
```
Now that we are root, we can start the generated topology by invoking the `start.sh` shell script in the generated directory. This takes a few seconds and produces a lot of output:

```
(env) root# generated/start.sh
Create veth pair veth-1a-101d and veth-101d-1a for link from super-1:if-1a to spine-1:1:if-101d
Create veth pair veth-1b-102d and veth-102d-1b for link from super-1:if-1b to spine-1:2:if-102d
Create veth pair veth-1c-103d and veth-103d-1c for link from super-1:if-1c to spine-1:3:if-103d
...SNIP...
Create veth pair veth-4h-108g and veth-108g-4h for link from super-4:if-4h to spine-3:2:if-108g
Create veth pair veth-4i-109g and veth-109g-4i for link from super-4:if-4i to spine-3:3:if-109g
Create network namespace netns-1 for node super-1
Create network namespace netns-2 for node super-2
Create network namespace netns-3 for node super-3
Create network namespace netns-4 for node super-4
Create veth pair veth-1001a-101a and veth-101a-1001a for link from leaf-1:1:if-1001a to spine-1:1:if-101a
Create veth pair veth-1001b-102a and veth-102a-1001b for link from leaf-1:1:if-1001b to spine-1:2:if-102a
...SNIP...
Start RIFT-Python engine for node super-1
Start RIFT-Python engine for node super-2
Start RIFT-Python engine for node super-3
...SNIP...
Start RIFT-Python engine for node spine-3-1
Start RIFT-Python engine for node spine-3-2
Start RIFT-Python engine for node spine-3-3
```

If the generated topology is a file instead of a directory it means you forgot the `--netns-per-node` or `-n` option when you invoked `config_generator`.

If you get errors about operations not being permitted, it means you did not run the script with root privileges.

After the start script completes, you end up back in the shell. All the RIFT-Python routers are running as background daemons in their own network namespaces.

**Connect to a node in the fat tree topology**

To connect to the CLI of a particular RIFT router (which is now running in the background), use one of the generated `connect-<node-name>.sh` convenience scripts. Behind the scenes, this script goes into the correct network namespace and telnets to the correct port. For example, use the following command to connect to router spine-1-1:

```
(env) root# generated/connect-spine-1-1.sh
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
spine-1-1>
```

Now, you can execute commands for that node as normal. For example, use the `show tie-db` command to look at the contents for the TIE database (also known as the link state database):
You will notice that in netns-per-node mode each RIFT engine contains only one node:

If you want to access the CLI of a different router, exit out of the current engine and connect to a different engine. (Make sure you use exit and not stop; the latter terminates the engine.)

**Validate leaf-to-leaf connectivity in the fat tree**

The configuration generator also generates a check.sh script. This script issues both a ping and a traceroute from every leaf node to every other leaf node. If the leaf nodes have multiple loopback interfaces, it tries all combinations of loopback interfaces.
Here is what the output looks like if all pings are successful:

```
(env) root# generated/check.sh
*** ping ***
OK: ping leaf-1-1 88.0.1.1 -> leaf-1-2 88.0.2.1
OK: ping leaf-1-1 88.0.1.1 -> leaf-1-3 88.0.3.1
OK: ping leaf-1-1 88.0.1.1 -> leaf-2-1 88.0.4.1
...SNIP...
OK: ping leaf-3-3 88.0.9.1 -> leaf-3-1 88.0.7.1
OK: ping leaf-3-3 88.0.9.1 -> leaf-3-2 88.0.8.1
*** traceroute ***
traceroute leaf-1-1 88.0.1.1 -> leaf-1-2 88.0.2.1
traceroute to 88.0.2.1 (88.0.2.1), 5 hops max, 60 byte packets
 1 99.5.6.6  0.014 ms  0.004 ms  0.003 ms
 2 88.0.2.1  0.010 ms  0.005 ms  0.006 ms
traceroute leaf-1-1 88.0.1.1 -> leaf-1-3 88.0.3.1
traceroute to 88.0.3.1 (88.0.3.1), 5 hops max, 60 byte packets
 1 99.5.6.6  0.011 ms  0.004 ms  0.003 ms
 2 88.0.3.1  0.011 ms  0.005 ms  0.004 ms
...SNIP...
traceroute leaf-3-3 88.0.9.1 -> leaf-2-3 88.0.6.1
traceroute to 88.0.6.1 (88.0.6.1), 5 hops max, 60 byte packets
 1 99.53.54.54  0.012 ms  0.004 ms *
 2 99.87.88.87  0.007 ms  0.005 ms *
 3  ***
 4  ***
 5  ***
traceroute leaf-3-3 88.0.9.1 -> leaf-3-1 88.0.7.1
traceroute to 88.0.7.1 (88.0.7.1), 5 hops max, 60 byte packets
 1 99.51.52.52  0.028 ms  0.004 ms  0.004 ms
 2 88.0.7.1  0.012 ms  0.005 ms  0.004 ms
traceroute leaf-3-3 88.0.9.1 -> leaf-3-2 88.0.8.1
traceroute to 88.0.8.1 (88.0.8.1), 5 hops max, 60 byte packets
 1 99.53.54.54  0.012 ms  0.004 ms  0.003 ms
 2 88.0.8.1  0.011 ms  0.005 ms  0.005 ms
Number of failures: 0
```

Note: by the time you read this, I may have already added an command-line option to the check command to only do the traceroutes if you explicitly ask for them (they are slow).

Here is what the output looks like if at least one ping failed:

---

Page 21 of 39
(env) root# generated/check.sh

*** ping ***
OK: ping leaf-1-1 88.0.1.1 -> leaf-1-2 88.0.2.1
OK: ping leaf-1-1 88.0.1.1 -> leaf-1-3 88.0.3.1
OK: ping leaf-1-1 88.0.1.1 -> leaf-2-1 88.0.4.1
OK: ping leaf-1-1 88.0.1.1 -> leaf-2-2 88.0.5.1
FAIL: ping leaf-1-1 88.0.1.1 -> leaf-2-3 88.0.6.1
...SNIP...
OK: ping leaf-3-3 88.0.9.1 -> leaf-3-1 88.0.7.1
OK: ping leaf-3-3 88.0.9.1 -> leaf-3-2 88.0.8.1

*** traceroute ***
traceroute to leaf-1-1 88.0.1.1 (88.0.1.1), 5 hops max, 60 byte packets
  1  99.5.6.6  0.068 ms 99.1.2.2  0.071 ms 99.5.6.6  0.025 ms
  2  99.59.60.59  0.097 ms 88.0.2.1  0.088 ms  0.088 ms
...SNIP...
traceroute to leaf-1-1 88.0.1.1 (88.0.1.1), 5 hops max, 60 byte packets
  1  99.5.6.6  0.157 ms * *
  2  99.111.112.111  0.161 ms  0.148 ms  0.158 ms
  3  * * *
  4  * * *
  5  * * *
...SNIP...
traceroute to leaf-3-3 88.0.9.1 (88.0.9.1), 5 hops max, 60 byte packets
  1  * *  99.49.50.50  0.052 ms
  2  88.0.8.1  0.098 ms  0.079 ms  0.072 ms

Number of failures: 15

*** THERE WERE FAILURES ***

The check script only checks leaf-node to leaf node connectivity; it does not attempt to ping the spine nodes nor the super-spine nodes. This is because in RIFT you typically cannot ping the loopback address of an ancestor node (i.e. any node at a higher level). The reason for this is that the nodes at lower levels only have default routes and not more specific routes to the loopbacks for the nodes at higher levels. This has been discussed at length in on the RIFT core architecture mailing list.
Unleash the chaos monkeys!

There is only one remaining script that we have not yet discussed: the chaos.sh script. This is where the chaos monkeys live.

When you run the script, it will start causing random events in the network. By default, the script will generate 10 events and there will be 5 seconds between events. But these defaults can be changed using the following configuration parameters in the meta configuration file:

```
chaos:
  nr-events: 20          # Generate 20 events
  event-interval: 10.0    # 10.0 seconds between events
```

Each event will either break something or repair something that was broken by a previous event.

There is currently only one type of breakage event, namely full bi-directional failure of a link. In the future, I plan to add other types of failure events, such as uni-directional link failures, partial link failures (drop some percentage of packets), packet re-ordering, packet duplication, node failures, etc.

The chaos monkeys are not particularly careful. They may cause more damage than RIFT can handle. For example they may partition the network. Thus, we cannot assume that all leaf-to-leaf pings should continue to work during a chaos test. In the future I may add a configuration parameter for more considerate monkeys that limit their damage to some maximum. That way we can test whether all failure scenarios that RIFT is supposed to handle, are actually handled correctly.

If there are still any things broken by the end of the test run, the script will repair all remaining broken things (without any delay in between). This has an important implication! When the script terminates, nothing is broken anymore; everything has been repaired. This means that the network should reconverge to fully operational status even if it was partitioned somewhere in the middle of the test. Thus, if the check.sh script passes before a chaos test run, it should also pass after a chaos test run. If not, something has become "stuck" in the RIFT protocol or the implementation.

There is one problem with random testing: if the test finds a problem, it may be difficult to reproduce the problem because of the random nature of the testing. But we don't have that problem here! The randomness is only in the generation of the chaos.sh script. Each time you run config_generator tool on the same meta topology, a different chaos.sh script will be generated. But once a chaos.sh script has been generated, you can run it as many times as you want. Every time you run it, it will run the exact same sequence of breakage and repair.
events with the exact same timing. This makes it easier to reproduce problems once you find them. You can even copy and paste the chaos.sh script into a bug report.

Once you discover a problem and once you are able to reproduce it, you may need extra information to get to the root cause of it. You might need to stop the script at a specific point (the equivalent of a breakpoint in programming), so that you can connect to the RIFT routers and investigate their state. You might want to execute show commands at specific points in the script. You might want to run specific commands such as ping or traceroute or tcpdump or ip at specific points in the script. All of this is trivial! The generated chaos.sh script is just a normal shell script and you can edit it as you see fit.

Enough talk. Let's start flinging some excrement and look at the output of a chaos.sh script. This script takes some time to run (there is a 5 second delay between each line):

```
$ generated/chaos.sh
Bring link if-2h-if-108e down (bi-directional failure)
Bring link if-2f-if-106e down (bi-directional failure)
Bring link if-1006a-if-104c down (bi-directional failure)
Bring link if-1007b-if-108a down (bi-directional failure)
Bring link if-3e-if-105f down (bi-directional failure)
Bring link if-1001b-if-102a down (bi-directional failure)
Bring link if-3i-if-109f down (bi-directional failure)
Bring link if-2d-if-104e down (bi-directional failure)
Bring link if-1001c-if-103a down (bi-directional failure)
Bring link if-1003a-if-101c down (bi-directional failure)
Bring link if-2h-if-108e up
Bring link if-2d-if-104e up
Bring link if-3e-if-105f up
Bring link if-1001c-if-103a up
Bring link if-2f-if-106e up
Bring link if-1003a-if-101c up
Bring link if-3i-if-109f up
Bring link if-1005a-if-104c up
Bring link if-1001b-if-102a up
Bring link if-1007b-if-108a up
$
```

After the chaos.sh script completed, we can run the check.sh script again to make sure everything reconverged:
Yay! All is good. The network did reconverge properly.

Trouble-shooting tools

I have spent a lot of time using the RIFT-Python CLI and using bash shell commands to debug various issues that I discovered while implementing RIFT-Python. In the following sections I will share the various commands and techniques that I found useful to debug problems. In fact many of the commands and techniques were introduced to debug specific issues.

Trouble-shooting tool: the routing table, the forwarding table, and the kernel

If a ping is not working, the first step is to look at the various routing tables.

Use show routes to see the routes that RIFT calculated in its Routing Information Base (RIB).
You can also look at a specific route using `show routes prefix <prefix>`:

