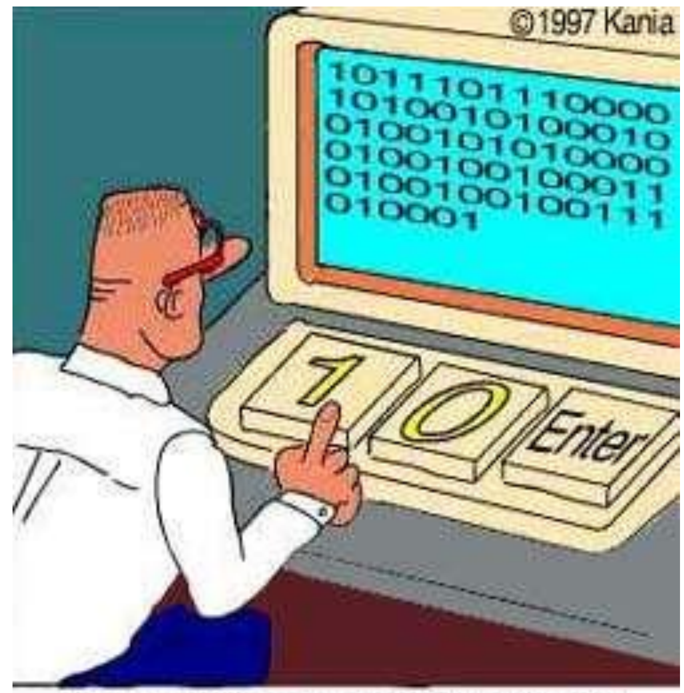
Languages for Software-Defined Networks

Nate Foster, Michael J. Freedman, Arjun Guha, Rob Harrison, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Jennifer Rexford, Cole Schlesinger, Alec Stor and David Walker

Programming the Network

- SDN gives programmers control
- Control does not imply easy to use
- Bottom line: using OpenFlow is hard



Real programmers code in binary.

Example: repeater/monitor

- We want to create a repeater that also provides counter data on network traffic
- Using OpenFlow we need to take into account the way these rules will be installed
- And how they impact each other

Repeater and monitor

- Task 1: forward port 1 to port 2 and port 2 to port 1
- Task 2: count http packets from port 2

Repeater/monitor in OpenFlow

Take them all into account!

```
def switch_join(s):
  pat1 = finport:1g
  pat2web = finport:2, srcport:80g
  pat2 = finport:2g
  install(s, pat1, DEFAULT, [fwd(2)])
  install(s, pat2web, HIGH, [fwd(1)])
  install(s, pat2, DEFAULT, [fwd(1)])
  query_stats(s, pat2web)
```

Making changes must be fun…

What we need is…

- An abstraction!
- The Frenetic family:
  - Pyretic (python)
  - Frenetic-OCaml
- Write modular programs
- Get statistics without polling for them explicitly

Operations needed

1. Querying network state
2. Expressing Policies
3. Reconfiguring the network

(All these will need to be supported by the runtime)

Operations: Querying

- A desired query might require a series of switch rules:
  - Statistics by source IP or by flow (install on the go)
  - Compound rules
- No polling for the data:
  - Turning queries into event-driven programming
  - “Every” keyword
- Packets might collect at the controller while the rules are being installed
  - Any identical packet arriving while processing should wait

Aggregate type: what type of counter to install

```
Select(bytes) *
Where(inport=2 & srcport=80) *
GroupBy([srcip]) *
Every(60)
Limit(1)
```

