lecture 10:
Pyretic revist

5590: software defined networking

anduo wang, Temple University

TTLMAN 401B, R 17:30-20:00

Pyretic revisit: dynamic policy

query policy

define policy

C ::= A | P[C] | C|C | C»C | Q

Q ::= packets | count

packet, count buckets

- resulting located packets diverted to “buckets” in the controller
- application registers listeners with buckets
- buckets passes entire packets to the listeners

query policy

- application registers listeners with buckets
- buckets passes entire packets to the listener

```
def printer(pkt):
  print pkt

def dpi():
  Q = packets(None,[])
  Q.when(printer)
  return match(srcip='1.2.3.4')[Q]

def main():
  return dpi() | flood
```

example: deep packet inspection

- application registers listeners with buckets
- buckets passes entire packets to the listener

```
def printer(pkt):
  print pkt

def dpi():
  Q = packets(None,[])
  Q.when(printer)
  return match(srcip='1.2.3.4')[Q]

def main():
  return dpi() | flood
```

create a query policy, monitoring all traffic

example: deep packet inspection

- application registers listeners with buckets
- buckets passes entire packets to the listener

```
def printer(pkt):
  print pkt

def dpi():
  Q = packets(None,[])
  Q.when(printer)
  return match(srcip='1.2.3.4')[Q]

def main():
  return dpi() | flood
```

register `printer` as a listener

example: deep packet inspection

- application registers listeners with buckets
- buckets passes entire packets to the listener

```
def printer(pkt):
  print pkt

def dpi():
  Q = packets(None,[])
  Q.when(printer)
  return match(srcip='1.2.3.4')[Q]

def main():
  return dpi() | flood
```

each time a packet arrives at `packet_bucket`, `printer` is called, printing the (passed) packet

example: deep packet inspection

- application registers listeners with buckets
- buckets passes entire packets to the listener

```
def printer(pkt):
  print pkt

def dpi():
  Q = packets(None,[])
  Q.when(printer)
  return match(srcip='1.2.3.4')[Q]

def main():
  return dpi() | flood
```

construct a policy that collects traffic from `1.2.3.4`

dynamic policy

dynamic policy

dynamic policy

- changes in response to network changes
- query policies drive changes to other policies

dynamic policy

dynamic policy

- changes in response to network changes
- query policies drive changes to other policies

pattern

- query policy A collects network change
- A register dynamic policy B as listener
- upon network change
  - A passes the change (pkt) to B
  - B updates its policy dynamically

example: round-robin load-balancer

```
class rrlb(DynamicPolicy):

 def __init__(self,s,servers):
  self.switch = s
  self.servers = servers
  ...

  Q = packets(limit=1,group_by=['srcip'])
  Q.register_callback(self.round_robin)
  self.policy = match(dstport=80) >> Q

 def round_robin(self,pkt):
  self.policy = if_(match(srcip=pkt['srcip']),
                    modify(dstip=self.server), self.policy)
  self.client += 1
  self.server = self.servers[self.client % m]


servers = ['2.2.2.8','2.2.2.9']
rrlb_on_switch3 = rrlb(3,servers)
```

example: round-robin load-balancer

```
class rrlb(DynamicPolicy):

 def __init__(self,s,servers):
  self.switch = s
  self.servers = servers
  ...

  Q = packets(limit=1,group_by=['srcip'])
  Q.register_callback(self.round_robin)
  self.policy = match(dstport=80) >> Q

 def round_robin(self,pkt):
  self.policy = if_(match(srcip=pkt['srcip']),
                    modify(dstip=self.server), self.policy)
  self.client += 1
  self.server = self.servers[self.client % m]


servers = ['2.2.2.8','2.2.2.9']
rrlb_on_switch3 = rrlb(3,servers)
```

create a query policy

example: round-robin load-balancer

```
class rrlb(DynamicPolicy):

 def __init__(self,s,servers):
  self.switch = s
  self.servers = servers
  ...

  Q = packets(limit=1,group_by=['srcip'])
  Q.register_callback(self.round_robin)
  self.policy = match(dstport=80) >> Q

 def round_robin(self,pkt):
  self.policy = if_(match(srcip=pkt['srcip']),
                    modify(dstip=self.server), self.policy)
  self.client += 1
  self.server = self.servers[self.client % m]


servers = ['2.2.2.8','2.2.2.9']
rrlb_on_switch3 = rrlb(3,servers)
```

create a query policy

register `round_robin`

example: round-robin load-balancer

```
class rrlb(DynamicPolicy):

 def __init__(self,s,servers):
  self.switch = s
  self.servers = servers
  ...

  Q = packets(limit=1,group_by=['srcip'])
  Q.register_callback(self.round_robin)

  self.policy = match(dstport=80) >> Q

 def round_robin(self,pkt):
  self.policy = if_(match(srcip=pkt['srcip']),
                    modify(dstip=self.server), self.policy)
  self.client += 1
  self.server = self.servers[self.client % m]


servers = ['2.2.2.8','2.2.2.9']
rrlb_on_switch3 = rrlb(3,servers)
```

create a query policy

register round_robin

update server assignment

example: round-robin load-balancer

```
class rrlb(DynamicPolicy):

 def __init__(self,s,servers):
  self.switch = s
  self.servers = servers
  ...

  Q = packets(limit=1,group_by=['srcip'])
  Q.register_callback(self.round_robin)
  self.policy = match(dstport=80) >> Q

 def round_robin(self,pkt):
  self.policy = if_(match(srcip=pkt['srcip']),
                    modify(dstip=self.server), self.policy)
  self.client += 1
  self.server = self.servers[self.client % m]


servers = ['2.2.2.8','2.2.2.9']
rrlb_on_switch3 = rrlb(3,servers)
```

create a query policy

register round_robin

update server assignment

3 runs rrlb for two servers

Pyretic revisit:
abstract topology abstraction

topology abstraction revisited

modular programming constrains

- what each module sees and can do

enabled by network objects

- an abstract topology
  - can be a mix of physical and virtual switches
  - can be multiple levels of nesting on top of the one real network
- a policy function applied to this topology