```
spine-1-1> show routes prefix 0.0.0.0/0
  Prefix | Owner     | Next-hops
+--------+-----------+------------------
| 0.0.0.0/0 | North SPF | veth-101d-1a 99.55.56.55 |
|         |           | veth-101e-2a 99.73.74.73 |
|         |           | veth-101f-3a 99.91.92.91 |
|         |           | veth-101g-4a 99.109.110.109 |
```

Use `show forwarding` to see the routes that RIFT calculated in its Forwarding Information Base (FIB). This is the best route for a given prefix, and the one that RIFT will install in the kernel route table.

Use `show kernel routes table main` to see the routes that RIFT actually did install in the kernel route table, but it also include kernel routes that come from other protocols than RIFT (e.g. kernel routes):
This should produce the exact same information (just in a different format) as the `ip route list` shell command (which you need to execute in the context of the correct namespace using the `ip netns exec` shell command).

```
$ # ip netns exec netns-101 ip route list
default proto 99
    next hop via 99.91.92.91 dev veth-101f-3a weight 2
    next hop via 99.55.56.55 dev veth-101d-1a weight 2
    next hop via 99.109.110.109 dev veth-101g-4a weight 2
    next hop via 99.73.74.73 dev veth-101e-2a weight 2
88.0.1.1 via 99.1.2.1 dev veth-101a-1001a proto 99
88.0.2.1 via 99.7.8.7 dev veth-101b-1002a proto 99
88.0.3.1 via 99.13.14.13 dev veth-101c-1003a proto 99
99.1.2.0/24 dev veth-101a-1001a proto kernel scope link src 99.1.2.2
99.7.8.0/24 dev veth-101b-1002a proto kernel scope link src 99.7.8.7
99.55.56.0/24 dev veth-101d-1a proto kernel scope link src 99.55.56.55
99.109.110.0/24 dev veth-101g-4a proto kernel scope link src 99.109.110.109
99.73.74.73 dev veth-101e-2a proto kernel scope link src 99.73.74.73
99.109.110.0/24 dev veth-101g-4a proto kernel scope link src 99.109.110.110
```

If the routes that you were expecting are not there, you can start working backwards to find out why there are not there.

**Trouble-shooting tool: shortest path first calculations**

The first thing to look at is the details of the most recent Shortest Path First (SPF) (also known as Dijkstra) run.
spine-1-l> show spf
SPF Statistics:
+-----------------+
| SPF Runs | 36 |
+-----------------+
| SPF Deferrals | 76 |
+-----------------+

South SPF Destinations:
+-----------------+
| Destination | Cost | Predecessor | Tags | IPv4 Next-hops | IPv6 Next-hops |
+-----------------+
| 101 (spine-1-1) | 0 | | | |
+-----------------+
| 1001 (leaf-1-1) | 1 | 101 | | veth-101a-1001a 99.1.2.1 | veth-101a-1001a fe80::24ec:55ff:fe03:4ae7 |
+-----------------+
| 88.1.1.1/32 | 1 | 101 | | |
+-----------------+

North SPF Destinations:
+-----------------+
| Destination | Cost | Predecessor | Tags | IPv4 Next-hops | IPv6 Next-hops |
+-----------------+
| 1 (super-1) | 1 | 101 | | veth-101d-1a 99.55.56.55 | veth-101d-1a fe80::28b2:ffff:fe93:b9b7 |
+-----------------+
| ::/0 | 2 | 1 | veth-101d-1a 99.55.56.55 | veth-101d-1a fe80::28b2:ffff:fe93:b9b7 |
| | 2 | veth-101e-2a 99.73.74.73 | veth-101e-2a fe80::283d:e3ff:fe28:c74a |
| | 3 | veth-101f-3a 99.91.92.91 | veth-101f-3a fe80::e071:96ff:fe41:a0f2 |
| | 4 | veth-101g-4a 99.109.110.109 | veth-101g-4a fe80::3c98:4bff:fe6c:754b |
+-----------------+

Trouble-shooting tool: the link state database

If some node or some prefix is missing from the SPF run, it might be because it is not in the link-state database. Thus, our next stop is the show tie-db command. (tie-db stands for Topology Information Element - Data Base):
