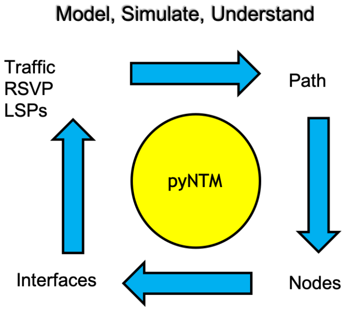
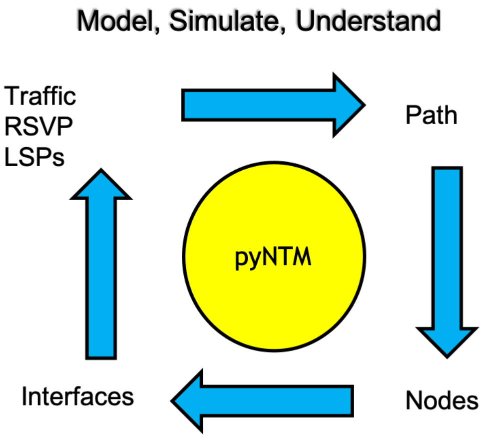
pyNTM Training Module 4 – RSVP LSPs and Shared Risk Link Groups (SRLGs)

Network Traffic Modeler in Python3



Course Topics

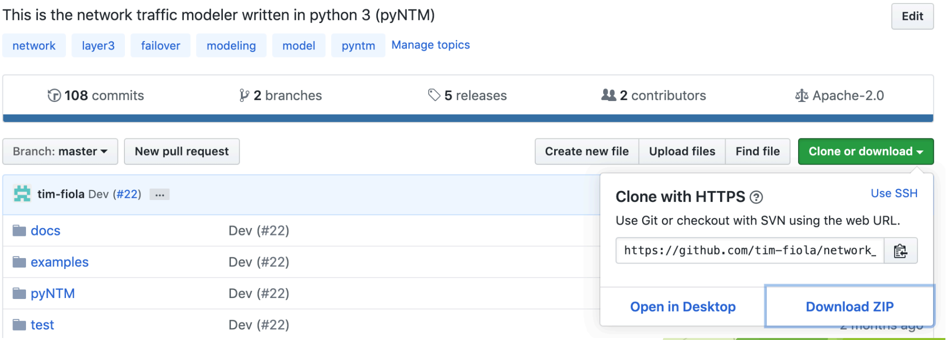
- RSVP LSP model data files
- RSVP types and behaviors
  - Auto bandwidth
  - Fixed bandwidth
  - LSPs and Demands
- Getting an LSP path
- Seeing demands on an LSP
- Demand path when demand is on LSP
- Shared Risk Link Groups (SRLGs)
- Adding an SRLG
- Failing an SRLG

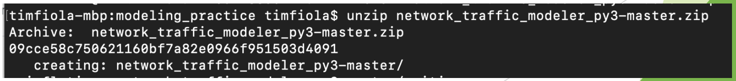


Exercise setup

Copy the repository zip file to a practice directory and unzip it

- Copying the repository will allow you to use some of the additional tools to improve your user experience
  - Visualization
  - Simple user interface





Set up your virtual environment (optional)

- Go into the archive directory
  - Look for *requirements.txt*
- Follow directions below to create the virtual environment
- Example is to the right →

A virtual environment provides an isolated environment and ensures no interference from existing installations and/or dependencies

```
[timfiola-mbp:modeling_practice timfiola$ cd network_traffic_modeler_py3-master
[timfiola-mbp:network_traffic_modeler_py3-master timfiola$ ls -lt
total 80
-rwxr-xr-x@  1 timfiola  935  11306 Nov 13 12:20 LICENSE
-rwxr-xr-x@  1 timfiola  935     25 Nov 13 12:20 Manifest.in
-rwxr-xr-x@  1 timfiola  935   1772 Nov 13 12:20 README.md
-rwxr-xr-x@  1 timfiola  935   3087 Nov 13 12:20 TODO.md
drwxr-xr-x@ 10 timfiola  935    320 Nov 13 12:20 docs
drwxr-xr-x@ 10 timfiola  935    320 Nov 13 12:20 examples
drwxr-xr-x@ 12 timfiola  935    384 Nov 13 12:20 pyNTM
-rwxr-xr-x@  1 timfiola  935     32 Nov 13 12:20 requirements.txt
-rwxr-xr-x@  1 timfiola  935     87 Nov 13 12:20 requirements_dev.txt
-rwxr-xr-x@  1 timfiola  935    344 Nov 13 12:20 setup.cfg
-rwxr-xr-x@  1 timfiola  935    927 Nov 13 12:20 setup.py
drwxr-xr-x@ 25 timfiola  935    800 Nov 13 12:20 test
[timfiola-mbp:network_traffic_modeler_py3-master timfiola$
[timfiola-mbp:network_traffic_modeler_py3-master timfiola$ virtualenv -p python3 venv
```