Header: one counter each by this field

Event instead of polling: when to raise the event

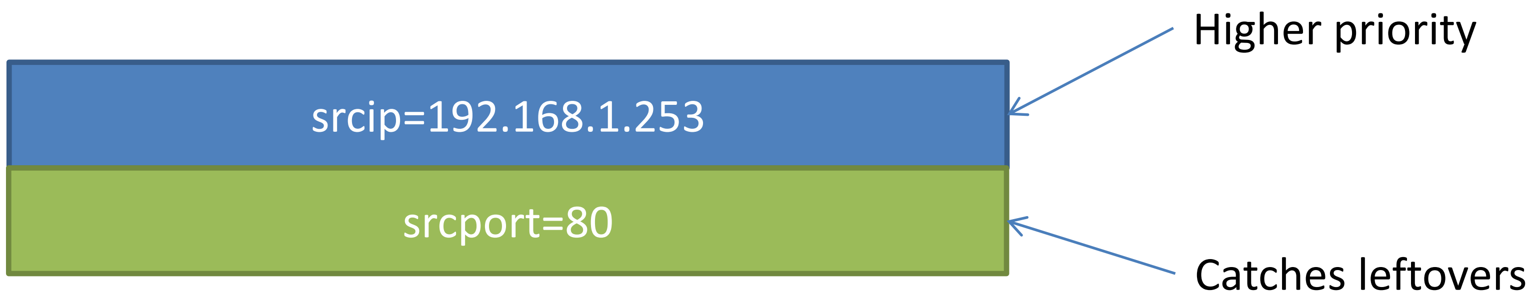
Limit packets to the controller: only show this many packets

A Query Language: Design

- High-level predicates
- Dynamic unfolding
- Limiting traffic
- Polling and combining statistics

High level predicates

- Something as simple as negation requires an elaborate hierarchy of rules:



- Some complex rules can be optimized in the switch table, the programmer doesn’t have to worry about that
- GroupBy predicates: one rule per item

Dynamic unfolding

- Limited space for rules on the switch
- Some counters require many rules to be installed on the fly
  - Counters by source IP require 2^{32} counters…
- GroupBy in Frenetic: rules will be installed on the fly without the programmer spelling it out

Limiting traffic

- The controller and the switch do not communicate instantly
- Packets can queue up to the controller before the first one is handled
- Instead of having to code the controller ignoring the second packet:
  - Limit(1)

Polling and combining

- Statistics are often checked periodically
- Stats are also spread on many switches
- Better event-driven than polled
- Every (60):
  - Every 60 seconds
  - Collect info from all the switches
  - And raise a program event

Operations: Network Policies

- Different network policies might have rules that interfere with queries
  - Or with each other
- We already saw our repeater/monitor would need three rules:
  - Inport1
  - Inport2web
  - Inport2
- Priorities matter!
- It gets worse the more complex the functionality

Modularity: back to our example

- Write different rules and queries side by side
- Don’t have to take them into account
- Put them together:

```
def repeater():
    …
def monitor():
    …
def main():
    repeater()
    monitor()
```

Can completely re-do this code without touching anything else!

Putting it together

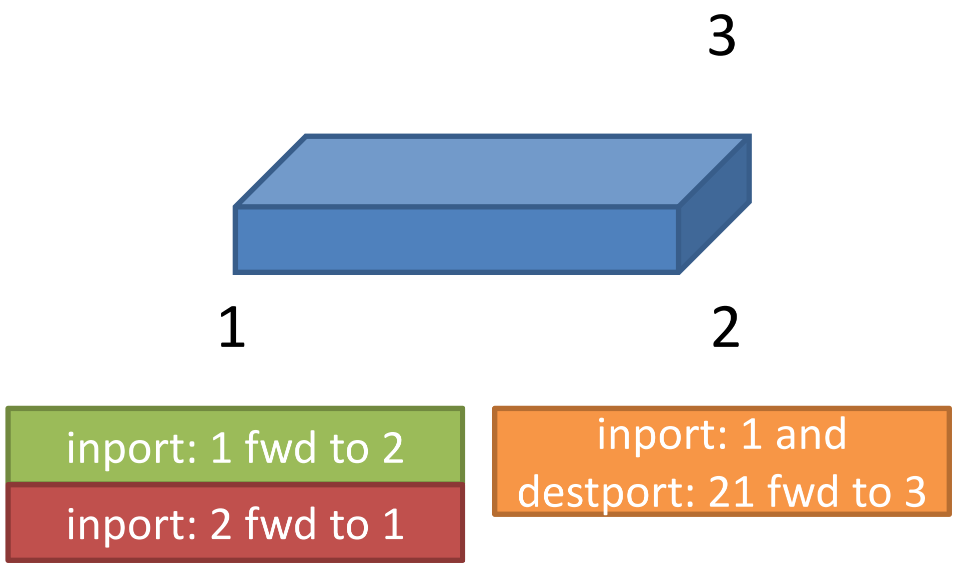
- Composing different rules is delegated to the runtime system
- We trust it to compose them “correctly”
- Composition types:
  - Parallel composition: both sets of rules on the same stream of packets
  - Sequential composition: one module acts on the output of the other