MAC learning

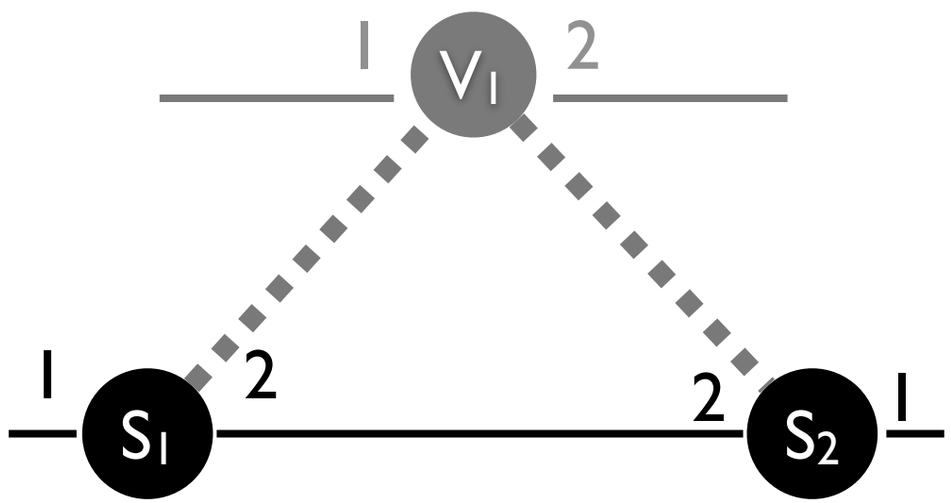
- see one big switch, learns where the hosts are located

switching

- sees the entire physical network
- performs routing from one edge link to another

MAC learning
(derived network)

switching fabric
(underlying network)



coordinating modules

MAC learning module

- specifies chosen output port(s)

switching module

- directs traffic on a path to the (corresponding) egress port(s)

coordination via abstract packet model

virtual packet header

- a module can push, pop, and inspect
- MAC learning directs traffic from one input port to an output port
- switching sees a virtual header indicating the corresponding ingress/egress
- runtime performs the mapping

example: transforming topology

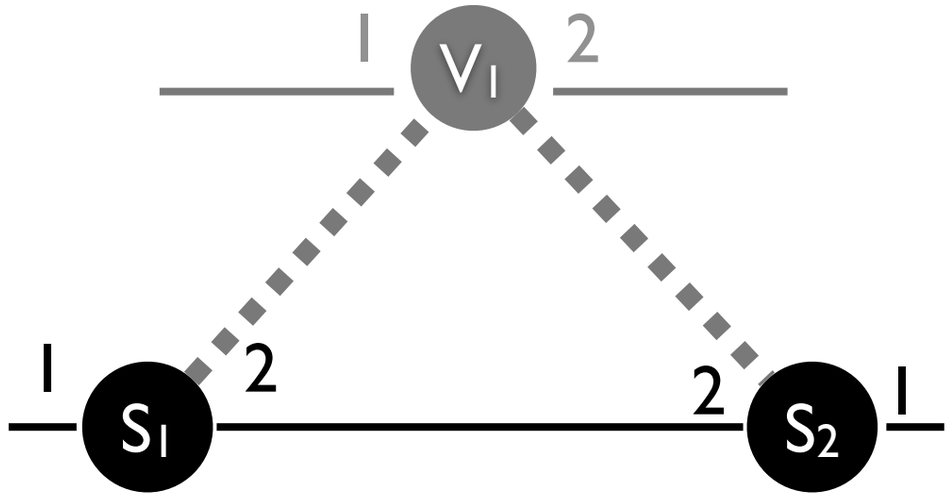
a dictionary (mapping)

- (vswitch, vport) ➡ (switch, port)

  derived network underlying network

derived network

underlying network

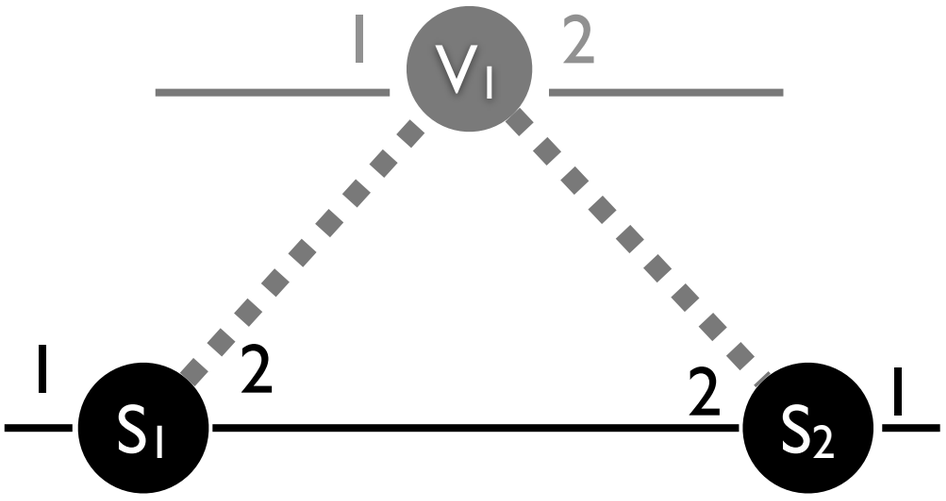


example: transforming topology

a dictionary (mapping)

