

The GÉANT Testbed Service (GTS)

DSL Training v.1.0-b4308

Susanne Naegele-Jackson – RRZE

Fabio Farina – GARR

Robert Szuman – PSNC

Blazej Pietrzak – PSNC

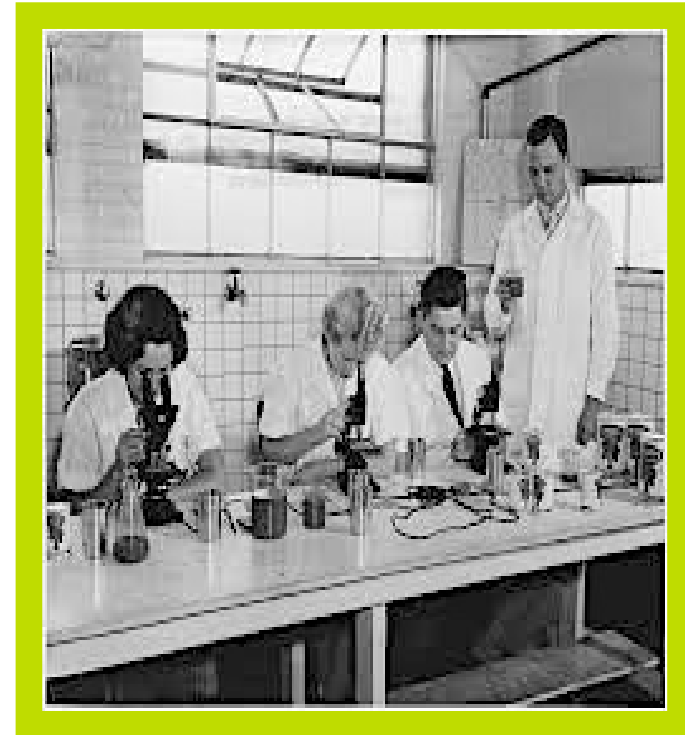
14 October 2015

Introduction

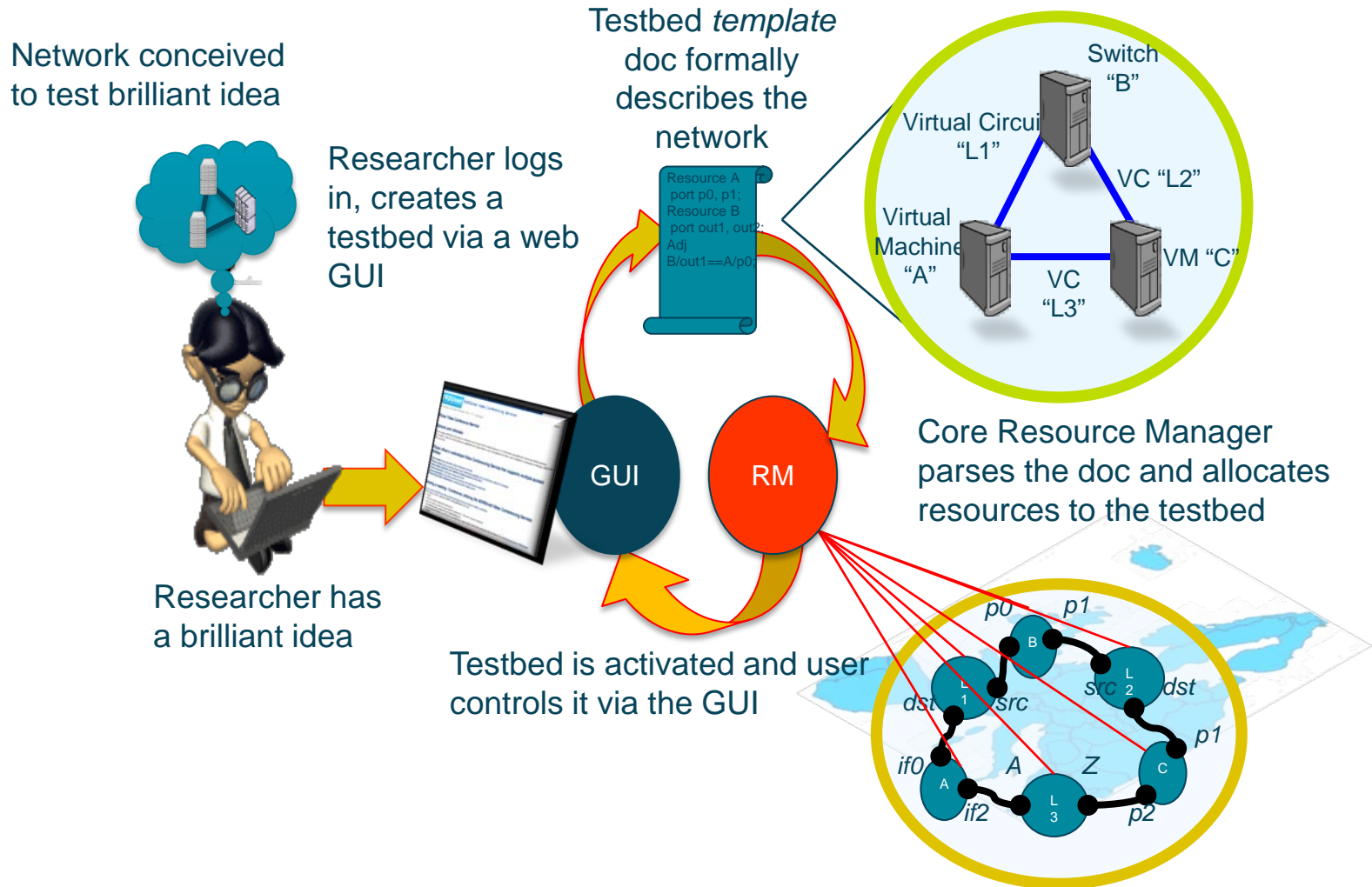
The GÉANT Testbed Service



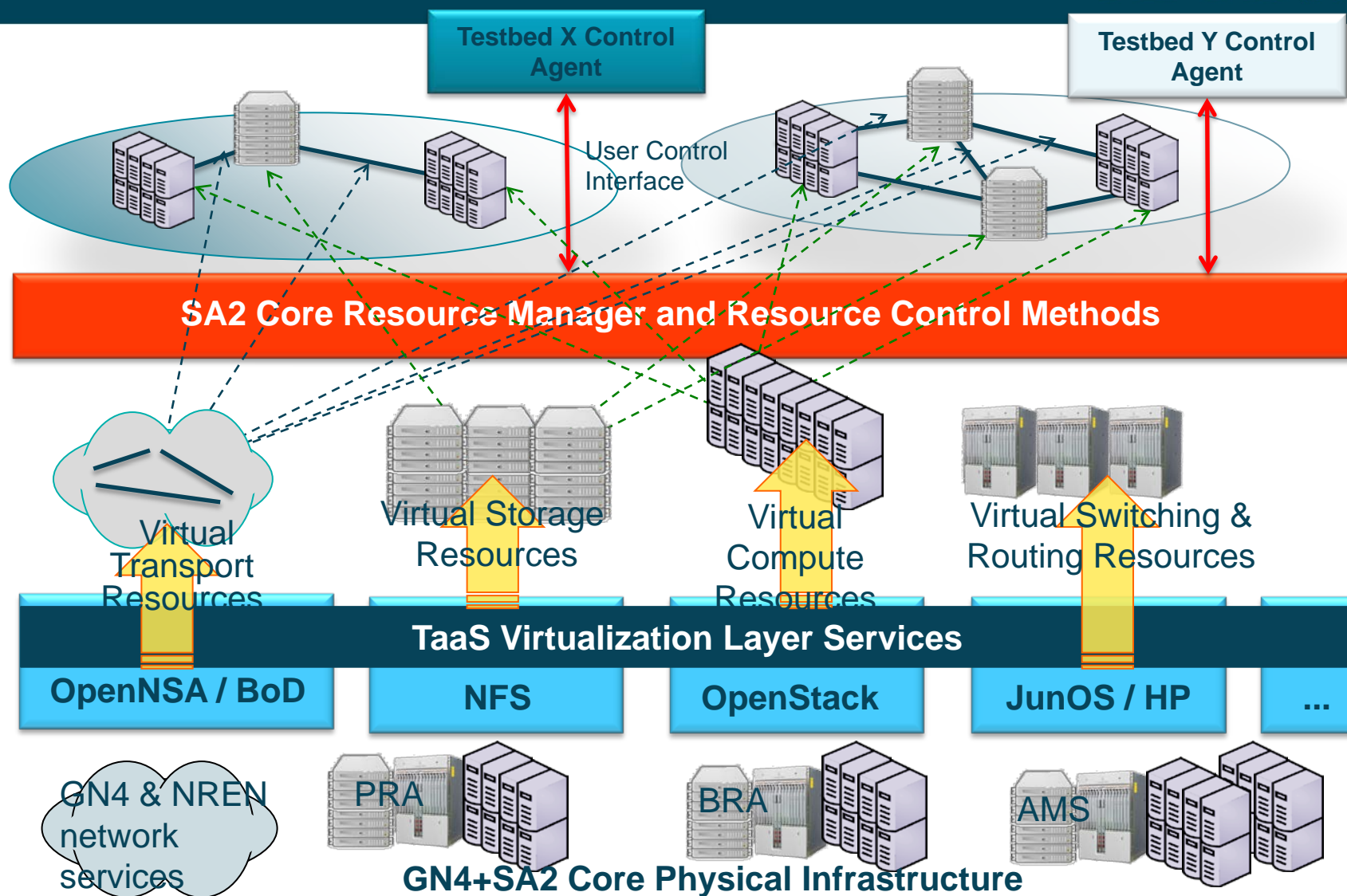
- **Accelerating Network Innovation**
 - Make it easy for network research (broadly construed) to quickly and effectively test innovative technologies!
- **SA2 objective:**
 - Provide experimental laboratories for network research**
 - **Simple:** Minimal learning curve
 - **Agile:** Rapid prototype, short iteration cycle
 - **Secure:** Support high risk experiments, protect others
 - **Flexible:** Supports a wide range of research activities
 - **Scalable:** Support very large, diverse experiments.
 - **World Wide:** geographically distributed, reaches beyond GÉANT