Create your virtualenv

Create an isolated virtual environment under the directory "venv" with python3:

```
$ virtualenv -p python3 venv
```

Activate "venv" that sets up the required env variables:

```
$ source venv/bin/activate
```

Install required packages with "pip":

```
$ pip install -r requirements.txt
```

```
[timfiola-mbp:network_traffic_modeler_py3-master timfiola$ source venv/bin/activate
[(venv) timfiola-mbp:network_traffic_modeler_py3-master timfiola$
[(venv) timfiola-mbp:network_traffic_modeler_py3-master timfiola$ pip install -r requirements.txt
Collecting networkx
```

About the model file

- Tab separated
- RSVP_LSP_TABLE has 4 columns
  - source – LSP source node
  - dest – LSP destination node
  - name – name of LSP
  - configured_setup_bw
    - If fixed bandwidth reservation LSP, populate a value for the configured_setup_bw
    - If auto-bandwidth LSP, leave blank

```
INTERFACES_TABLE
node_object_name	remote_node_object_name	name	cost	capacity
A	B	A-to-B	20	125
B	A	B-to-A	20	125
B	D	B-to-D	20	125
D	B	D-to-B	20	125
A	C	A-to-C	30	150
C	A	C-to-A	30	150
D	C	D-to-C	30	150
C	D	C-to-D	30	150
A	E	A-to-E	10	300
E	A	E-to-A	10	300
D	F	D-to-F	10	300
F	D	F-to-D	10	300
A	D	A-to-D	40	20
D	A	D-to-A	40	20
B	G	B-to-G	10	100
G	D	G-to-D	10	100
G	B	G-to-B	10	100
D	G	D-to-G	10	100

NODES_TABLE
name	lon	lat
A	50	0
B	0	-50
C	0	50
D	-50	0
E	75	0
F	-75	0
G	30	30

DEMANDS_TABLE
source	dest	traffic	name
A	D	80	dmd_a_d_1
A	D	70	dmd_a_d_2
F	E	400	dmd_f_e_1
A	F	40	dmd_a_f_1

RSVP_LSP_TABLE
source	dest	name	configured_setup_bw
A	D	lsp_a_d_1
A	D	lsp_a_d_2
F	E	lsp_f_e_1
```

Let's get started! (RSVP)

- Switch to the *examples* directory in the repository
- Start python3
- Append parent directory to your sys path
  - Allows imports from folders in the parent
  - Import the Model object
- Load Model from data file
  - *lsp_model_test_file.csv* has Interfaces, Nodes, Demands, and RSVP LSPs
- Observe model file

```
[>>> import sys
[>>> sys.path.append('../')
```

```
[>>> model2 = Model.load_model_file('lsp_model_test_file.csv')
[>>>
[>>> model2.update_simulation()
Routing the LSPs . . .
Routing 2 LSPs in parallel LSP group A-D; 1/2
Routing 1 LSPs in parallel LSP group F-E; 2/2
LSPs routed (if present); routing demands now . . .
Demands routed; validating model . . .
```

```
>>> model2
Model(Interfaces: 18, Nodes: 7, Demands: 4, RSVP_LSPs: 3)
>>>
```