- (vswitch, vport) ➡ (switch, port)

derived network underlying network



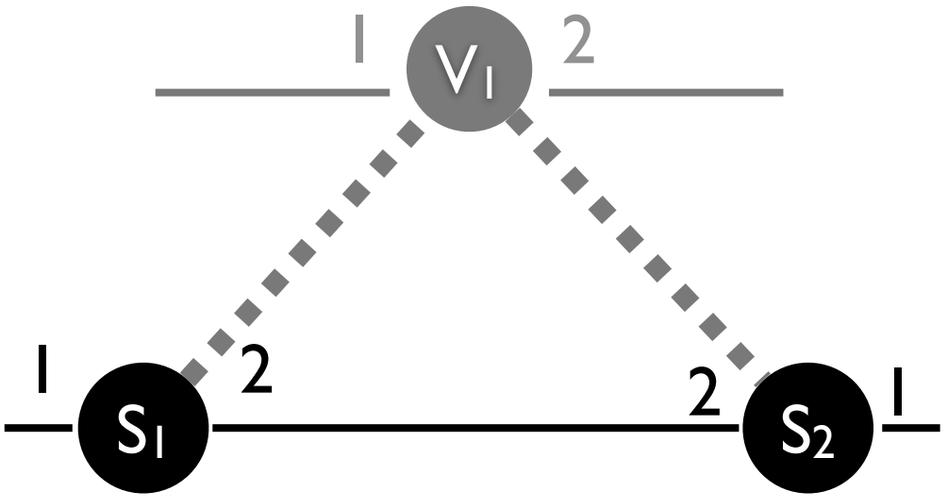

```
def bfs_vmap(topo):
    vswitch = 1
    vport = 1
    for (switch, port) in topo.egress_locations:
        vmap[(vswitch, vport)] = (switch, port)
        vport += 1
    return vmap
```

recall the packet model {switch: A, inport: 3, vswitch: V, ...}

example: transforming policy

Pyretic generates ingress function

- "lifts" packets from the underlying network up to the derived

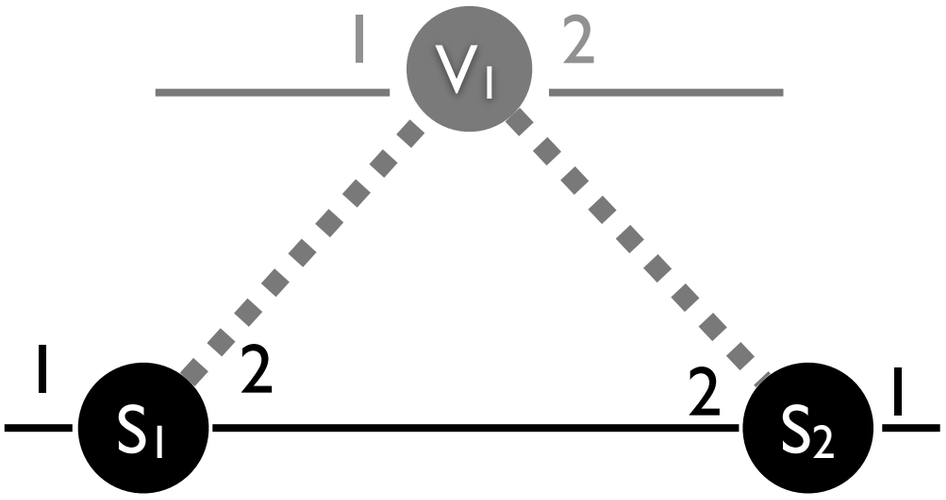



```
ingress_policy =
( match(switch=S1, inport=1)
   [push(vswitch=V, vinport=1)]
| match(switch=S2, inport=1)
   [push(vswitch=V, vinport=2)])
```

example: transforming policy

Pyretic generates egress function

- "lowers" packets from the derived to the underlying network

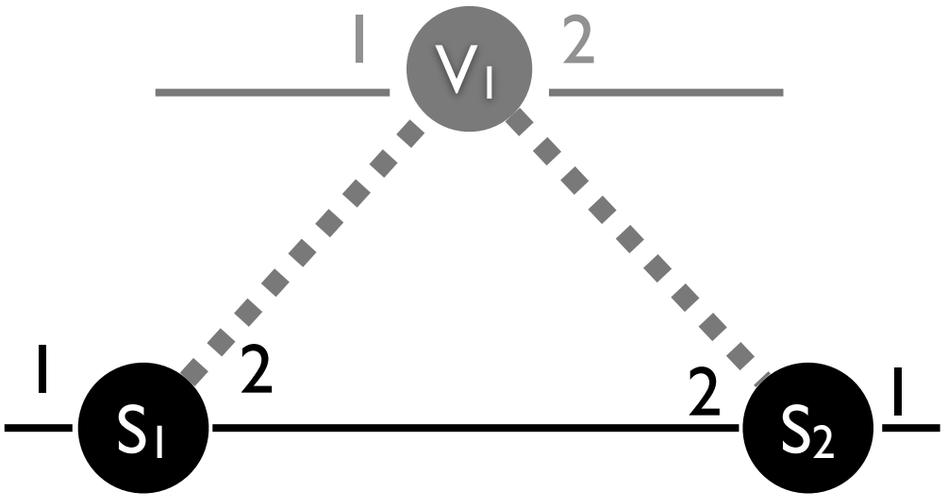



```
egress_policy = match(vswitch=V)
    [if_(match(switch=S1, voutport=1)
      | match(switch=S2, voutport=2),
      pop(vswitch, vinport, voutport),
      passthrough)]
```