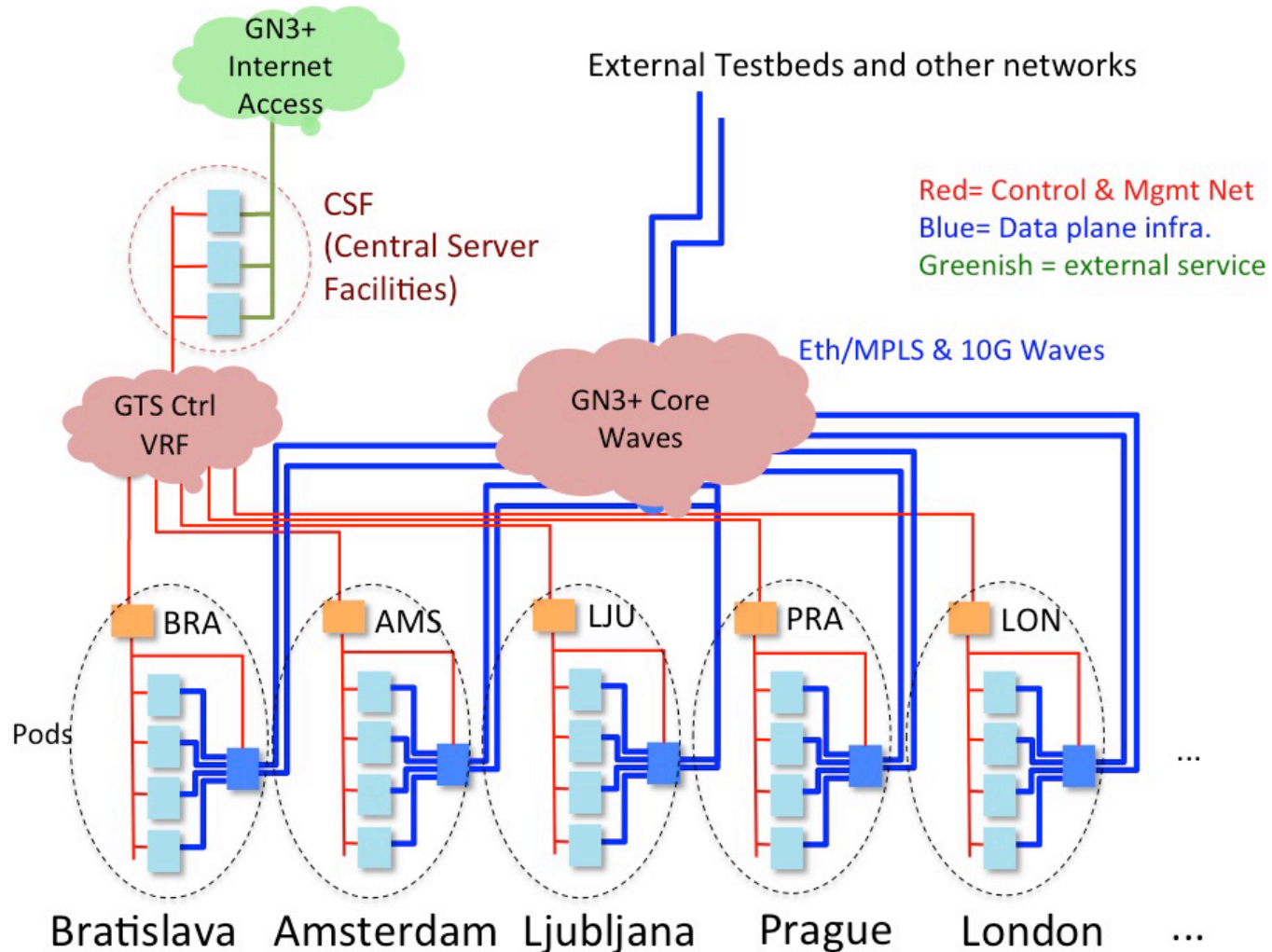


How it works



The architecture

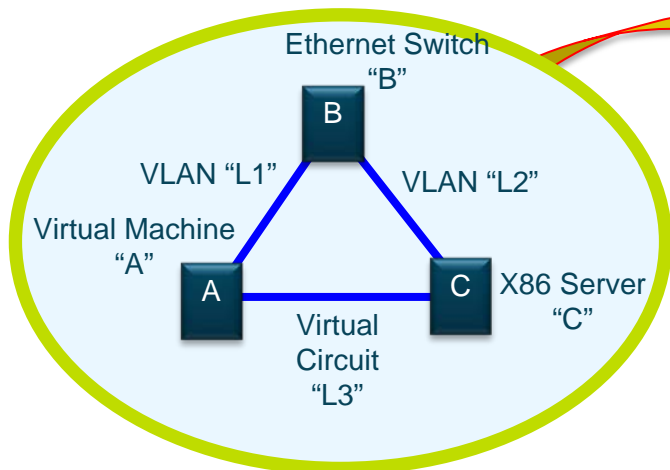




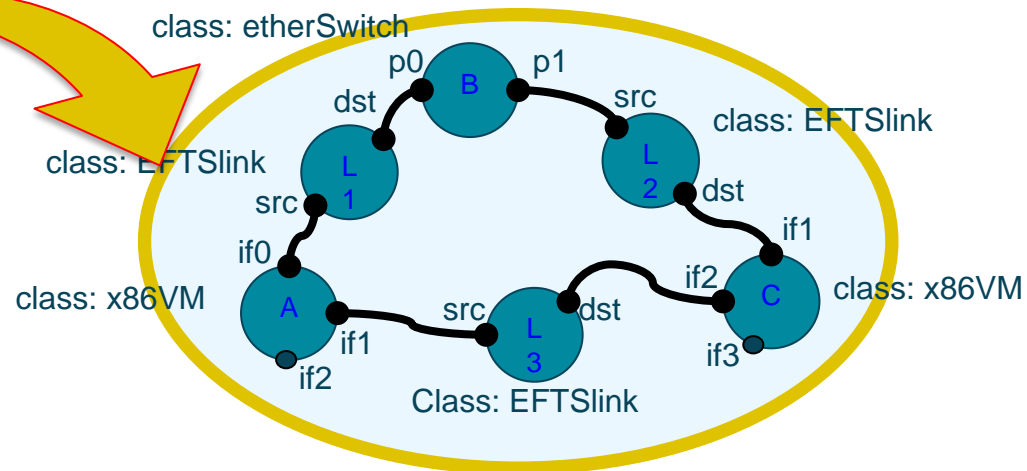
The Resource Graph

The TaaS Architecture treats all [testbed] networks as **graphs**

Testbed “Alpha” Description



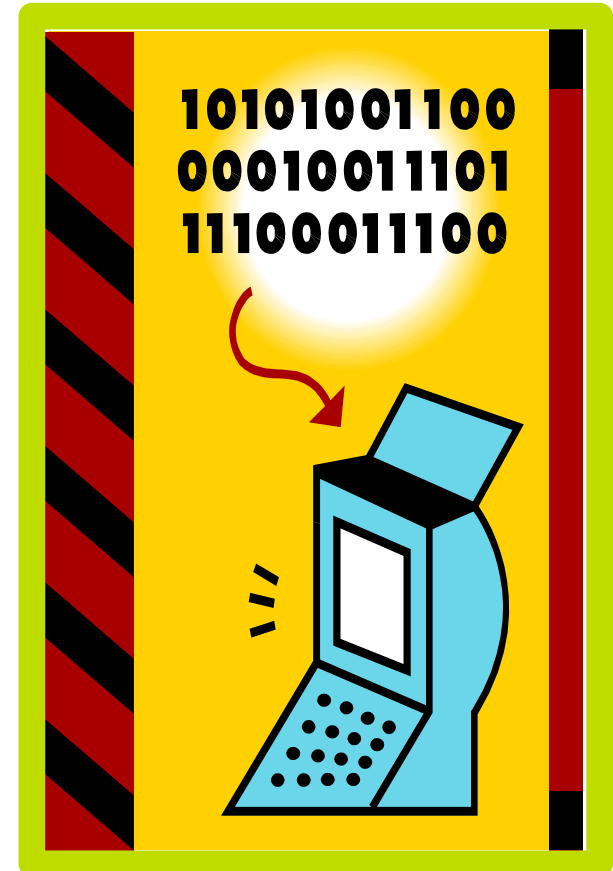
Internally, all testbed components are treated as generalized **virtual Resources**. All Resources all have a set of explicitly defined **data flow Ports**. User specified Port adjacency relations define the testbed topology.



“Derived Resource Graph” data plane

Object Oriented Testbed Descriptions

- The DSL is based on the “groovy” OO language
 - Allows **programmatic** descriptions of network layout
 - Enables users to define **repetitive** connectivity patterns/processes
- **Composite resources**
 - Large **complex** testbeds are constructed from simpler more atomic resources
- **Users can define their own resource classes**
 - Novel **hardware or functional agents** can be introduced



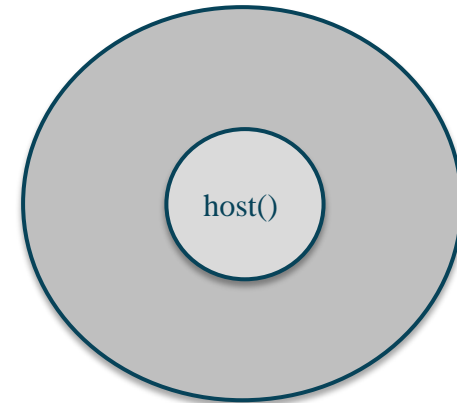
Questions?