Right now the command is "all or nothing" but I plan to add variations of the `show tie-db` command to look at specific subsets of information.

Once you discover that a TIE is missing from the link state database, the next question is: why it is not there? There are a number of tools to help you answer that question.

**Trouble-shooting tool: send queues**

If there is a problem with link state flooding or with link state database synchronization, the problem often manifests itself with stuck entries in one of the queues. Use the `show interface <interface> queues` command to view the queues. In this example all queues are empty (which is good - after everything has converged they should be empty).
Trouble-shooting tool: flooding reduction

Sometimes unexpected behavior of flooding-reduction can be the cause of convergence problems. Use the show flooding-reduction command to look at the details of the flood repeater (FR) election. Here I show the output for a leaf node because the election process is more interesting for leaves (spines don't have grandparents):
leaf-1-1# show flooding-reduction

Parents:

<table>
<thead>
<tr>
<th>Name</th>
<th>Interface</th>
<th>System ID</th>
<th>State</th>
<th>Direction</th>
<th>Flood Repeater</th>
<th>Flood Repeater</th>
</tr>
</thead>
<tbody>
<tr>
<td>veth-1001a-101a</td>
<td>101</td>
<td>spine-1-1:veth-101a-1001a</td>
<td>4</td>
<td>1: 4-4</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>veth-1001c-103a</td>
<td>103</td>
<td>spine-1-3:veth-103a-1001c</td>
<td>4</td>
<td>1: 4-4</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>veth-1001b-102a</td>
<td>102</td>
<td>spine-1-2:veth-102a-1001b</td>
<td>4</td>
<td>1: 4-4</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

Grandparents:

<table>
<thead>
<tr>
<th>Name</th>
<th>System ID</th>
<th>Parent</th>
<th>Count</th>
<th>Grandparent</th>
<th>Flood Repeater</th>
<th>Redundantly</th>
<th>Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>veth-1001a-101a</td>
<td>101</td>
<td>spine-1-1:veth-101a-1001a</td>
<td>4</td>
<td>1: 4-4</td>
<td>True</td>
<td></td>
<td></td>
</tr>
<tr>
<td>veth-1001c-103a</td>
<td>103</td>
<td>spine-1-3:veth-103a-1001c</td>
<td>4</td>
<td>1: 4-4</td>
<td>True</td>
<td></td>
<td></td>
</tr>
<tr>
<td>veth-1001b-102a</td>
<td>102</td>
<td>spine-1-2:veth-102a-1001b</td>
<td>4</td>
<td>1: 4-4</td>
<td>False</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interfaces:

<table>
<thead>
<tr>
<th>Name</th>
<th>Neighbor</th>
<th>Neighbor</th>
<th>Neighbor</th>
<th>Neighbor is</th>
<th>This Node is</th>
<th>Flood Repeater</th>
<th>Flood Repeater</th>
</tr>
</thead>
<tbody>
<tr>
<td>veth-1001a-101a</td>
<td>spine-1-1:veth-101a-1001a</td>
<td>101</td>
<td>THREE_WAY</td>
<td>North</td>
<td>True</td>
<td>Not Applicable</td>
<td></td>
</tr>
<tr>
<td>veth-1001b-102a</td>
<td>spine-1-2:veth-102a-1001b</td>
<td>102</td>
<td>THREE_WAY</td>
<td>North</td>
<td>False</td>
<td>Not Applicable</td>
<td></td>
</tr>
<tr>
<td>veth-1001c-103a</td>
<td>spine-1-3:veth-103a-1001c</td>
<td>103</td>
<td>THREE_WAY</td>
<td>North</td>
<td>True</td>
<td>Not Applicable</td>
<td></td>
</tr>
</tbody>
</table>

Trouble-shooting tool: adjacencies

Another common reason for nodes being unreachable is that their adjacencies did not come up. Use the show interfaces command to see a summary of all interfaces and their adjacency state. In this example all interfaces are THREE_WAY which is good.
Use the command `show interface <interface>` to see the details of a particular interface:

```
spine-1-1> show interfaces
+-----------------+-------------------------------+-----------------+-----------------+
| Interface Name   | Neighbor Name                 | Neighbor System ID | Neighbor State   |
+-----------------+-------------------------------+-----------------+-----------------+
| veth-101a-1001a | leaf-1-1:veth-1001a-101a     | 1001             | THREE_WAY       |
| veth-101b-1002a | leaf-1-2:veth-1002a-101b     | 1002             | THREE_WAY       |
| veth-101c-1003a | leaf-1-3:veth-1003a-101c     | 1003             | THREE_WAY       |
| veth-101d-1a    | super-1:veth-1a-101d         | 1                | THREE_WAY       |
| veth-101e-2a    | super-2:veth-2a-101e         | 2                | THREE_WAY       |
| veth-101f-3a    | super-3:veth-3a-101f         | 3                | THREE_WAY       |
| veth-101g-4a    | super-4:veth-4a-101g         | 4                | THREE_WAY       |
+-----------------+-------------------------------+-----------------+-----------------+
```
spine-1-1> show interface veth-101a-1001a

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>veth-101a-1001a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Interface Name</td>
<td>veth-101a-1001a</td>
</tr>
<tr>
<td>Advertised Name</td>
<td>spine-1-1:veth-101a-1001a</td>
</tr>
<tr>
<td>Interface IPv4 Address</td>
<td>99.1.2.2</td>
</tr>
<tr>
<td>Interface IPv6 Address</td>
<td>fe80::9cb9:dff:fe79:9459%veth-101a-1001a</td>
</tr>
<tr>
<td>Interface Index</td>
<td>75</td>
</tr>
<tr>
<td>Metric</td>
<td>100</td>
</tr>
<tr>
<td>LIE Receive IPv4 Multicast Address</td>
<td>224.0.0.120</td>
</tr>
<tr>
<td>LIE Receive IPv6 Multicast Address</td>
<td>FF02::0078</td>
</tr>
<tr>
<td>LIE Transmit IPv4 Multicast Address</td>
<td>224.0.0.120</td>
</tr>
<tr>
<td>LIE Transmit IPv6 Multicast Address</td>
<td>FF02::0078</td>
</tr>
<tr>
<td>LIE Transmit Port</td>
<td>10000</td>
</tr>
<tr>
<td>Flooding Receive Port</td>
<td>10001</td>
</tr>
<tr>
<td>System ID</td>
<td>101</td>
</tr>
<tr>
<td>Local ID</td>
<td>1</td>
</tr>
<tr>
<td>MTU</td>
<td>1400</td>
</tr>
<tr>
<td>POD</td>
<td>0</td>
</tr>
<tr>
<td>Failure</td>
<td>ok</td>
</tr>
<tr>
<td>State</td>
<td>THREE_WAY</td>
</tr>
<tr>
<td>Received LIE Accepted or Rejected</td>
<td>Accepted</td>
</tr>
<tr>
<td>Received LIE Accept or Reject Reason</td>
<td>This node is not leaf and neighbor is leaf</td>
</tr>
<tr>
<td>Neighbor is Flood Repeater</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>

Neighbor:

<table>
<thead>
<tr>
<th>Name</th>
<th>leaf-1-1:veth-1001a-101a</th>
</tr>
</thead>
<tbody>
<tr>
<td>System ID</td>
<td>1001</td>
</tr>
<tr>
<td>IPv4 Address</td>
<td>99.1.2.1</td>
</tr>
<tr>
<td>IPv6 Address</td>
<td>fe80::24ec:55ff:fe03:4ae7%veth-101a-1001a</td>
</tr>
<tr>
<td>LIE UDP Source Port</td>
<td>52347</td>
</tr>
<tr>
<td>Link ID</td>
<td>1</td>
</tr>
<tr>
<td>Level</td>
<td>0</td>
</tr>
<tr>
<td>Flood UDP Port</td>
<td>10001</td>
</tr>
<tr>
<td>MTU</td>
<td>1400</td>
</tr>
<tr>
<td>POD</td>
<td>0</td>
</tr>
<tr>
<td>Hold Time</td>
<td>3</td>
</tr>
<tr>
<td>Not a ZTP Offer</td>
<td>False</td>
</tr>
<tr>
<td>You are Flood Repeater</td>
<td>True</td>
</tr>
<tr>
<td>Your System ID</td>
<td>101</td>
</tr>
<tr>
<td>Your Local ID</td>
<td>1</td>
</tr>
</tbody>
</table>

Trouble-shooting tool: statistics

You can view statistics at the interface level, at the node level, and at the engine level. Some statistics that are particularly helpful for debugging problems:

- Statistics about the number and rate of packets sent and received (split out by type of packet and by address family, i.e. IPv4 or IPv6).
- Statistics about the number and rate of events and transitions in the finite state machines (the LIE FSM is part of the interface statistics, and the ZTP FSM is part of the node statistics).
Use the show interface <interface> statistics command to view the interface statistics (only a small subset is shown here):