example: transforming policy

Pyretic generates fabric policy

- forwarding between ports in the derived network via a path in the underlying network




```
fabric_policy = match(vswitch=V)[
 ( match(switch=S1, voutport=1)[fwd(1)]
  |match(switch=S1, voutport=2)[fwd(2)]
  |match(switch=S2, voutport=1)[fwd(2)]
  |match(switch=S2, voutport=2)[fwd(1)])]
```

virtualizing template

```
def virtualize(ingress_policy,
                   egress_policy,
                   fabric_policy,
                   derived_policy):
    return if_(~match(vswitch=None),
      (ingress_policy >>
       move(switch=vswitch,inport=vinport) >>
       derived_policy >>
       move(vswitch=switch,vinport=inport,voutport=outport)),
     passthrough) >>
fabric_policy >>
egress_policy
```

virtualizing template

```
def virtualize(ingress_policy,
               egress_policy,
               fabric_policy,
               derived_policy):
    return if_(~match(vswitch=None),
      (ingress_policy >>
       move(switch=vswitch,inport=vinport) >>
       derived_policy >>
       move(vswitch=switch,vinport=inport,voutport=outport)),
     passthrough) >>
fabric_policy >>
egress_policy
```

typo?
“~”

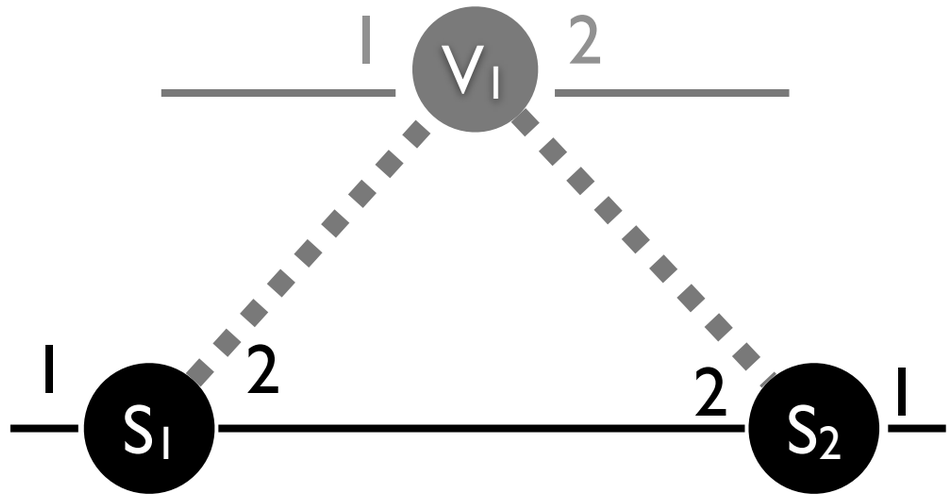
virtualizing template

```
def virtualize(ingress_policy,
                   egress_policy,
                   fabric_policy,
                   derived_policy):
    return if_(~match(vswitch=None),
      (ingress_policy >>
       move(switch=vswitch,inport=vinport) >>
       derived_policy >>
       move(vswitch=switch,vinport=inport,voutport=outport)),
     passthrough) >>
fabric_policy >>
egress_policy
```

virtualizing template

```
def virtualize(ingress_policy,
                    egress_policy,
                    fabric_policy,
                    derived_policy):
    return if_(~match(vswitch=None),
      (ingress_policy >>
       move(switch=vswitch,inport=vinport) >>
       derived_policy >>
       move(vswitch=switch,vinport=inport,voutport=outport)),
     passthrough) >>
    fabric_policy >>
    egress_policy
```

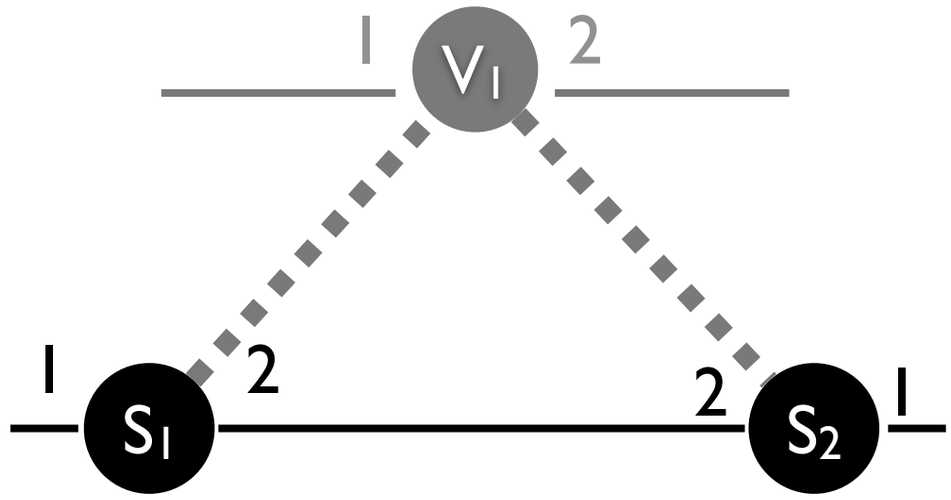
packet processing



{switch:S1, inport:1, … }

```
ingress_policy >>

move(switch=vswitch,inport=vinport)>>

derived_policy (flood) >>

move(vswitch=switch,vinport=inport,vouptport=outport) >>

passthrough

fabric_policy >>

egress_policy
```