DSL Training

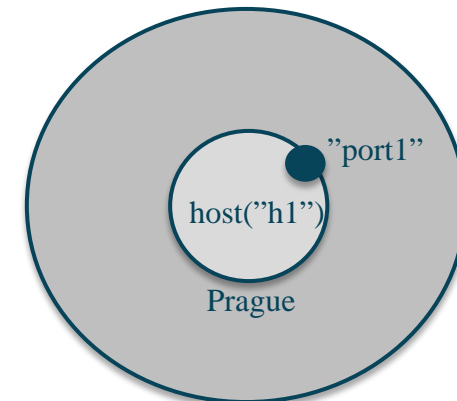
```
testbed {  
    id = "OneHost"  
  
    host {  
    }  
}
```

testbed ("OneHost")



```
testbed {  
    id = "OneHost"  
    description = "One host in Prague"  
  
    host {  
        id="h1"  
        location="pra"  
        port { id="port1" }  
    }  
}
```

testbed ("OneHost")

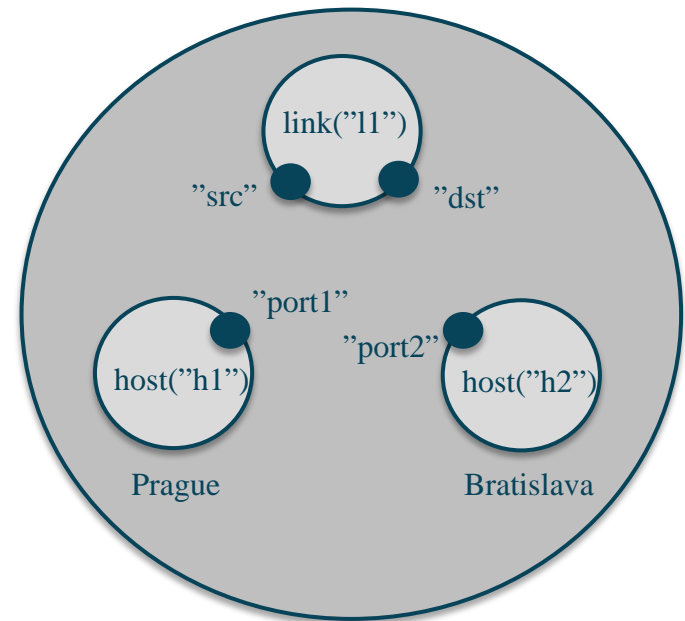


HostsLine.groovy 1/2



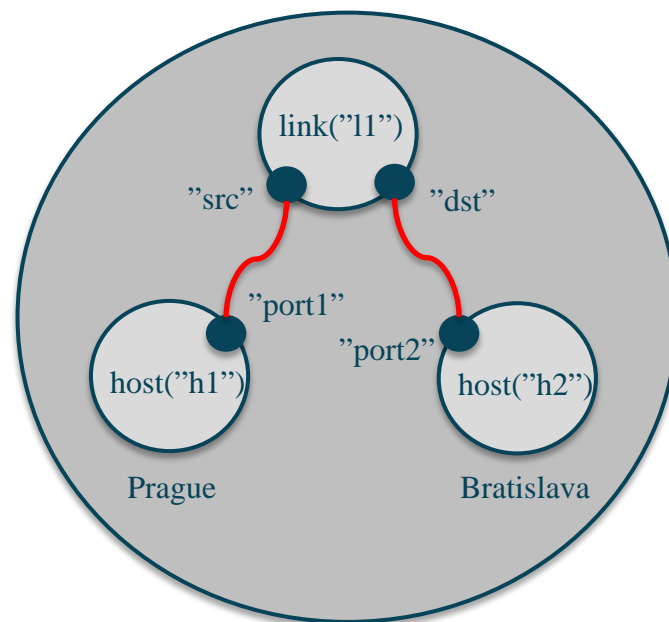
```
HostsLine {  
  description = "PRA host linked with BRA host"  
  
  host {  
    id="h1"  
    location="pra"  
    port { id="port1" }  
  }  
  
  host {  
    id="h2"  
    location="bra"  
    port { id="port2" }  
  }  
  
  link {  
    id="l1"  
    port { id="src" }  
    port { id="dst" }  
  }  
  ...  
}
```

HostsLine ()



```
HostsLine {  
  description = "PRA host linked with BRA host"  
  
  host {  
    id="h1"  
    location="pra"  
    port { id="port1" }  
  }  
  
  host {  
    id="h2"  
    location="bra"  
    port { id="port2" }  
  }  
  
  link {  
    id="l1"  
    port { id="src" }  
    port { id="dst" }  
  }  
  
  adjacency h1.port1, l1.src  
  adjacency h2.port2, l1.dst  
}
```

HostsLine ()

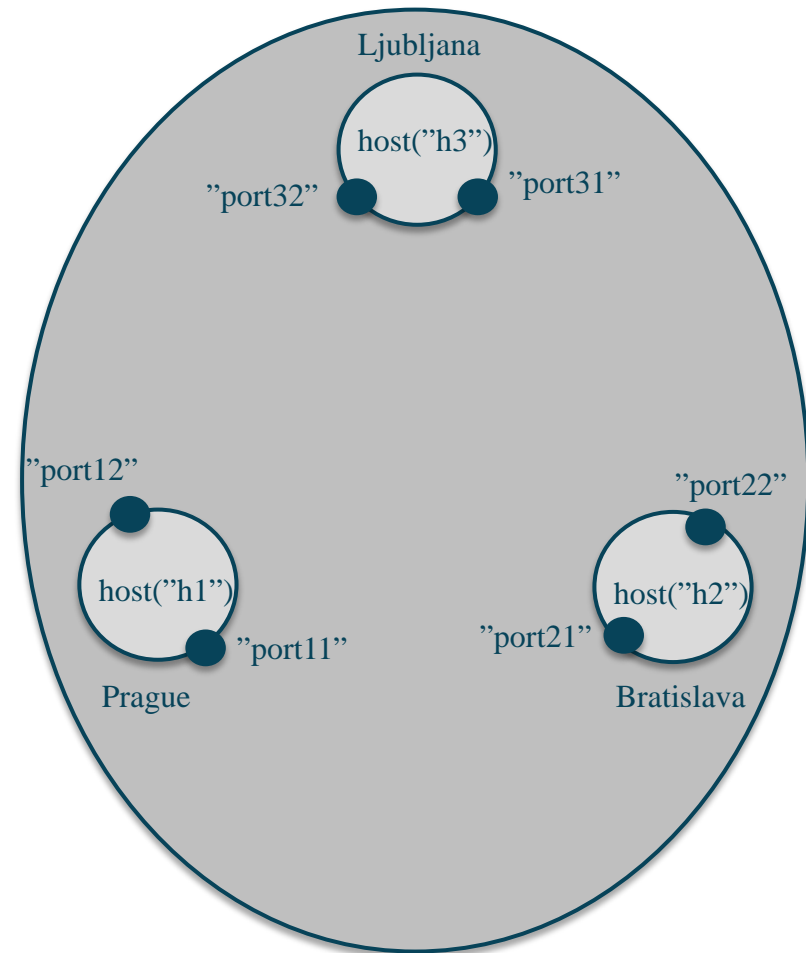


Triangle.groovy 1/3



```
type triangle {  
  description = "Triangle between PRA, BRA and LJU"  
  
  host {  
    id="h1"  
    location="pra"  
    port { id="port11" }  
    port { id="port12" }  
  }  
  
  host {  
    id="h2"  
    location="bra"  
    port { id="port21" }  
    port { id="port22" }  
  }  
  
  host {  
    id="h3"  
    location="lju"  
    port { id="port31" }  
    port { id="port32" }  
  }  
  
  ...  
}
```