Parallel Composition

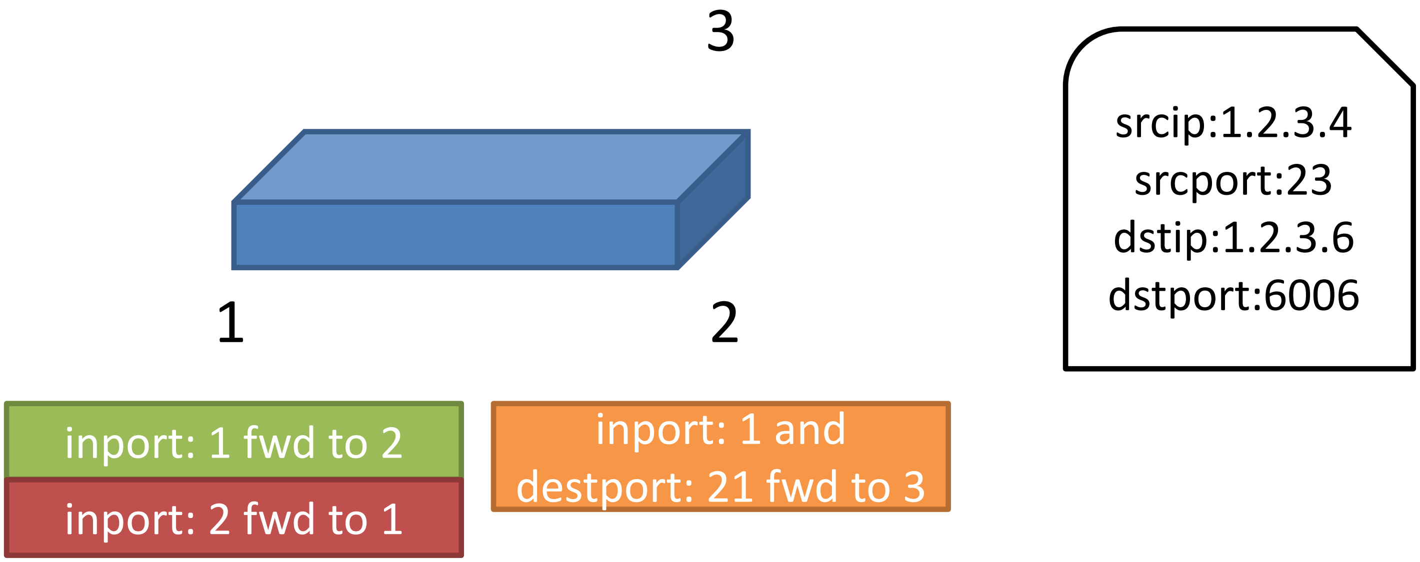
- Two modules that work on the same packets
- For instance: the repeater and the monitor, or replication
- If both modules produce forwarding rules, the resulting rule is the union

```
def main():
    return p1() | p2()
```