```plaintext
spine-1-1> show interface veth-101a-1001a statistics
Traffic:
<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
<th>Last Rate</th>
<th>Last Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX IPv4 LIE Packets</td>
<td>5436 Packets, 886068 Bytes</td>
<td>1.00 Packets/Sec, 163.00 Bytes/Sec</td>
<td>0d 00h:00m:00.85s</td>
</tr>
<tr>
<td>TX IPv4 LIE Packets</td>
<td>5437 Packets, 891646 Bytes</td>
<td>1.00 Packets/Sec, 164.00 Bytes/Sec</td>
<td>0d 00h:00m:00.85s</td>
</tr>
<tr>
<td>Total RX Packets</td>
<td>13615 Packets, 2933114 Bytes</td>
<td>3.00 Packets/Sec, 663.61 Bytes/Sec</td>
<td>0d 00h:00m:00.86s</td>
</tr>
<tr>
<td>Total TX Packets</td>
<td>13622 Packets, 4333459 Bytes</td>
<td>3.00 Packets/Sec, 1005.67 Bytes/Sec</td>
<td>0d 00h:00m:00.85s</td>
</tr>
<tr>
<td>Total RX Errors</td>
<td>0 Packets, 0 Bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total TX Errors</td>
<td>0 Packets, 0 Bytes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LIE FSM:
<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
<th>Last Rate</th>
<th>Last Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Transitions None -[None]-&gt; ONE_WAY</td>
<td>1 Transition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event-Transitions ONE_WAY -[LIE_RECEIVED]-&gt; ONE_WAY</td>
<td>1 Transition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event-Transitions ONE_WAY -[NEW_NEIGHBOR]-&gt; TWO_WAY</td>
<td>1 Transition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events SEND_LIE</td>
<td>5436 Events</td>
<td>1.00 Events/Sec</td>
<td>0d 00h:00m:00.85s</td>
</tr>
<tr>
<td>Events TIMER_TICK</td>
<td>5435 Events</td>
<td>1.00 Events/Sec</td>
<td>0d 00h:00m:00.85s</td>
</tr>
<tr>
<td>Events UNACCEPTABLE_HEADER</td>
<td>0 Events</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are lots and lots and lots of statistics. You can use the exclude-zero option to skip displaying the counters which are zero. You can use the clear ... statistics command to reset the counters back to zero.

Trouble-shooting tool: logs

Currently, all RIFT-Python nodes are hard-coded to write their logs to the `rift.log` file. If there are multiple RIFT routers, they all write their logs to the same file, so the file will contain all logs from all routers, properly ordered in the order in which the logs occurred.