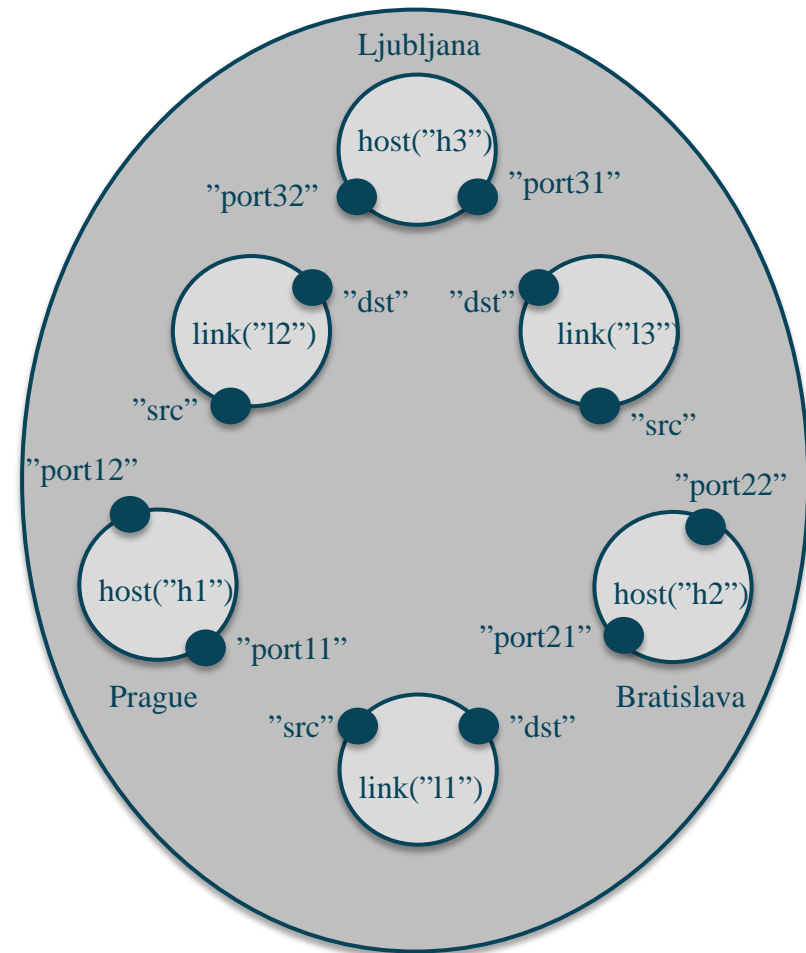
type triangle ()



Triangle.groovy 2/3

```
...  
link {  
    id="l1"  
    port { id="src" }  
    port { id="dst" }  
}  
  
link {  
    id="l2"  
    port { id="src" }  
    port { id="dst" }  
}  
  
link {  
    id="l3"  
    port { id="src" }  
    port { id="dst" }  
}  
  
...
```

type triangle ()



Triangle.groovy 3/3



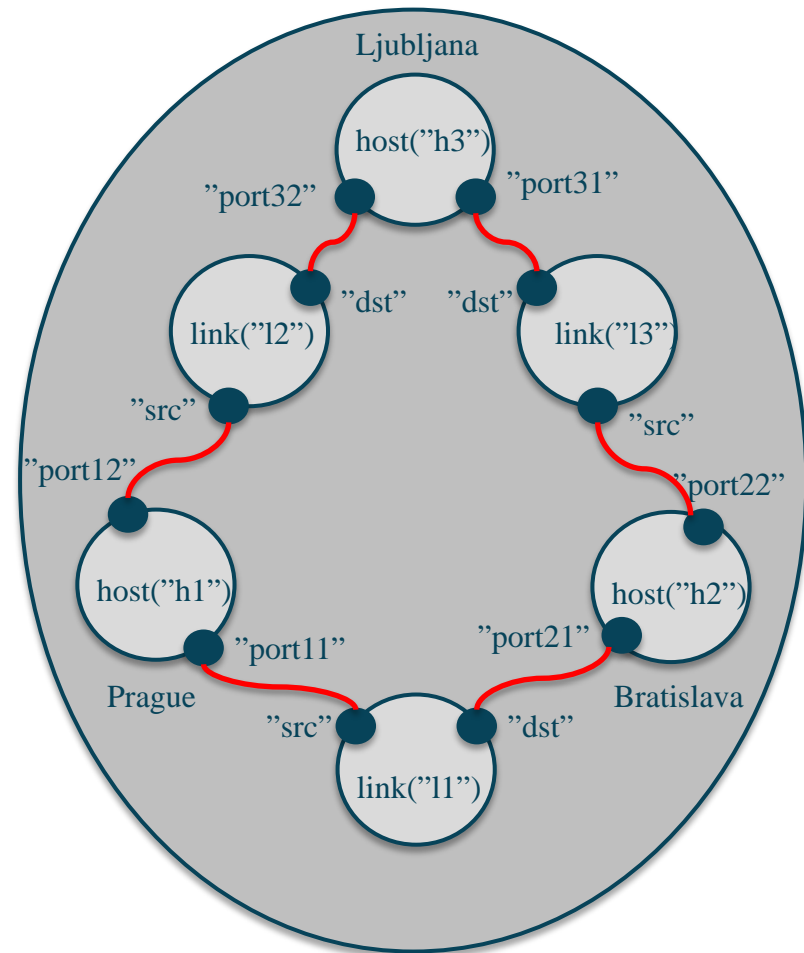
```
...
link {
    id="l1"
    port { id="src" }
    port { id="dst" }
}

link {
    id="l2"
    port { id="src" }
    port { id="dst" }
}

link {
    id="l3"
    port { id="src" }
    port { id="dst" }
}

adjacency h1.port11, l1.src
adjacency h2.port21, l1.dst
adjacency h1.port12, l2.src
adjacency h3.port32, l2.dst
adjacency h2.port22, l3.src
adjacency h3.port31, l3.dst
}
```

type triangle ()