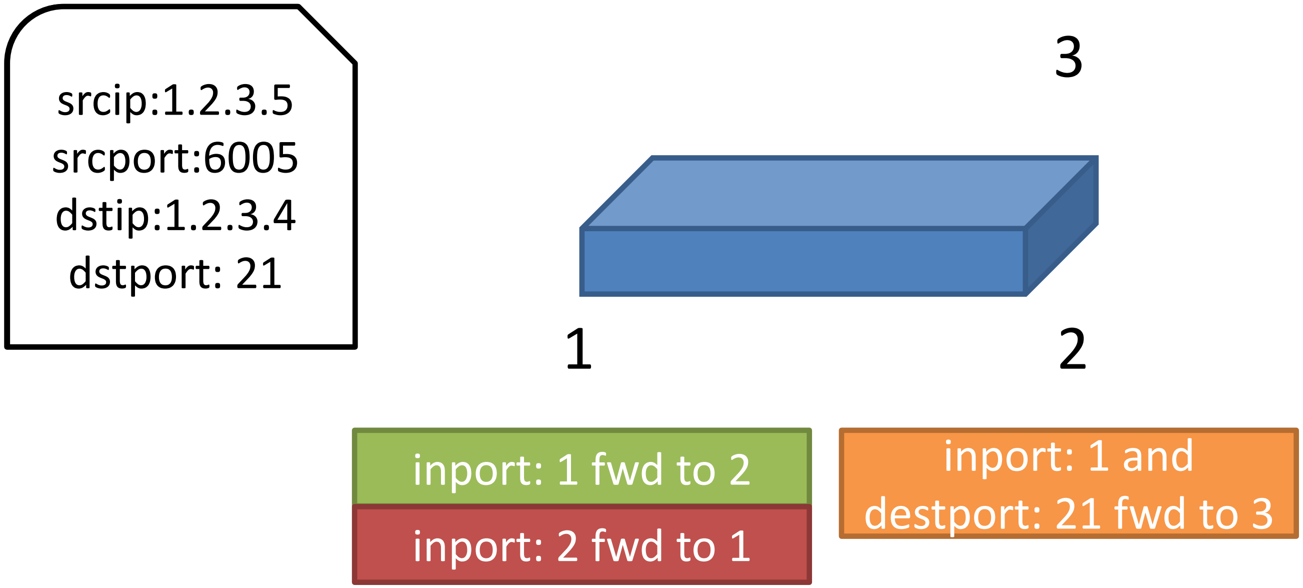
Example: replicate all ftp traffic



Example: replicate all ftp traffic



Example: replicate all ftp traffic



Sequential Composition

- Rules run one after the other: packets left after running the first go on to the next, etc.
- Example: when creating a firewall

```
def main():
        return p1() >> p2()
```