Here is an example contents of the log (this is a very tiny sample):

```
...SNIP...
event=None actions-and-pushed-events=stop_hold_down_timer,level_compute,COMPUTATION_DONE to-
state=COMPUTE_BEST_OFFER implicit=False
2019-02-24 20:58:30.823:WARNING:node.kernel:[edge_1001] Kernel networking is not supported on this
platform
2019-02-24 20:58:30.836:INFO:node.if.fsm:[edge_1001:if_1001_102] FSM transition sequence-nr=28 from-
state=None event=None actions-and-pushed-events=cleanup,send_lie to-state=ONE_WAY implicit=False
...SNIP...
```

By default, all nodes only write logs of severity INFO and higher to the log (i.e. not DEBUG logs). You can change this using the `--log-level` command line option for the `rift` package:

```
$ python rift --log-level debug topology/two_by_two_by_two.yaml
Command Line Interface (CLI) available on port 59136
```

Currently changing the level is an all or nothing affair. In the future I will introduce a more
sophisticated mechanism to allow changing the logging level for specific categories (RX
packets, TX packets, SPF runs, ...) and for specific objects (nodes, interfaces, ...).

Until then, the logs are crude tool. If you turn on DEBUG logging for more than a couple of
nodes, the system will slow to a crawl.

For now, the most useful thing to do with logs in a large topology is to look for WARNING,
ERROR, and CRITICAL messages, which are often a symptom of trouble (and if not, they
should be reduced in severity to avoid false alarms):
Trouble-shooting tool: log visualization

While logs are not very helpful in debugging large topologies, they are an invaluable tool for debugging the RIFT protocol design itself and the detailed behavior of the implementation in very small topologies with just a few nodes.

If you have just 2 or 3 or 4 or maybe 5 nodes, you can turn the debug severity to DEBUG on all nodes. This will capture every little thing that happens, including every sent and receive message, every FSM event and transition, every SPF run, every FR election etc. etc.

Even in these small topologies, the logs tend to become so huge that you quickly lose the forest for the trees. For that reason, I have implemented a visualization tool that converts the log text file into a graphical diagram that show the timeline of events in a graphical manner.

Use the rift/visualize_log tool to convert a log file into a graphical representation (rift.log.html) that you can view using a browser.

(Note: the visualize_log tool is located in the rift direction because of some Python issues, but will soon move back to the tools directory once those issues are fixed.)

An example diagram is shown below. This is a small slice; you can scroll up and down and left and right. The vertical black lines represent nodes and interfaces. The colored bubbles represent events. And the diagonal blue lines represent various types of messages.
There are some checkboxes at the top of the diagram to allow you to filter out or display specific categories of information (this is zoomed in a bit more than the above screenshot):
Stop the fat tree topology

Now that we are done with our testing, we can use the `stop.sh` script to stop the entire topology. This kills all the RIFT-Python processes, deletes all the virtual ethernet interfaces, and deletes all the namespaces.

```
(env) root# generated/stop.sh
Stop RIFT-Python engine for node super-1
Delete interface veth-1a-101d for node super-1
Delete interface veth-1b-102d for node super-1
...SNIP...
Delete interface veth-109e-2i for node spine-3-3
Delete interface veth-109f-3i for node spine-3-3
Delete interface veth-109g-4i for node spine-3-3
Delete network namespace netns-1 for node super-1
Delete network namespace netns-2 for node super-2
...SNIP...
Delete network namespace netns-108 for node spine-3-2
Delete network namespace netns-109 for node spine-3-3
```

If things have become a mess (e.g. because of bugs or crashes), it is often easiest to simply stop and restart the whole topology.

I have noticed that if you destroy and recreate name spaces and virtual ethernet interfaces many times, thinks get "wonky" after a couple of cycles. Restart the AWS instance as a work-around. I have not yet found the root cause for this issue.

Install Juniper RIFT

So far I have only discussed running RIFT-Python routers in the topology. I also want to include RIFT-Juniper routers. My gut feeling is that it should be trivial to run RIFT-Juniper in the same topology, since it is just a matter of running RIFT-Juniper in a network namespace and configuring it to use the veth interfaces. But I have not made the changes to `config_generator` tool to make it easy to start a mixture of RIFT-Python and RIFT-Juniper nodes (similar to what I did for the `tests/interop.py` automated unit testing script).

I may get around to doing this before the hackathon, or we can do it together at the hackathon.

Thank you and goodbye

Thank you for sticking with it and making it to the end of this guide. I looking forward to seeing all of you at the hackathon,

Bruno Rijsman