TriangleLoop.groovy



```
triangle {
  description = "Triangle using Groovy
  language iterators to define adjacencies."
  id = "t1"
  def hosts = []
  def links = []

  3.times { idx ->
    def h1 = host {
      id = "host$idx"
      port { id = "p1" }
      port { id = "p2" }
    }
    hosts << h1

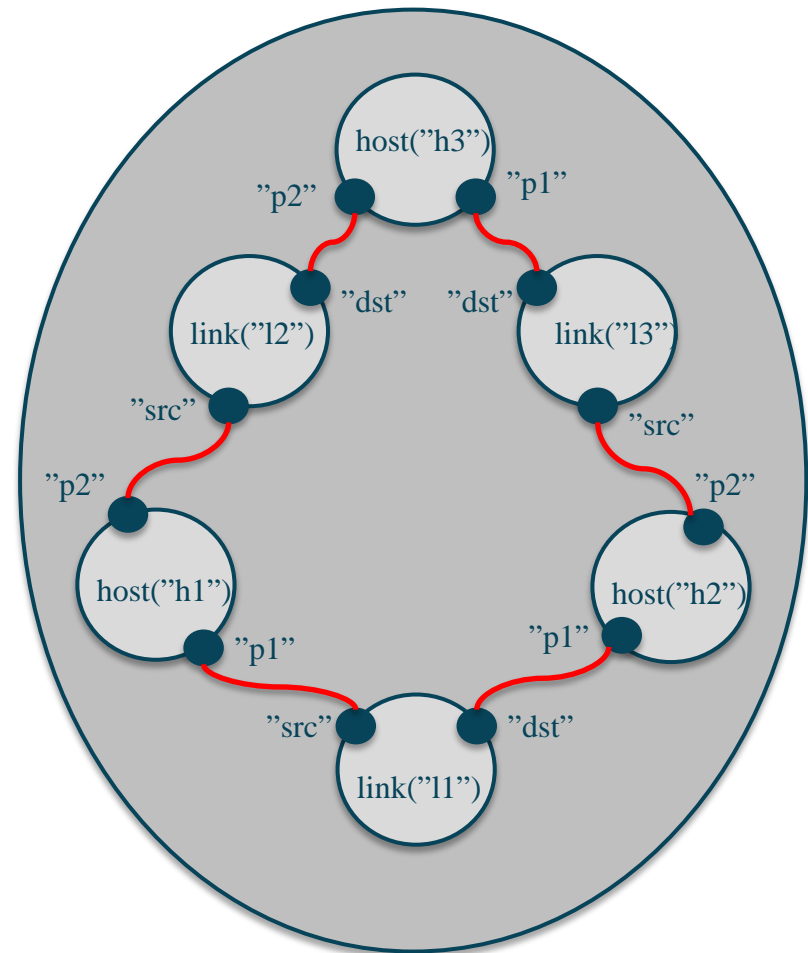
    def l1 = link {
      id = "link$idx"
      port { id = "src" }
      port { id = "dst" }
    }
    links << l1

    adjacency h1.p1, l1.src

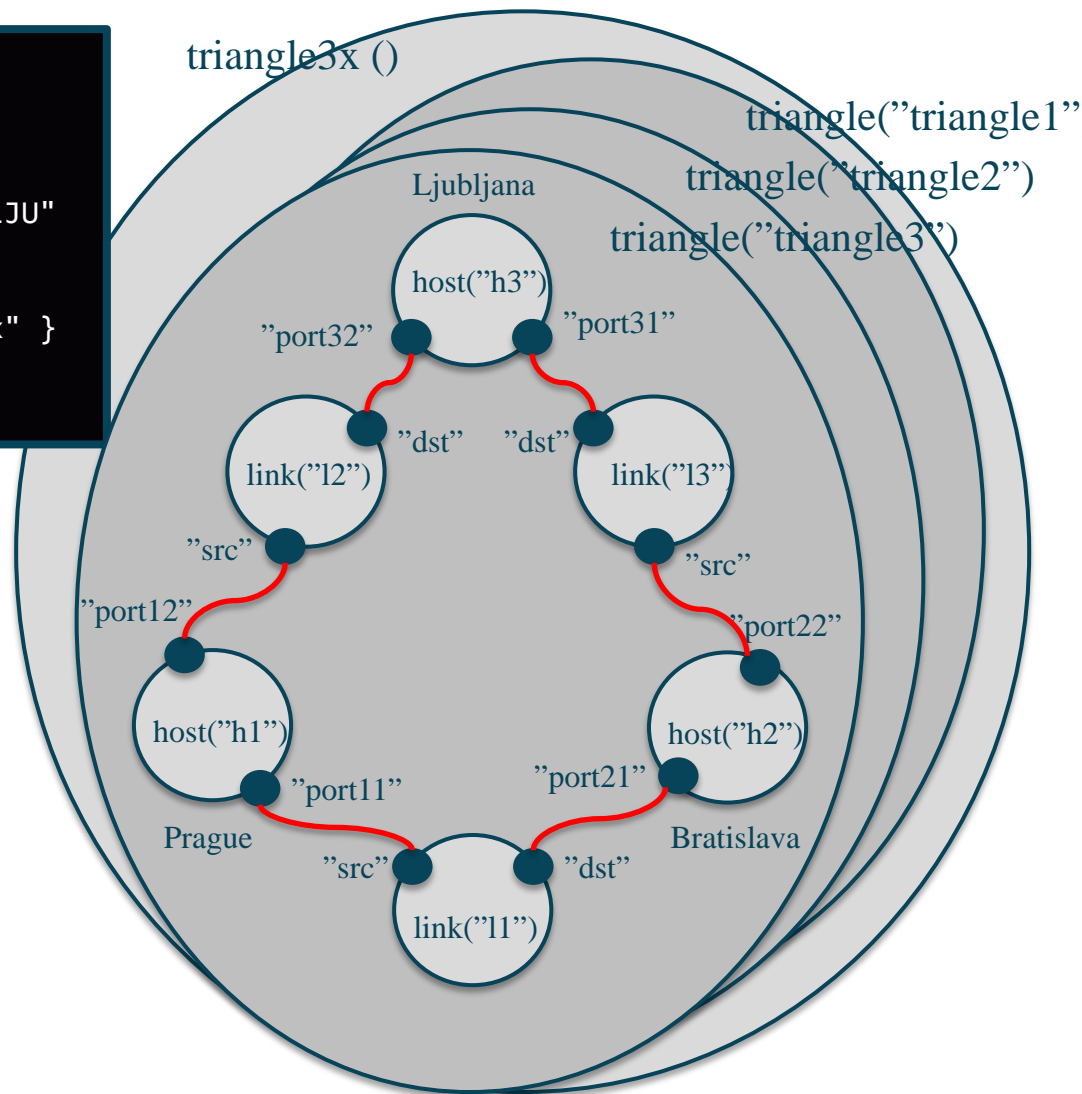
  }

  3.times { idx -> adjacency hosts[(idx +
    1) % 3].p2, links[idx].dst }
}
```

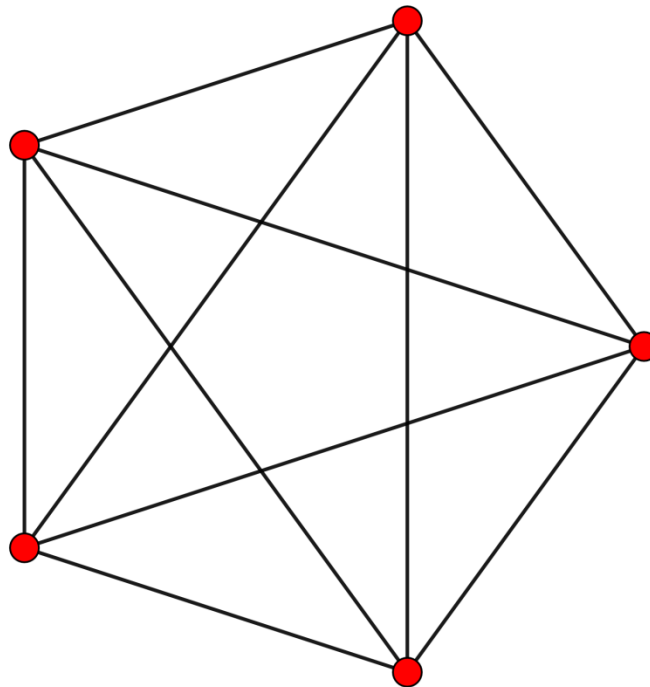
triangle ("t1")



```
...  
triangle3x {  
    description = "Three times triangle  
                  between PRA, BRA and LJU"  
  
    3.times { idx ->  
        triangle { id = "triangle$idx" }  
    }  
}
```