Example: firewall

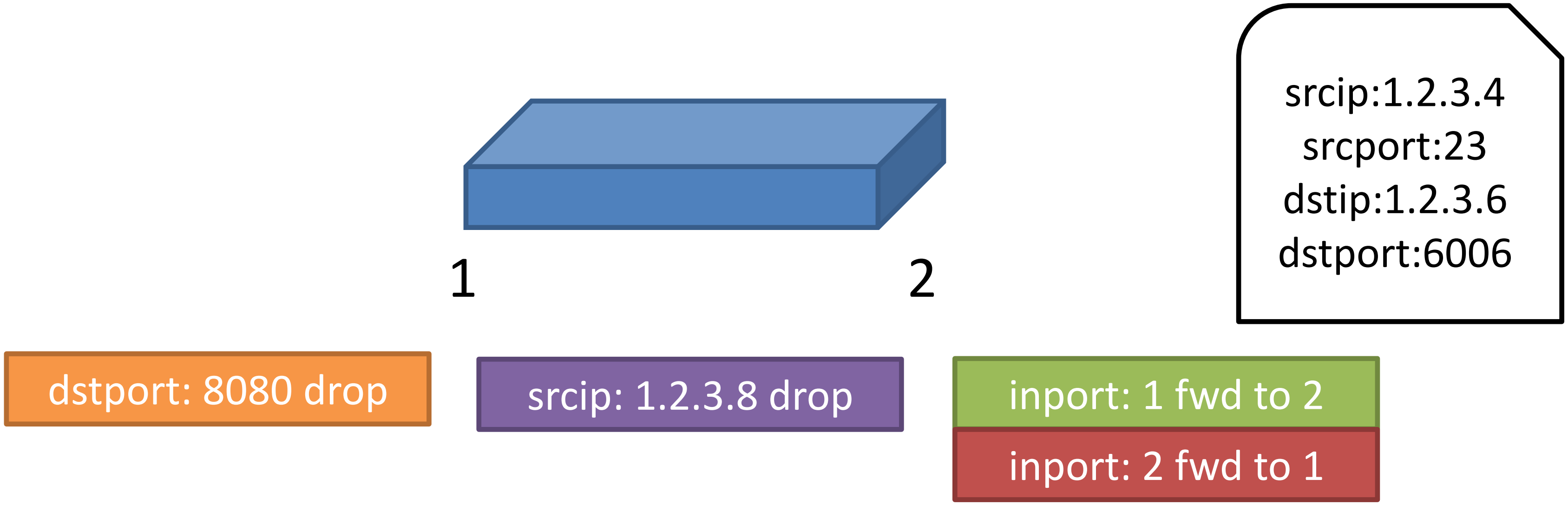
dstport: 8080 drop

srcip: 1.2.3.8 drop

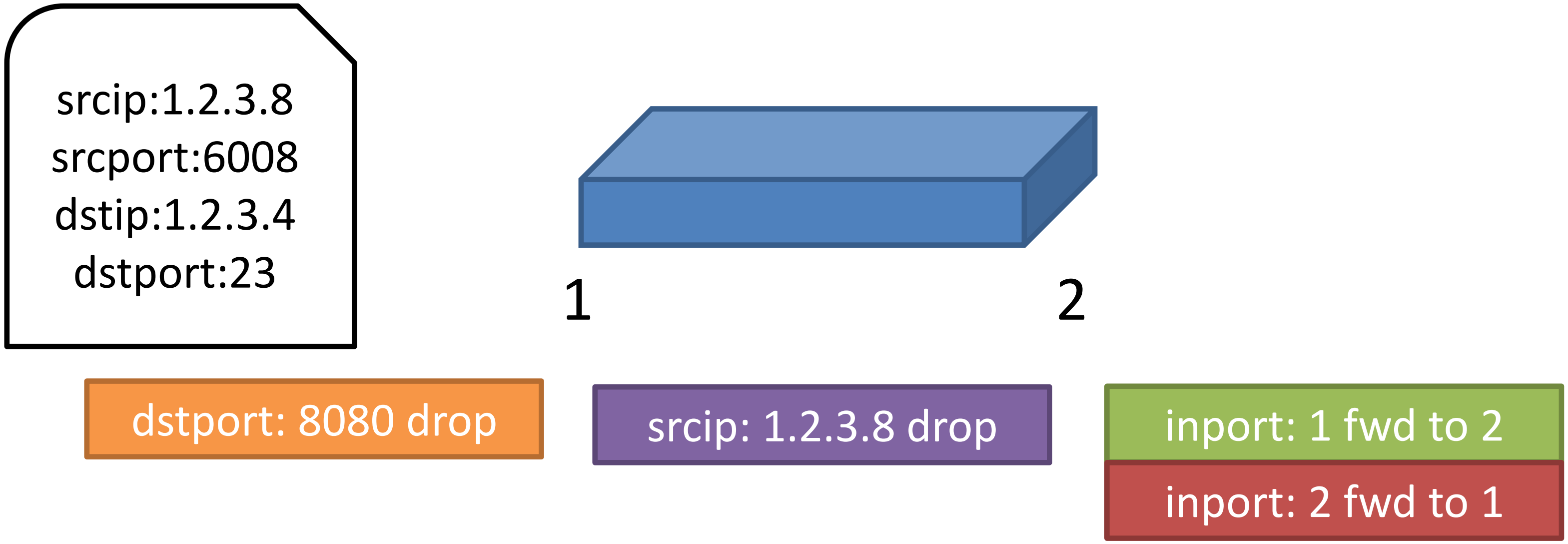
inport: 1 fwd to 2

inport: 2 fwd to 1

Example: firewall



Example: firewall

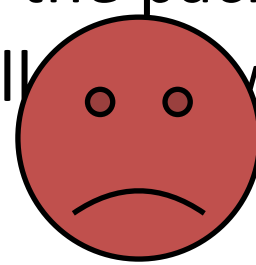


What the runtime system does

- The runtime system is the code that runs behind the programmer's code
- Something that implements the complex functionalities that the user code uses
- In the case of Frenetic: a python/Ocaml library whose code runs behind the abstract ops
- Like JVM: provides the implementation for "system" functionality

The Runtime System: Suggested Impl (microflow)

- When a packet comes in:
  - Test all queries and registered forwarding policies
  - Collect actions for the switch
- If no queries need packets like this:
  - Install forwarding rules
- If other queries might need packets like this
  - Manually forward the packet, but install no rule
  - Future packets wil[...] warded to the controller again



Runtime System: An Efficient Impl

- Instead of dynamically unfolding all the rules
- Generate rules (with wildcards) before packets are ever seen
- Proactive, not reactive
- Frenetic uses NetCore: another abstraction over OpenFlow
- When can’t be generated ahead of time: *reactive specialization* (a form of unfolding)

