Ch.6 SDN(2)
Arch & APP

Outline

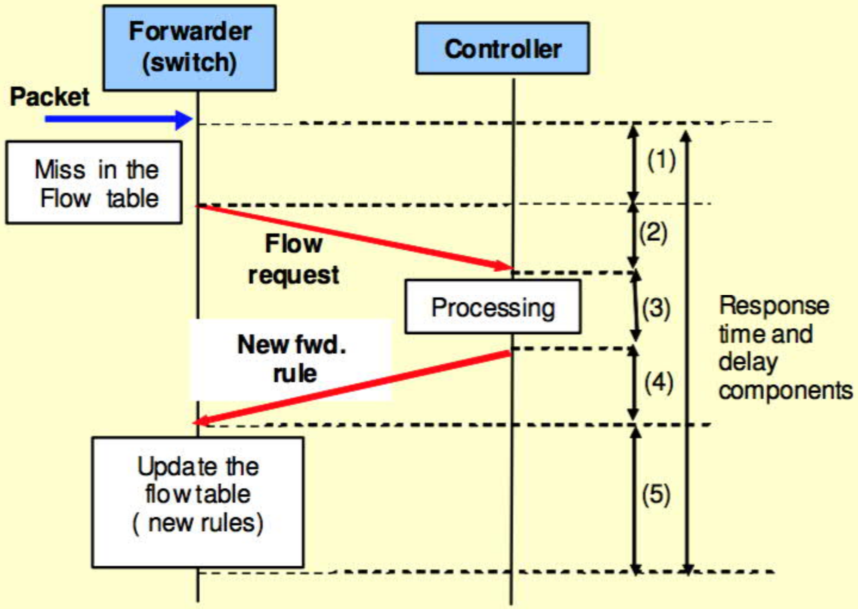
- Arch
  - Scalability
  - Programming
- App
  - Security

3. Control Plane Scalability in SDN

SDN Problem: new flows setup - response time

- Signaling overhead components
  - Switch (CPU, mem, ..)- (1), (5)
  - Controller (CPU, mem, ..) - (3)
  - Transport through network – (2), (4)

  - Transport: (2), (4) –> place controller closer to the switch
  - Switch
    - OpenVSwitch: install tens of (10**3) flows/s with < 1 ms latency
    - HW switches: install few (10**3) with 10ms latency
      - Weak mgmt CPU
      - Low speed communication CPU-switching chipset
    - Hope for faster switches
  - Frequent events can stress
    - Controller resources
    - Control channel
    - Switch



On demand flow setup

3. Control Plane Scalability in SDN

- **Solutions:**
  - **Direct solutions**
    - Increase controller processing power
    - Increase switch processing power
  - **Aggregation of rules**
  - Proactive installation of rules
    - Problems: no host mobility support, not enough memory in switches
  - **Delegate more responsibilities to the data plane**
    - to switch control plane [e.g.Diffane, DevoFlow]
  - **Distributed controllers**
    - Flat structure multiple controllers [e.g. ONIX]
    - Recursive controller design [e.g. Xbar]
    - Hierarchical controller design [e.g.Kandoo]