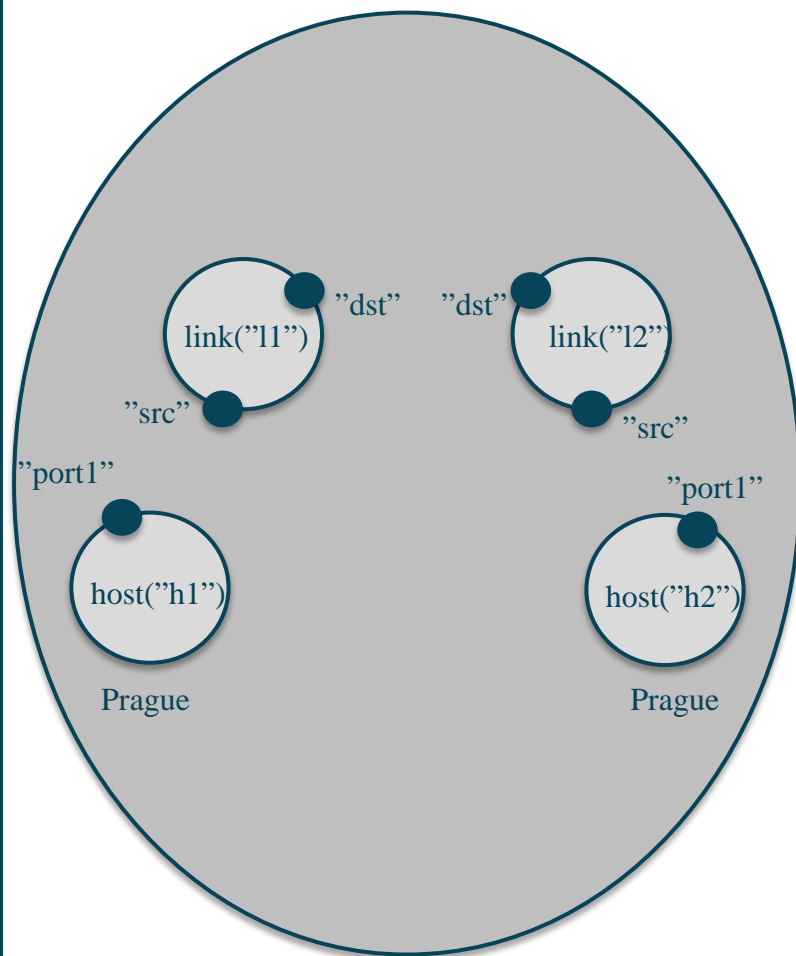


- Study the Groovy file and figure out how the pentagon topology can be created.
- Try and use the iterators for both links and hosts
- Don't forget, the links are treated as resources and you have to define adjacencies between the ports



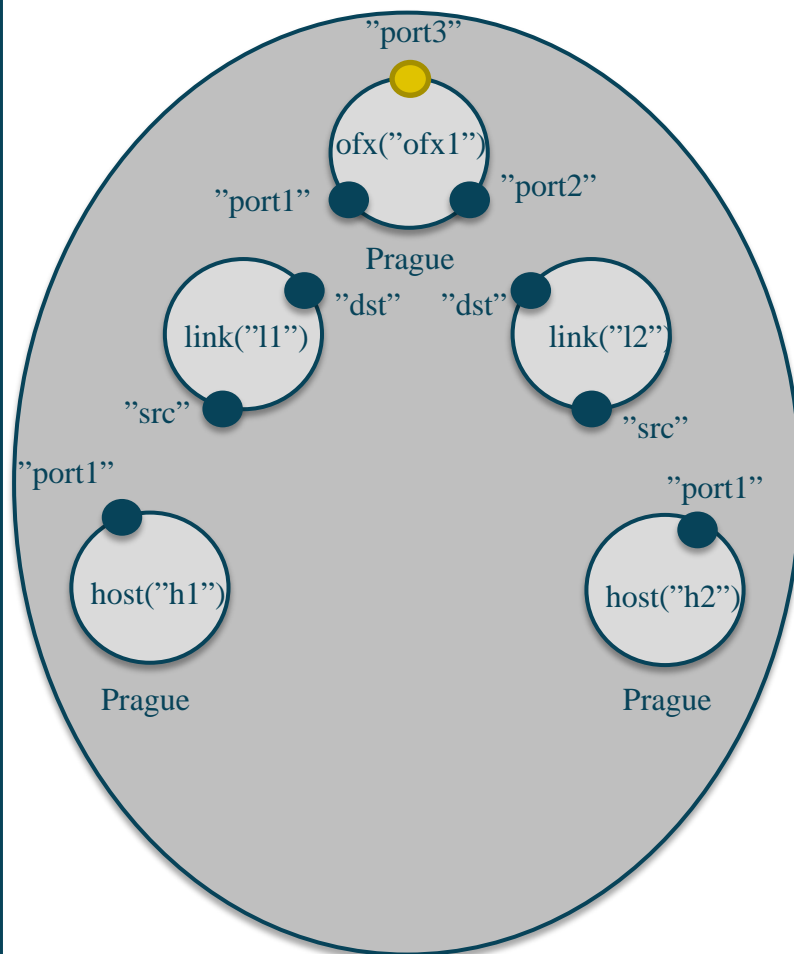
```
OneOFX {  
  description = "Two hosts connected to Openflow switch"  
  id = "t1"  
  
  host {  
    id="h1"  
    location="pra"  
    port { id="port1" }  
  }  
  
  host {  
    id="h2"  
    location="pra"  
    port { id="port1" }  
  }  
  
  link {  
    id="l1"  
    port { id="src" }  
    port { id="dst" }  
  }  
  
  link {  
    id="l2"  
    port { id="src" }  
    port { id="dst" }  
  }  
  ...  
}
```

OneOFX ("t1")



```
...
ofx {
  id="ofx1"
  location="pra"
  fabricIPv4="10.10.100.1"
  controllerPort="9966"
  fabricSubnetMaskv4="255.255.255.0"
  controllerIPv4="10.10.100.100"
  port { id="port1" }
  port { id="port2" }
  port {
    id="port3"
    mode="CONTROL"
  }
}
...
```