packet processing



{switch:S1, inport:1, … }

ingress_policy >>

{switch:S1, inport:1, vswitch:V, vinport:1, … }

move(switch=vswitch,inport=vinport)>>

derived_policy (flood) >>

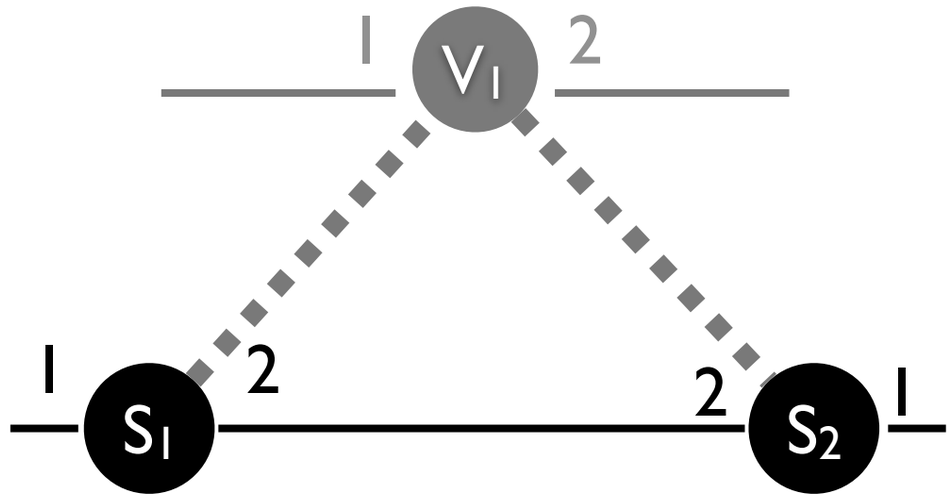
move(vswitch=switch,vinport=inport,vouptport=outport) >>

passthrough

fabric_policy >>

egress_policy

packet processing



{switch:S1, inport:1, … }

ingress_policy >>

{switch:S1, inport:1, vswitch:V, vinport:1, … }

move(switch=vswitch,inport=vinport)>>

{switch:[V, S1], inport:[1, 1]}

derived_policy (flood) >>

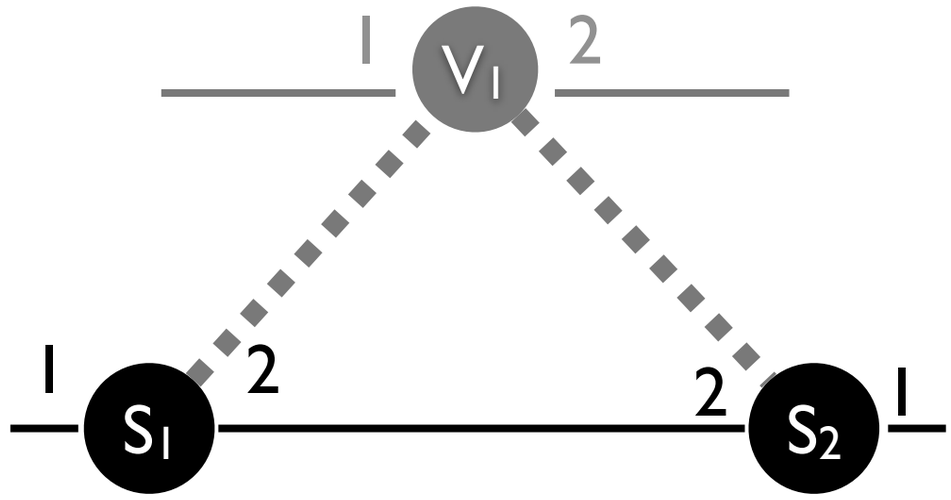
move(vswitch=switch,vinport=inport,vouptport=outport) >>

passthrough

fabric_policy >>

egress_policy

packet processing



{switch:S1, inport:1, … }

ingress_policy >>

{switch:S1, inport:1, vswitch:V, vinport:1, … }

move(switch=vswitch,inport=vinport)>>

{switch:[V, S1], inport:[1, 1]}

derived_policy (flood) >>

{switch:[V, S1], inport:[1, 1], outport:2, …}

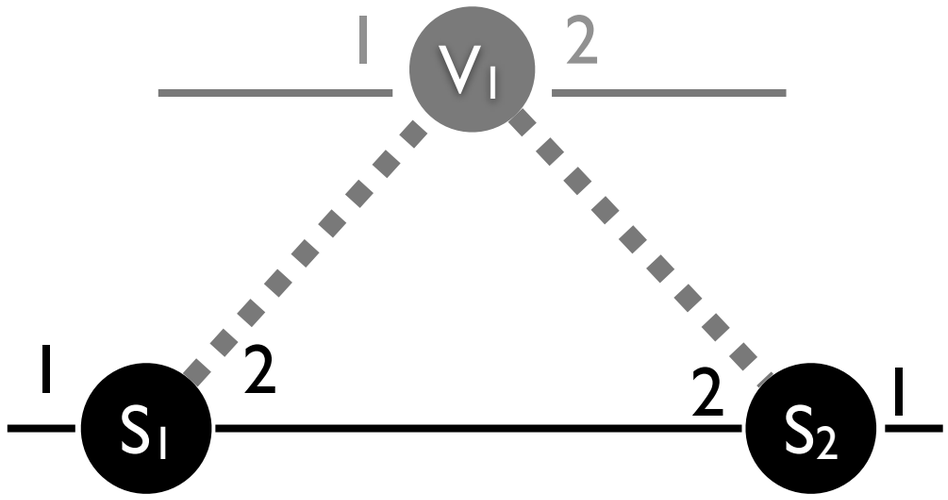
move(vswitch=switch,vinport=inport,vouptport=outport) >>