Operations: Consistency of updates

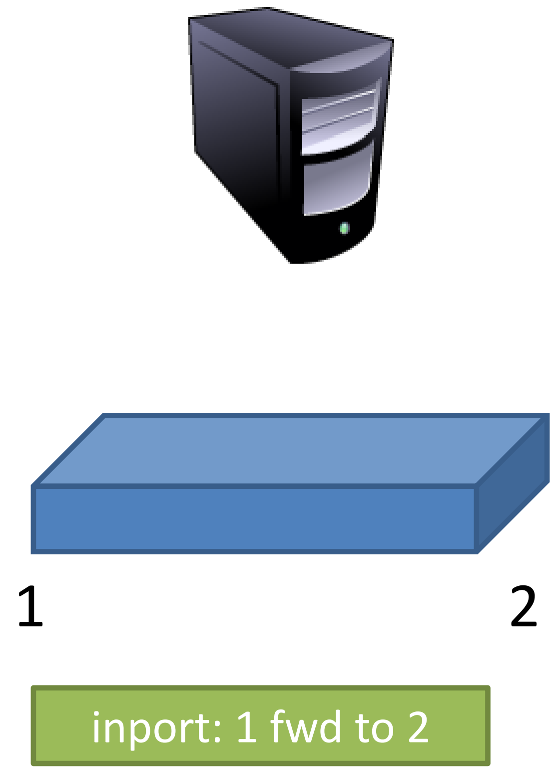
- Per packet consistency: every packet will be processed with exactly one set of rules throughout the network
  - Two phase update of the network
  - Packets are stamped with a version number for the rule set in the header