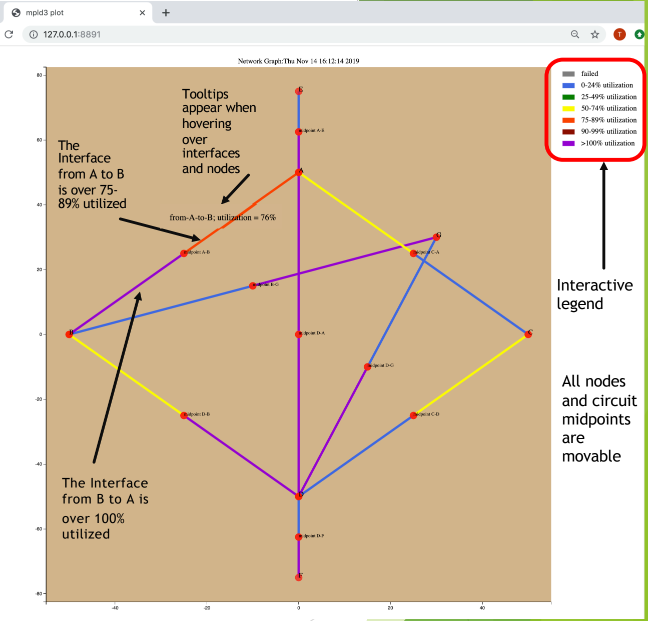
Visualization (beta) - optional

- Requires full repository download from github or extract the module
  - Easy access via the virtual environment setup from earlier in this guide
  - Reference the *Exercise setup* earlier in this presentation for full instructions to set up the environment to use the repository
- Make sure the model is converged!
  - *model2.update_simulation()* ← *model2* is the Model object
- *graph_network_interactive.make_interactive_network_graph* call
  - Takes Model object as argument
  - uses mpld3 python package under the covers
- Produces interactive graph in browser with tool tips, an interactive legend, and draggable Nodes and Interface endpoints for easier viewing
- Uses a Node's lat/lon (y,x) attributes to position Node on layout

```
>>> model2.update_simulation()
Routing the LSPs . . .
Routing 2 LSPs in parallel LSP group A-D; 1/2
Routing 1 LSPs in parallel LSP group F-E; 2/2
LSPs routed (if present); routing demands now . . .
Demands routed; validating model . . .
>>>
```

```
>>> from graph_network import graph_network_interactive
```

```
>>> graph_network_interactive.make_interactive_network_graph(model2)
>>> Serving to http://127.0.0.1:8891/    [Ctrl-C to exit]
127.0.0.1 - - [14/Nov/2019 16:12:15] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [14/Nov/2019 16:12:15] "GET /d3.js HTTP/1.1" 200 -
127.0.0.1 - - [14/Nov/2019 16:12:15] "GET /mpld3.js HTTP/1.1" 200 -
```



RSVP LSP types and behaviors

- pyNTM supports two RSVP LSP types
- Auto-bandwidth
  - Attempts to adjust signaled bandwidth and path to accommodate the amount of traffic on the LSP
- Fixed bandwidth
  - Signaled bandwidth is fixed at a configured level, regardless of how much traffic is on the LSP
- **pyNTM will route a Demand over RSVP LSPs *only* if the Demand and LSPs share the same source Node and destination Node

Working with an LSP

- Show the LSP objects in the Model
- Select an LSP for analysis

```
>>> from pprint import pprint
>>>
>>> for lsp in model2.rsvp_lsp_objects:
...     pprint(lsp)
...     print()
...
RSVP_LSP(source = F, dest = E, lsp_name = 'lsp_f_e_1')

RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_1')

RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_2')

>>>
```

```
Help on method get_rsvp_lsp in module pyNTM.model:

get_rsvp_lsp(source_node_name, dest_node_name, lsp_name='none') method of pyNTM.m
odel.Model instance
    Returns the RSVP LSP from the model with the specified source node
    name, dest node name, and LSP name.
~
```

```
>>> help(model2.get_rsvp_lsp)

>>> lsp1 = model2.get_rsvp_lsp('A', 'D', 'lsp_a_d_1')
>>> lsp1
RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_1')
>>>
```