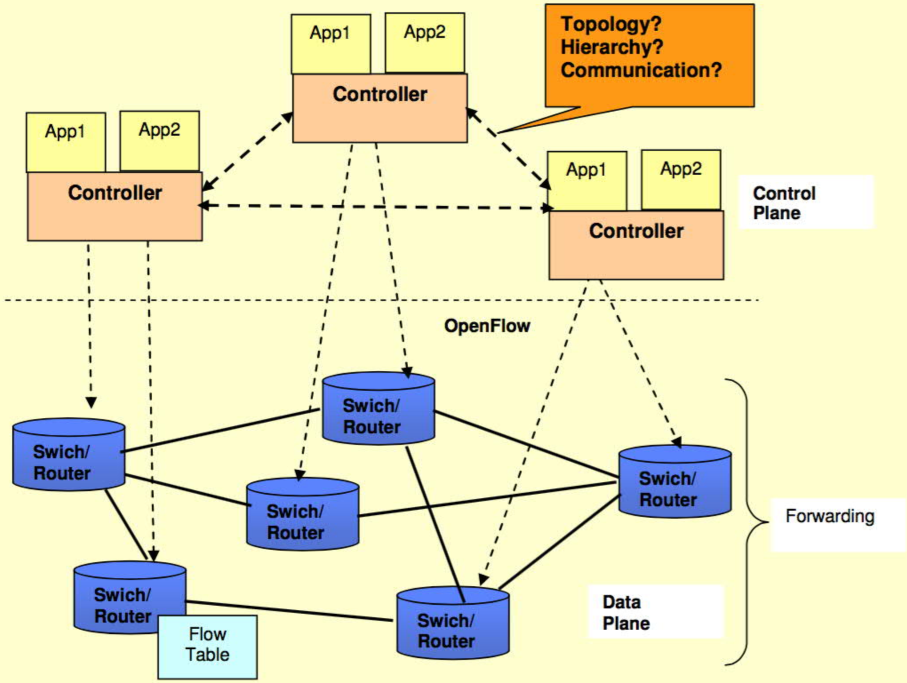
3. Control Plane Scalability in SDN

- **SDN Scalability-related solutions**
- **Increase the controller processing power**

- *Source: A. Tootoonchian et al., "On Controller Performance in Software-Defined Networks," Proc. USENIX Hot-ICE '12,2012, pp. 10–10.*
  - Multicore systems for higher level of parallelism
  - And improved IO performance
    - Simple modifications to the NOX controller → performance increase more than 10 times an order of magnitude on a single core. (w.r.t 30000 flows/sec)
    - A single controller can support larger networks, (if the controller channel has enough bandwidth and acceptable latency)

3. Control Plane Scalability in SDN

- SDN Scalability issues
  - **Multiple controller solution**
  - Control Plane and Data Plane – independent; they can have different topologies



3. Control Plane Scalability in SDN

- **Multiple controllers- (cont'd)**
- **Approaches:**
  - **Flat organization**
    - Distribution of the CPI while maintaining a logically centralized using a distributed file system, a distributed hash table and a pre-computation of all possible combinations respectively.
    - however they impose a strong requirement: a consistent network-wide view in all the controllers.
    - Examples: HyperFlow, Onix and Devolved controllers
  - **Hierarchical organization**
    - Hierarchical distribution of the controllers based on two layers:
    - (i) the bottom layer, a group of controllers with no interconnection, and no knowledge of the network-wide state,
    - (ii) the top layer, a logically centralized controller that maintains the network wide state.
    - Example: Kandoo

3. Control Plane Scalability in SDN

- **Solution proposals examples**
- **HyperFlow**
  - *Source: A. Tootoonchian et.al., "Hyperflow: A Distributed Control Plane for OpenFlow," Proc. 2010 INMConf., 2010.*
  - Among the first distributed (event-based) Control Plane for OpenFlow
  - **Logically centralized**
    - All the controllers share the same consistent network-wide view and locally serve requests without actively contacting any remote node, thus minimizing the flow setup times.
  - **Physically distributed**: scalable while keeping the network control centralization benefits
  - By passively synchronizing network-wide views of OpenFlow controllers, it localizes decision making to individual controllers, thus minimizing the CPl response time to data plane requests
  - It is resilient to network partitioning and component failures
  - It **enables interconnecting independently managed OpenFlow networks** - - an essential feature