passthrough

fabric_policy >>

egress_policy

packet processing



{switch:S1, inport:1, … }

ingress_policy >>

{switch:S1, inport:1, vswitch:V, vinport:1, … }

move(switch=vswitch,inport=vinport)>>

{switch:[V, S1], inport:[1, 1]}

derived_policy (flood) >>

{switch:[V, S1], inport:[1, 1], outport:2, …}

move(vswitch=switch,vinport=inport,vouptport=outport) >>

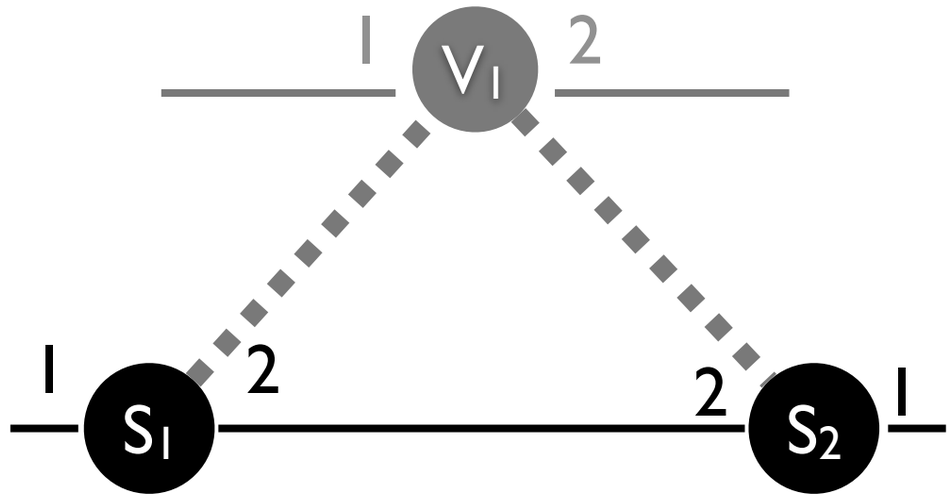
{switch:S1, inport:1, vswitch:V, vinport:1, voutport:2 }

fabric_policy >>

egress_policy

passthrough

packet processing



{switch:S1, inport:1, … }

ingress_policy >>

{switch:S1, inport:1, vswitch:V, vinport:1, … }

move(switch=vswitch,inport=vinport)>>

{switch:[V, S1], inport:[1, 1]}

derived_policy (flood) >>

{switch:[V, S1], inport:[1, 1], outport:2, …}

move(vswitch=switch,vinport=inport,vouptport=outport) >>

{switch:S1, inport:1, vswitch:V, vinport:1, voutport:2 }

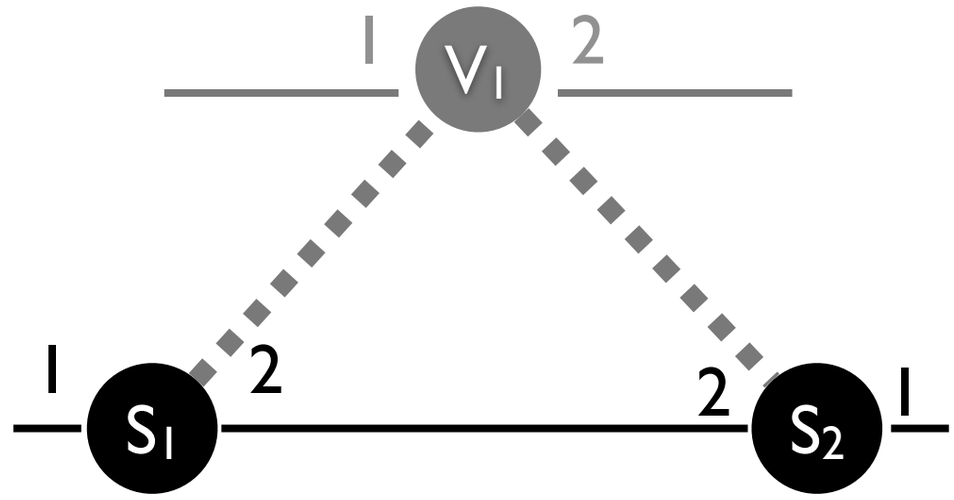
fabric_policy >>

{switch:S1, outport:2,vswitch:V, vinport:1, voutport:2 }

egress_policy

passthrough

packet processing



{switch:S1, inport:1, … }

ingress_policy >>

{switch:S1, inport:1, vswitch:V, vinport:1, … }

move(switch=vswitch,inport=vinport)>>

{switch:[V, S1], inport:[1, 1]}

derived_policy (flood) >>

{switch:[V, S1], inport:[1, 1], outport:2, …}

move(vswitch=switch,vinport=inport,vouptport=outport) >>

{switch:S1, inport:1, vswitch:V, vinport:1, voutport:2 }

fabric_policy >>

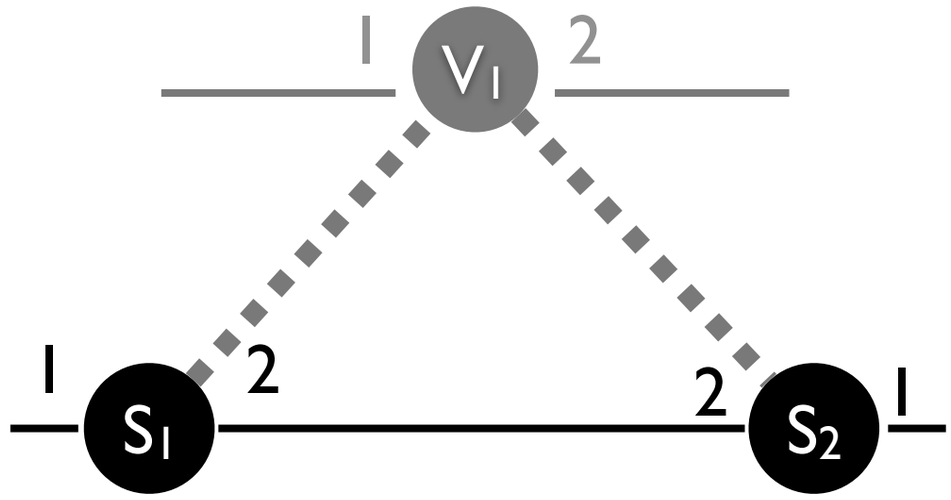
{switch:S1, outport:2,vswitch:V, vinport:1, voutport:2 }