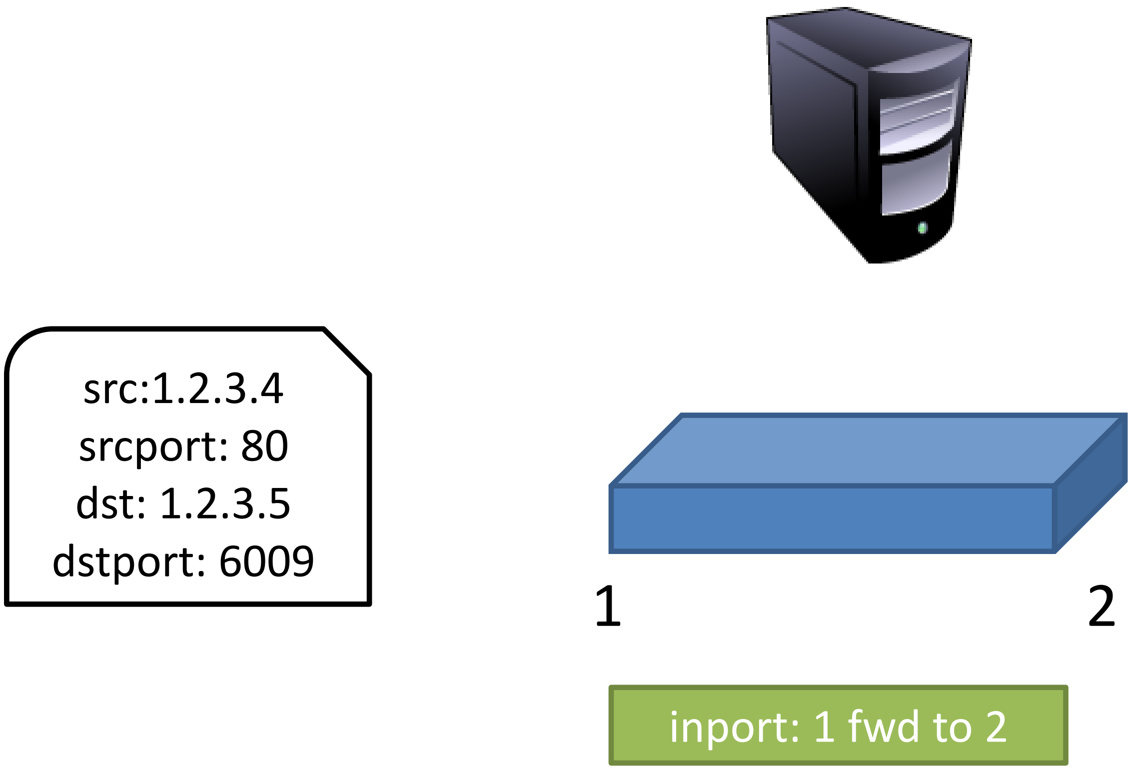
Consistency of updates

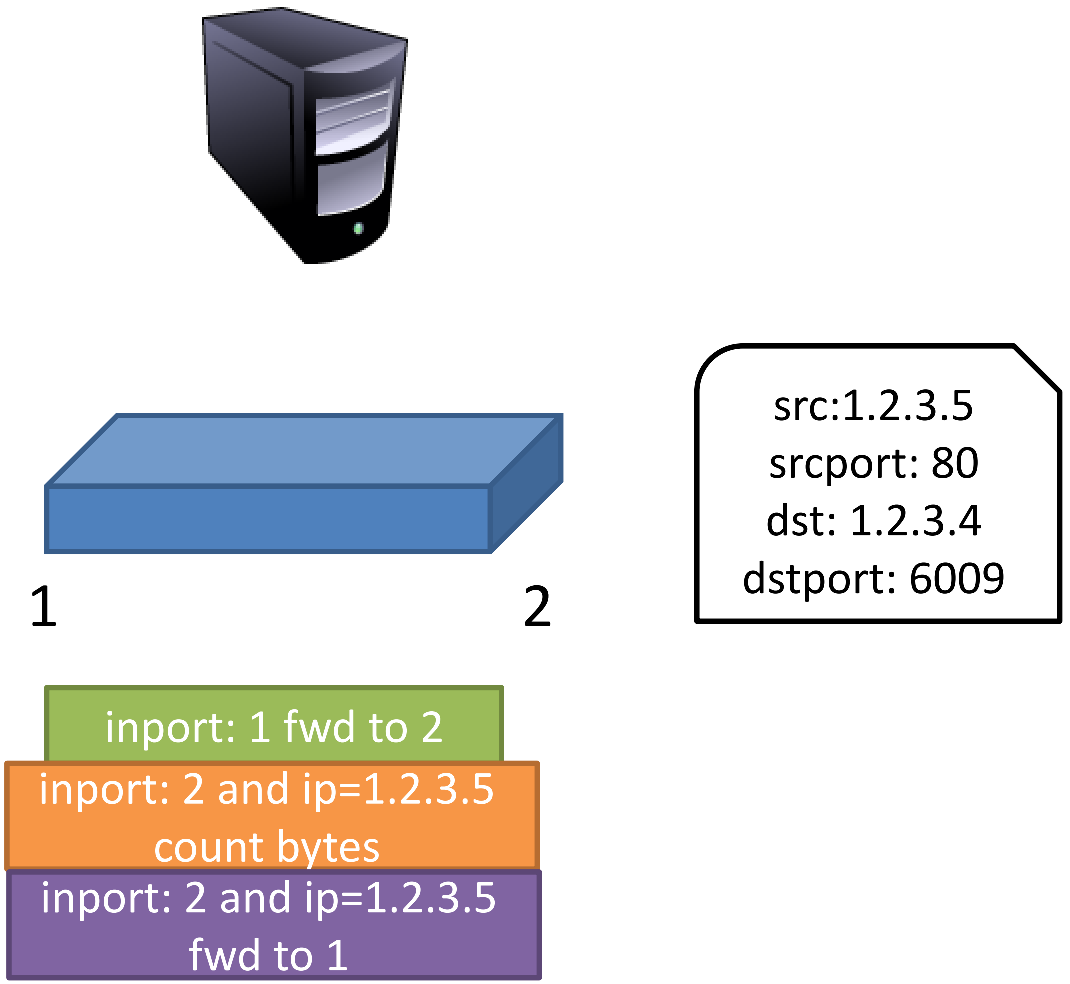
- Per flow consistency:
  - Sometimes whole streams need to be handled consistently (e.g. load balancing)
  - Rules expire only when all flows matching an old configuration are finished