RSVP LSPs and traffic

- Retrieve the LSPs reserved bandwidth
- Retrieve the amount of traffic on the LSP
- Retrieve the specific traffic demands on the LSP
- What we learned:
  - This LSP has 75 units of traffic
  - This LSP has reserved 75 units of bandwidth
  - This LSP carries 2 demands

```
>>> lsp1.reserved_bandwidth
75.0
>>>
>>> lsp1.traffic_on_lsp(model2)
75.0
>>>
>>> lsp1.demands_on_lsp(model2)
[Demand(source = A, dest = D, traffic = 70, name = 'dmd_a_d_2'),
 Demand(source = A, dest = D, traffic = 80, name = 'dmd_a_d_1')]
>>>
```

Further RSVP LSP traffic analysis

- When a Demand is carried by an RSVP LSP, that LSP will make up the Demand's path
  - In the example shown, we look at the first demand on the list of demands carried by lsp1
  - That demand is split across two LSPs
- When a Demand is IGP routed, its path will contain Interface objects
- Notice that the RSVP LSPs and Demand all share the same source and destination Nodes

```
>>> lsp1.demands_on_lsp(model2)
[Demand(source = A, dest = D, traffic = 70, name = 'dmd_a_d_2'),
 Demand(source = A, dest = D, traffic = 80, name = 'dmd_a_d_1')]
>>>
```

```
>>> dmd0 = lsp1.demands_on_lsp(model2)[0]
>>> dmd0
Demand(source = A, dest = D, traffic = 70, name = 'dmd_a_d_2')
```

```
>>> dmd0.path
[RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_1'),
RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_2')]
```

RSVP LSP Path Analysis

- There is a wealth of information in the RSVP_LSP path object
  - Interfaces: a list of Interfaces the LSP takes, ordered from source to destination
  - Path cost: the cost of the LSP
    - Per the RSVP protocol, this will be the IGP shortest path metric, regardless of the actual path the LSP takes
  - Baseline path reservable bandwidth
    - How much reservable bandwidth was available on the LSP's path when it was signaled

```
>>> for k,v in lsp1.path.items():
...     print("{}: {}".format(k, v))
...     print()
...
interfaces: [Interface(name = 'A-to-B', cost = 20, capacity = 12
5, node_object = Node('A'), remote_node_object = Node('B'), addr
ess = 7), Interface(name = 'B-to-D', cost = 20, capacity = 125,
node_object = Node('B'), remote_node_object = Node('D'), address
 = 4)]

path_cost: 40

baseline_path_reservable_bw: 125

>>>
```

RSVP LSP Path Analysis (continued)

- The RSVP LSP has a wealth of methods and attributes
  - Easily compare the LSPs effective metric to the LSP path's actual metric
    - *effective_metric* is the IGP shortest path cost between the source and destination nodes
      - It is equivalent to *path_cost* in the LSPs path info
    - *actual_metric* is the path cost of the LSP's signaled path
  - This allows you to easily analyze how many LSPs are on the shortest path and the cost differential between the shortest path and the LSP's actual path

```
Help on method effective_metric in module pyNTM.rsvp:

effective_metric(model) method of pyNTM.rsvp.RSVP_LSP instance
    Returns the metric for the best path. This value will be the
    shortest possible path from LSP's source to dest, regardless of
    whether the LSP takes that shortest path or not.
(END)
```

```
Help on method actual_metric in module pyNTM.rsvp:

actual_metric(model) method of pyNTM.rsvp.RSVP_LSP instance
    Returns the metric sum of the interfaces that the LSP actually
    transits.
~
```

```
>>> dir(lsp1)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc
__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__gt__', '__hash__', '__init__', '__init_subclass__', '_
_le__', '__lt__', '__module__', '__ne__', '__new__', '__re
duce__', '__reduce_ex__', '__repr__', '__setattr__', '__si
zeof__', '__str__', '__subclasshook__', '__weakref__', '_a
dd_rsvp_lsp_path', '_find_path_cost_and_headroom', '_find_
path_cost_and_headroom_routed_lsp', '_key', '_setup_bandwi
dth', 'actual_metric', 'configured_setup_bandwidth', 'dema
nds_on_lsp', 'dest_node_object', 'effective_metric', 'find
_rsvp_path_w_bw', 'lsp_name', 'path', 'reserved_bandwidth'
, 'route_lsp', 'setup_bandwidth', 'source_node_object', 't
raffic_on_lsp']
>>>
>>> lsp1.actual_metric(model2)
40
>>>
>>> lsp1.effective_metric(model2)
40
>>>
```