egress_policy

{switch:S1, outport:2,vswitch:V, vinport:1, voutport:2 }

passthrough

packet processing



{switch:S1, inport:1, … }

ingress_policy >>

{switch:S1, inport:1, vswitch:V, vinport:1, … }

move(switch=vswitch,inport=vinport)>>

{switch:[V, S1], inport:[1, 1]}

derived_policy (flood) >>

{switch:[V, S1], inport:[1, 1], outport:2, …}

move(vswitch=switch,vinport=inport,vouptport=outport) >>

{switch:S1, inport:1, vswitch:V, vinport:1, voutport:2 }

passthrough

fabric_policy >>

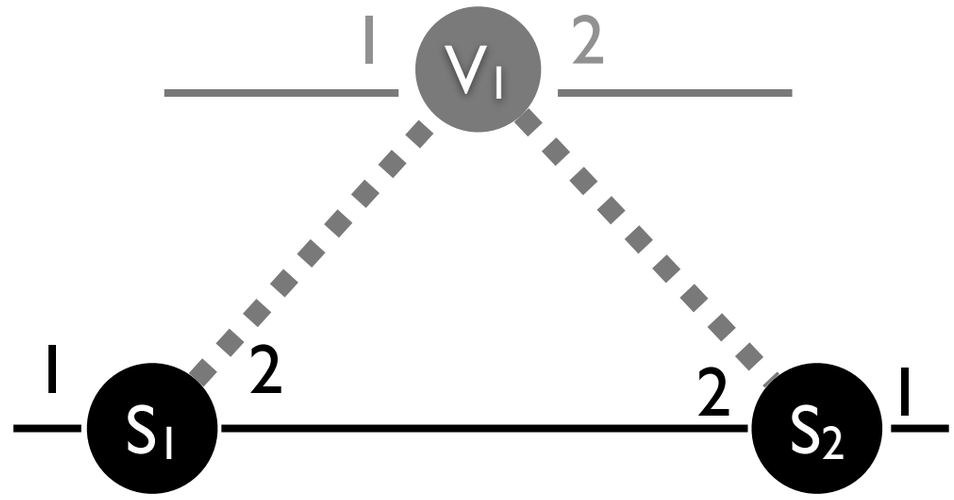
{switch:S1, outport:2,vswitch:V, vinport:1, voutport:2 }

egress_policy

{switch:S1, outport:2,vswitch:V, vinport:1, voutport:2 }

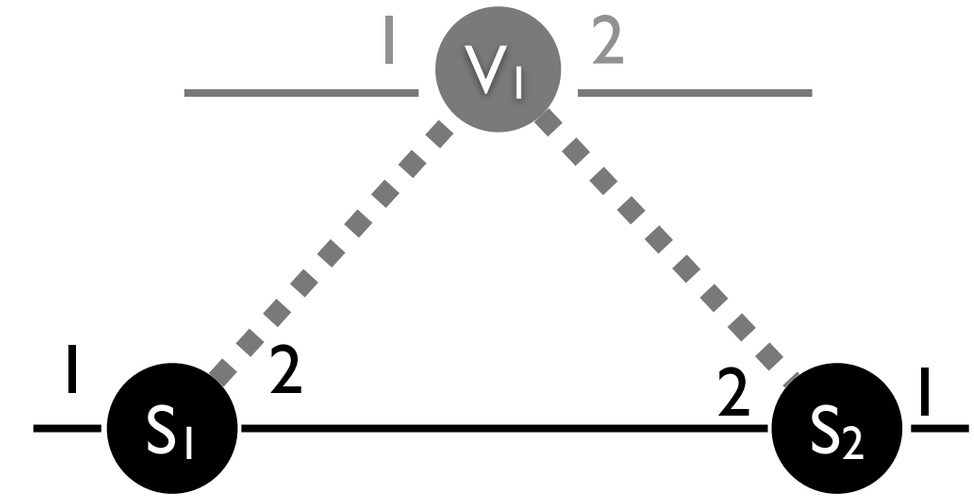
{switch:S2, inport:2,vswitch:V, vinport:1, voutport:2 }

packet processing




```
ingress_policy >>

move(switch=vswitch,inport=vinport)>>

derived_policy (flood) >>

move(vswitch=switch,vinport=inport,vouptport=outport) >>

fabric_policy >>

egress_policy

passthrough
```