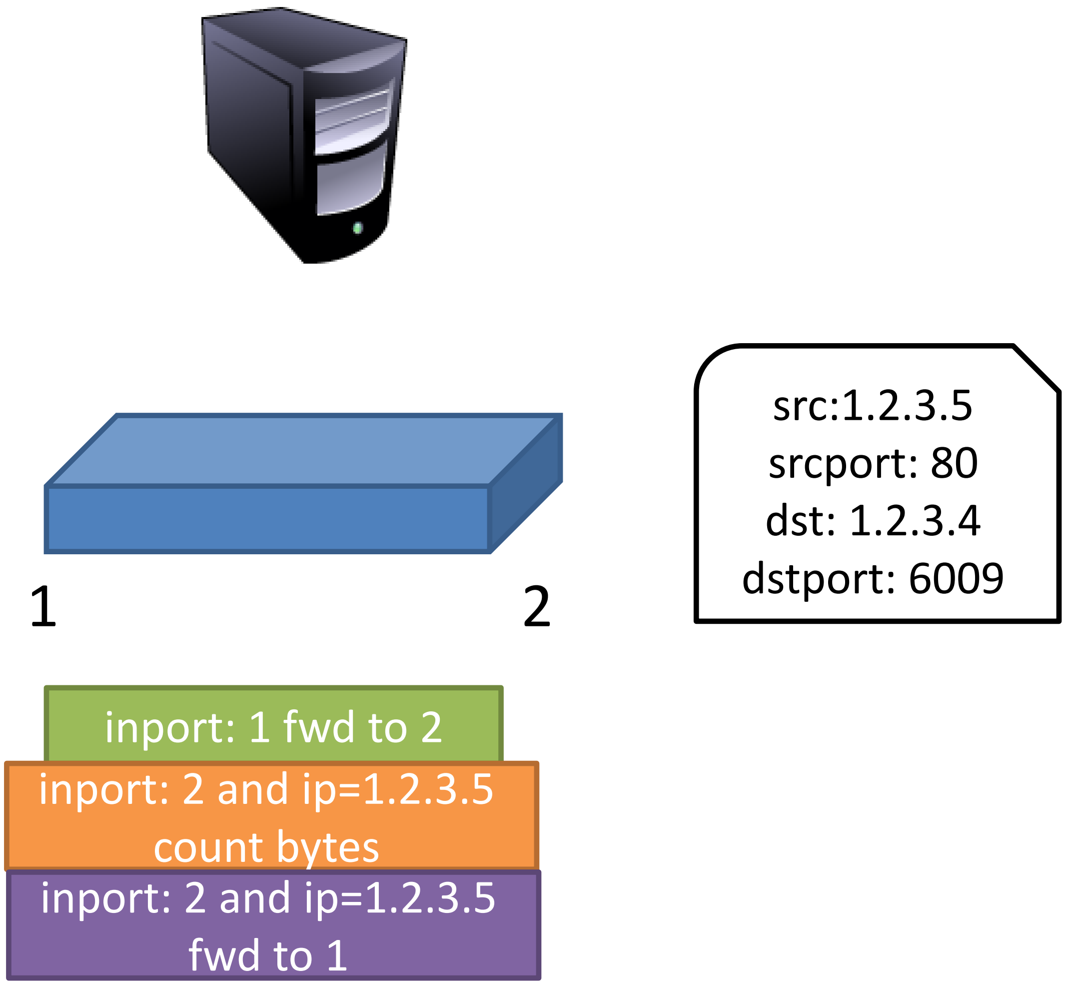
Back to repeater/monitor

```
def repeater():
      rules=[Rule(inport:1, [fwd(2)]),
            Rule(inport:2, [fwd(1)])]
      register(rules)
def web monitor():
      q = (Select(bytes) *
            Where(inport=2 & srcport=80) *
            Every(30))
      q >> Print()
def main():
      repeater()
      monitor()
```


1
2
inport: 1 fwd to 2


src:1.2.3.4
srcport: 80
dst: 1.2.3.5
dstport: 6009
1
2
inport: 1 fwd to 2


src:1.2.3.5
srcport: 80
dst: 1.2.3.4
dstport: 6009
1
2
inport: 1 fwd to 2
inport: 2 and ip=1.2.3.5
count bytes
inport: 2 and ip=1.2.3.5
fwd to 1


src:1.2.3.5
srcport: 80
dst: 1.2.3.4
dstport: 6009
1
2
inport: 1 fwd to 2
inport: 2 and ip=1.2.3.5
count bytes
inport: 2 and ip=1.2.3.5
fwd to 1

Additional Refernces

- Frenetic: A Network Programming Language (Foster et. al, 2011)
- Composing Software-Defined Network (Monsanto et. al, 2013)
- A Compiler and Run-time System for
- Network Programming Languages (Monsanto et. al, 2012)
- http://frenetic-lang.org