Comparing/configuring auto-bandwidth and static-bandwidth LSPs

- If an RSVP LSP has a value for the configured_setup_bandwidth attribute, it is static-bandwidth
- If the LSP does not have a value for this attribute, it is auto-bandwidth
- See the example on the next page

Example – Turn an auto-bandwidth LSP to a fixed bandwidth LSP and back again

- *lsp1* starts as an auto-bandwidth LSP
  - It tries to reserve the amount of bandwidth it is carrying
- Configure a value for *configured_setup_bandwidth* value on lsp1
  - Remember to update the simulation after making this change!
  - It still carries 75 units of traffic but only signals for the configured 20 units
- Change *lsp1* back to auto-bandwidth mode by changing *configured_setup_bandwidth* attribute to *None*
  - The *setup_bandwidth* reverts back to how much traffic the LSP is carrying (75 units)

```
>>> lsp1.setup_bandwidth
75.0
>>>
>>> lsp1.configured_setup_bandwidth
>>>
>>>
>>> lsp1.configured_setup_bandwidth = 20
>>> model2.update_simulation()
Routing the LSPs . . .
Routing 2 LSPs in parallel LSP group A-D; 1/2
Routing 1 LSPs in parallel LSP group F-E; 2/2
LSPs routed (if present); routing demands now . . .
Demands routed; validating model . . .
>>>
>>> lsp1.setup_bandwidth
20.0
>>> lsp1.configured_setup_bandwidth
20
>>>
>>> lsp1.traffic_on_lsp(model2)
75.0
```

```
>>> lsp1.configured_setup_bandwidth = None
>>>
>>> model2.update_simulation()
Routing the LSPs . . .
Routing 2 LSPs in parallel LSP group A-D; 1/2
Routing 1 LSPs in parallel LSP group F-E; 2/2
LSPs routed (if present); routing demands now . . .
Demands routed; validating model . . .
>>>
>>> lsp1.setup_bandwidth
75.0
>>>
```

Shared Risk Link Groups (SRLGs)

- A group of objects in the Model that all share the same risk of simultaneous failure
- Failing an SRLG fails all the component objects
- Unfailing an SRLG will unfail each component object as long as any given component object does not have another failure mode active
  - For example, if an Interface on a failed Node is also part of a failed SRLG and the SRLG unfails, the Interface will not restore until its host Node restores
- SRLGs can contain Interface and Node objects

Adding an SRLG to a Model

- Use the add_srlg Model method to create and SRLG

```
Help on method add_srlg in module pyNTM.model:

add_srlg(srlg_name) method of pyNTM.model.Model instance
    Adds SRLG object to Model
    :param srlg_name: name of SRLG
    :return:
(END)
```

```
>>> help(model2.add_srlg)

>>>
>>> model2.add_srlg('test_srlg')
>>>
>>> test_srlg = model2.get_srlg_object('test_srlg')
>>>
>>> test_srlg
SRLG(Name: test_srlg)
```

Adding objects to an SRLG

- Interfaces and Nodes each have a method to add that specific object to an SRLG
- When one Interface in a Circuit is added to an SRLG, the Circuit's other Interface will also be added to that SRLG

```
>>> dir(node_a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__
', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__su
bclasshook__', '__weakref__', '_failed', '_key', '_lat', '_lon'
, '_srlgs', 'add_to_srlg', 'adjacent_nodes', 'failed', 'interfa
ces', 'lat', 'lon', 'name', 'remove_from_srlg', 'srlgs']
>>>
```

```
Help on method add_to_srlg in module pyNTM.node:

add_to_srlg(srlg_name, model, create_if_not_present=False) meth
od of pyNTM.node.Node instance
    Adds self to an SRLG with name=srlg_name in model.
    :param srlg_name: name of srlg
    :param model: Model object
    :param create_if_not_present: Boolean.  Create the SRLG if
it
    does not exist in model already.  True will create SRLG in
    model; False will raise ModelException
    :return: None
(END)
```

Adding objects to an SRLG (continued)