packet processing

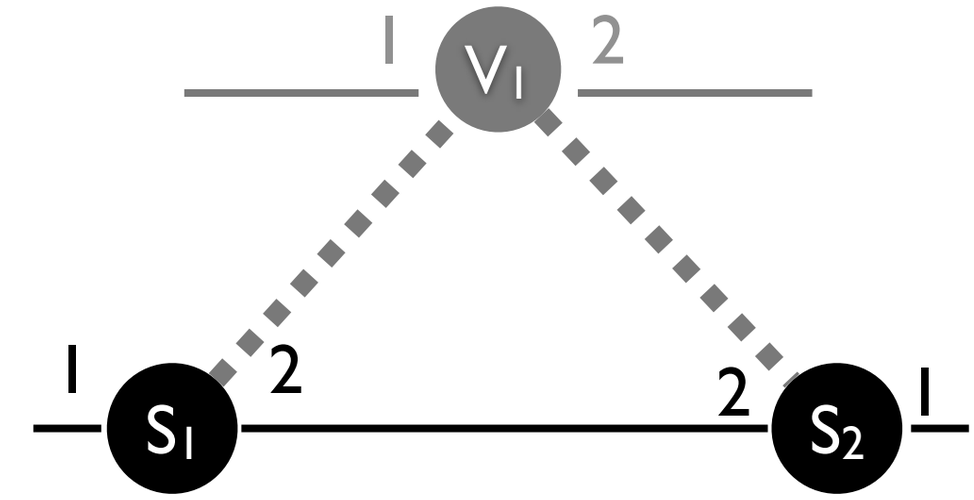



```
ingress_policy >>

move(switch=vswitch,inport=vinport)>>

derived_policy (flood) >>

move(vswitch=switch,vinport=inport,vouptport=outport) >>

fabric_policy >>

egress_policy
```

{switch:S2, inport:2,vswitch:V, vinport:1, voutport:

passthrough

packet processing



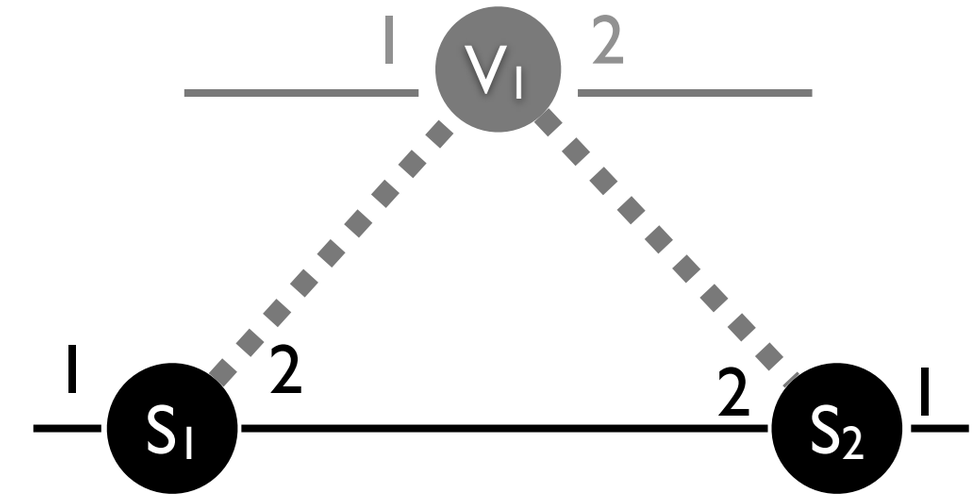

```
ingress_policy >>

move(switch=vswitch,inport=vinport)>>

derived_policy (flood) >>

move(vswitch=switch,vinport=inport,vouptport=outport) >>

fabric_policy >>

egress_policy
```

{switch:S2, inport:2,vswitch:V, vinport:1, voutport:

passthrough

{switch:S2, inport:2,vswitch:V, vinport:1, voutport:2

packet processing



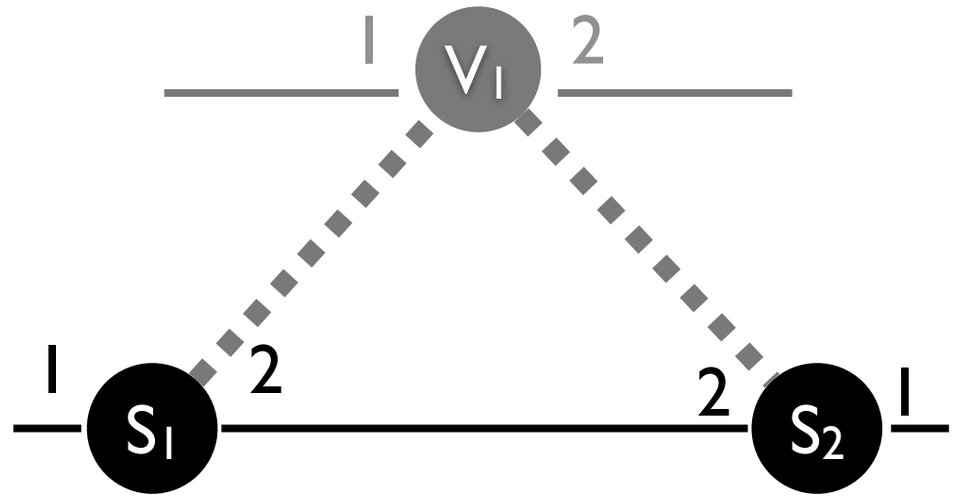

```
ingress_policy >>

move(switch=vswitch,inport=vinport)>>

derived_policy (flood) >>

move(vswitch=switch,vinport=inport,vouptport=outport) >>

fabric_policy >>

egress_policy

passthrough
```

{switch:S2, inport:2,vswitch:V, vinport:1, voutport:

{switch:S2, inport:2,vswitch:V, vinport:1, voutport:2

{switch:S2, inport:2,vswitch:V, output: 1, vinport:1, voutport:2 }

packet processing




```
ingress_policy >>

move(switch=vswitch,inport=vinport)>>

derived_policy (flood) >>

move(vswitch=switch,vinport=inport,vouptport=outport) >>

fabric_policy >>

egress_policy

passthrough
```

{switch:S2, inport:2,vswitch:V, vinport:1, voutport:

{switch:S2, inport:2,vswitch:V, vinport:1, voutport:2

{switch:S2, inport:2,vswitch:V, output: 1, vinport:1, voutport:2 }

{switch:S2, output: 1}