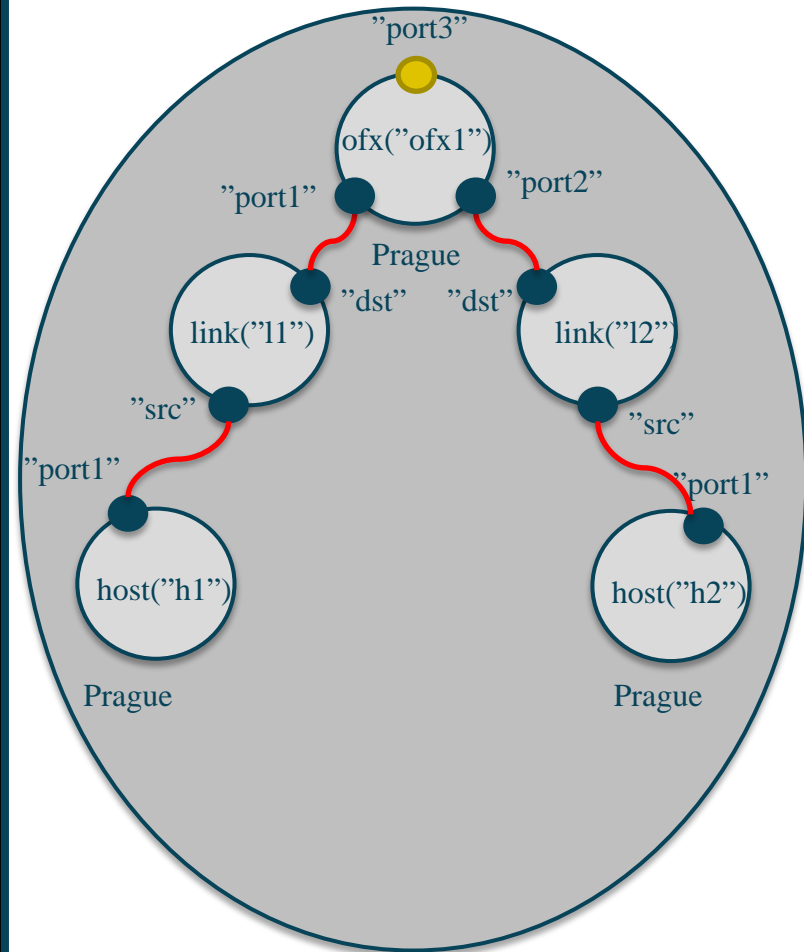
OneOFX ("t1")



```
...
ofx {
  id="ofx1"
  location="pra"
  fabricIPv4="10.10.100.1"
  controllerPort="9966"
  fabricSubnetMaskv4="255.255.255.0"
  controllerIPv4="10.10.100.100"
  port { id="port1" }
  port { id="port2" }
  port {
    id="port3"
    mode="CONTROL"
  }
}

adjacency h1.port1, l1.src
adjacency ofx1.port1, l1.dst
adjacency h2.port1, l2.src
adjacency ofx1.port2, l2.dst
}
```

OneOFX ("t1")



Questions?



- Visit the service page

<http://services.geant.net/gts/>

- Manuals, guides and training

<http://services.geant.net/gts/resources>

- For further information, support, special needs

support@gts.geant.net

GUI Demonstration

Tips & Tricks

Special purpose resources
DSL advanced usage hints

Testbed external domain stitching



- Connect your GTS testbed to your lab
- ExternalDomainPort resource type

```
ExternalDomainPort_example{
  host {
    id="h1"
    port { id="eth1" }
  }

  link {
    id="l1"
    port { id="src" }
    port { id="dst" }
  }

  ExternalDomain {
    id = "ProtoGENI-SL"
    location = "LON"
    port { id="ep1" }
  }

  adjacency h1.eth, l1.src
  adjacency ProtoGENI-SL.ep1, l1.dst
}
```

- ExternalDomain properties
 - id – User defined ID, case sensitive string;
 - location – “AMS, BRA, LJU, ...”. It’s where the External Domain VC terminates at the GTS edge
 - Port id – Either “ep1”, or “ep2”
- Semi-automatic provisioning!
 - GTS creates the link up to ExternalDomain port
 - Then, **you must contact us and your NREN** to connect your lab to GTS through GÉANT Network

- Defining types and using them as nested resources
- **Useful when** you have to repeat complex fragments

```
...
type NestedHost {
  host {
    id = "h1"
    port { id = "eth6" }
    port { id = "eth4" }
  }
  port { id = "p1" }
  adjacency p1, h1.eth4
}
...
```



```
NestedHostsLine {
  id = "t1"

  NestedHost {
    id = "n1"
  }

  host {
    id = "h2"
    port { id = "eth5" }
  }

  link {
    id = "link"
    port { id = "src" }
    port { id = "dst" }
  }

  adjacency n1.p1, link.src
  adjacency h2.eth5, link.dst
}
```



- Use types together with Groovy constructs, check the syntax online
 - Different iterators: times, each, for, while, ...
 - Data structures: maps, lists, intervals

```
type VertexEdge {
    host {
        id = "h"
        port { id = "eth1" }
        port { id = "eth2" }
    }
    link {
        id = "l"
        port { id = "src" }
        port { id = "dst" }
    }
}

adjacency h.eth1, l.src


port { id = "eth2" }
port { id = "dst" }

adjacency eth2, h.eth2
adjacency dst, l.dst
}
```



```
type Triangle {
    id = "t"
    3.times { idx -> VertexEdge { id = "v${idx}" } }
    3.times { idx ->
        adjacency this."v${(idx + 1) % 3}".p, this."v${idx}".dst
    }
}

def list = ["red", "blue", "green"]
list.each {
    Triangle { id = "Triangle_${it}" }
}
```



- One-liners declaration and user defined functions through delegation
- **Useful when** your experiment needs parametric instances

```
def createHost(composite, name) {
  composite.with {
    host { id = name; port { id = "eth1" } }
  }
}

HostsVC_byFunction {
  link {
    id = "link"
    port { id = "src" }
    port { id = "dst" }
  }
  adjacency createHost("h1").eth1, link.src
  adjacency createHost("h2").eth1, link.dst
}

HostsVC_byInline {
  ...
  adjacency host { id = "h1"; port { id = "eth1" } }.eth1, link.src
  adjacency host { id = "h2"; port { id = "eth1" } }.eth1, link.dst
}
```

- Full fledge OOP: inheritance, polymorphism
- **Useful when** you want full control over your testbed through Groovy

```
// Parent Class
class BaseClass{
    String nOFX
    Integer nVM

    def BaseClass(nVM=1, nOFX=1) {
        this.nOFX = nOFX
        this.nVM = nVM
        buildTestbed()
    }

    def buildTestbed() {
        // build your testbed
        // based on #OFXs, #VMs
    }
}
```

```
// Child Class:
@groovy.transform.InheritConstructors
class AnotherClass extends BaseClass {
    // overload parent definition
    def buildTestbed() {
        // call parent method
        super.buildTestbed()
    }
}

def testbed_pool = [:]
locs["tb1"] = new Child(2,1)
locs["tb2"] = new AnotherClass()
```