- Get the desired Interface and/or Node objects in the Model
- Use the add_to_srlg method from each object to add each to the SRLG
- You can programmatically verify which Nodes and Interfaces are part of a given SRLG
- Update the simulation!

```
>>> node_a = model2.get_node_object('A')
>>> node_e = model2.get_node_object('E')
>>> int_c_d = model2.get_interface_object('C-to-D', 'C')
```

```
>>> node_a.add_to_srlg('test_srlg', model2)
>>> node_e.add_to_srlg('test_srlg', model2)
>>> int_c_d.add_to_srlg('test_srlg', model2)
>>>
```

```
>>> test_srlg.node_objects
{Node('A'), Node('E')}
>>>
>>> test_srlg.interface_objects
{Interface(name = 'C-to-D', cost = 30, capacity = 150, node_obj
ect = Node('C'), remote_node_object = Node('D'), address = 1),
Interface(name = 'D-to-C', cost = 30, capacity = 150, node_obje
ct = Node('D'), remote_node_object = Node('C'), address = 1)}
>>>
```

```
>>> model2.update_simulation()
Routing the LSPs . . .
Routing 2 LSPs in parallel LSP group A-D; 1/2
Routing 1 LSPs in parallel LSP group F-E; 2/2
LSPs routed (if present); routing demands now . . .
Demands routed; validating model . . .
>>>
```

Failing an SRLG

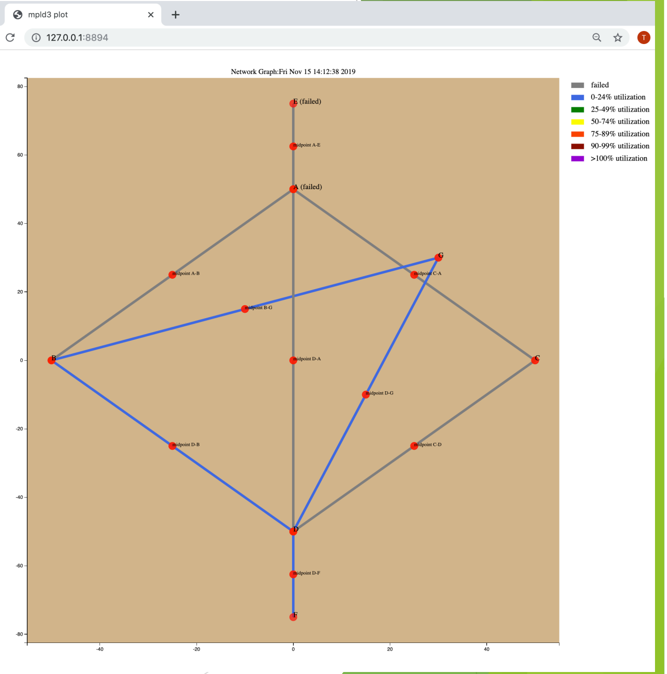
- Fail an SRLG with the Model fail_srlg method
- Use Model's *get_failed_node_objects* and *get_failed_interface_objects* methods to programmatically validate failed Nodes or Interfaces
- Update the simulation!
- Create network visualization (optional)

```
>>> model2.fail_srlg('test_srlg')
>>> model2.update_simulation()
Routing the LSPs . . .
Routing 2 LSPs in parallel LSP group A-D; 1/2
Routing 1 LSPs in parallel LSP group F-E; 2/2
LSPs routed (if present); routing demands now . . .
Demands routed; validating model . . .
>>>
```

```
>>> model2.get_failed_node_objects()
[Node('E'), Node('A')]
>>>
>>> len(model2.get_failed_interface_objects())
10
>>>
```

```
>>> graph_network_interactive.make_interactive_network_graph(model2)
>>> Serving to http://127.0.0.1:8894/    [Ctrl-C to exit]
127.0.0.1 - - [15/Nov/2019 14:12:39] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [15/Nov/2019 14:12:39] "GET /d3.js HTTP/1.1" 200 -
127.0.0.1 - - [15/Nov/2019 14:12:39] "GET /mpld3.js HTTP/1.1" 200 -

>>>
```

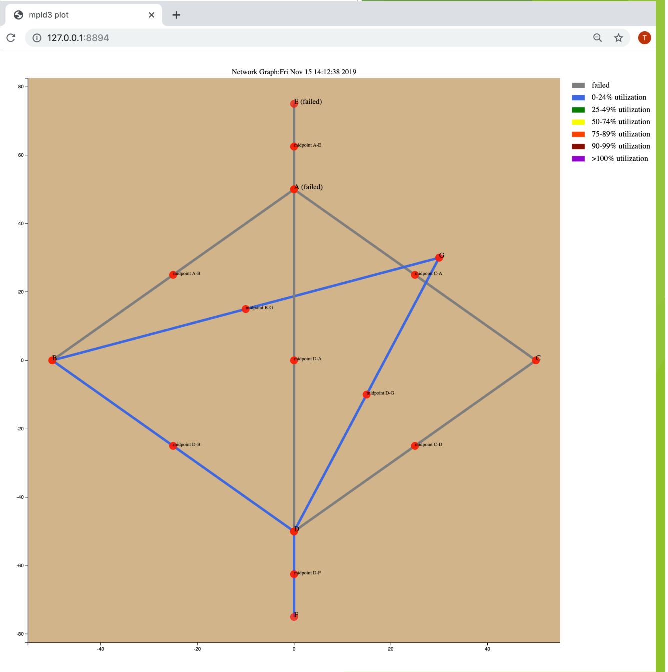


Assessing the impact of the failure

- Our visualization show no Interface over 24% utilized
- HOWEVER, use the *get_unrouted_demand_objects* Model method to assess which (if any) Demands are not able to route
  - Spoiler alert: None of the 3 existing demands can route after this devastating failure

```
>>> model2.get_failed_node_objects()
[Node('E'), Node('A')]
>>>
>>> len(model2.get_failed_interface_objects())
10
>>>
```

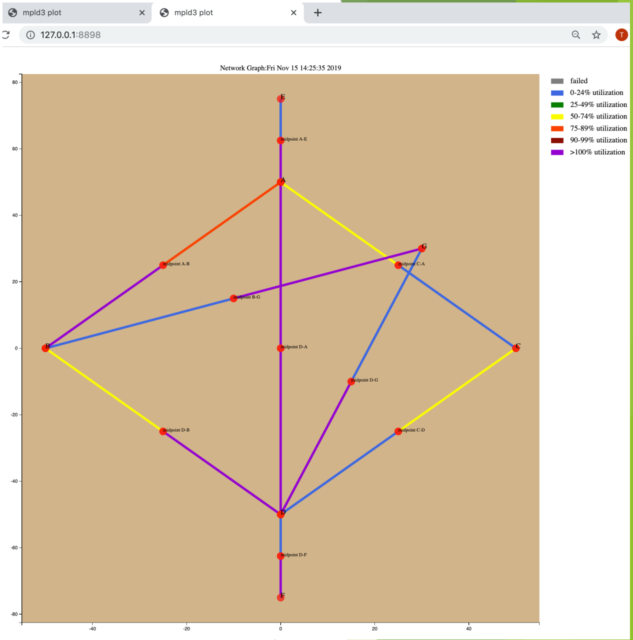
```
>>> model2.get_unrouted_demand_objects()
[Demand(source = A, dest = D, traffic = 70, name = 'dmd_a_d_2'),
 Demand(source = A, dest = F, traffic = 40, name = 'dmd_a_f_1'),
 Demand(source = A, dest = D, traffic = 80, name = 'dmd_a_d_1'),
 Demand(source = F, dest = E, traffic = 400, name = 'dmd_f_e_1')
]
```



Unfail the SRLG

- Use the *unfail_srlg* Model method to restore a failed SRLG
  - Use Model's *get_failed_node_objects* and *get_failed_interface_objects* methods to programmatically validate no failed Nodes or Interfaces
- Update the simulation!
- Visualize the network (optional)
  - Visualization shows network back to its pre-failed state

```
>>> model2.unfail_srlg('test_srlg')
>>> model2.update_simulation()
Routing the LSPs . . .
Routing 2 LSPs in parallel LSP group A-D; 1/2
Routing 1 LSPs in parallel LSP group F-E; 2/2
LSPs routed (if present); routing demands now . . .
Demands routed; validating model . . .
>>>
>>> model2.get_failed_node_objects()
[]
>>> model2.get_failed_interface_objects()
[]
>>>
```



FIN