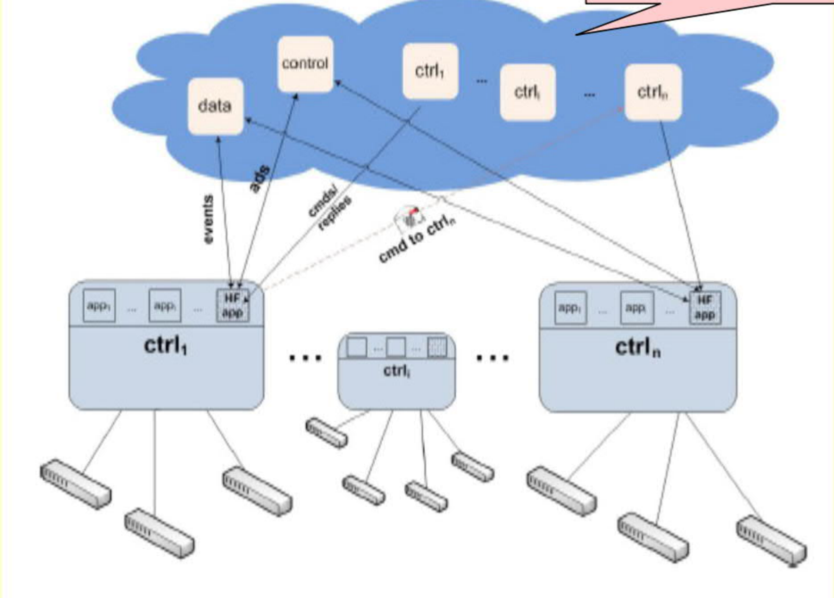
3. Control Plane Scalability in SDN

- **Solution proposals examples**
- **HyperFlow (cont'd)**
  - **Design Principles**
    - A HyperFlow (HF) -based network is composed by
      - ***OpenFlow switches/forwarders***
      - ***NOX controllers*** (decision elements) each running an instance of the HF controller application
      - ***Event propagation system*** for cross-controller communication.
    - All the controllers have a consistent network-wide view and run as if they are controlling the whole network.
    - They **all run the exact same controller software and set of applications.**
    - Each switch is connected to the closest controller
    - Controller failure → affected switches must be reconfigured to connect to an active nearby controller
    - **Each controller**
      - directly manages a subset of switches connected to it
      - indirectly programs or queries the rest (via comm.. with other controllers).

3. Control Plane Scalability in SDN

- Solution proposals examples
- HyperFlow (cont'd)
  - **High level view**

Event propagation system for cross-controller communication
Solution:
publish/subscribe system



Outline

- Arch
  - Scalability
  - Programming
- App
  - Security

Frenetic: A Programming Language for Software Defined Networks

Jennifer Rexford

Princeton University

http://www.frenetic-lang.org/

Joint work with Nate Foster, Dave Walker, Rob Harrison, Michael Freedman, Chris Monsanto, Mark Reitblatt, and Alec Story

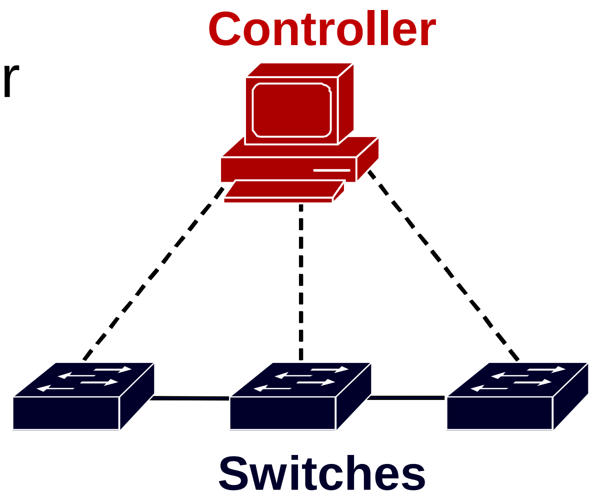
Network Programming is Hard

- Programming network equipment is hard
  - Complex software by equipment vendors
  - Complex configuration by network administrators
- Expensive and error prone
  - Network outages and security vulnerabilities
  - Slow introduction of new features

- SDN gives us a chance to get this right!
  - Rethink abstractions for network programming

Programming Software Defined Networks

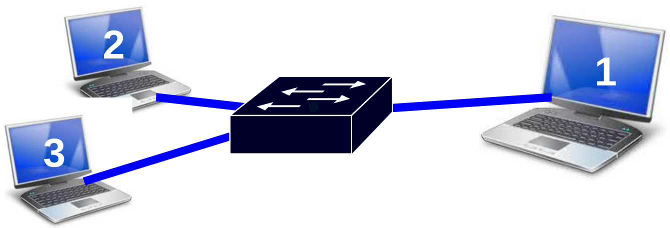
- OpenFlow already helps a lot
  - Network-wide view at controller
  - Direct control over data plane
- The APIs do not make it easy
  - Limited controller visibility
  - No support for composition
  - Asynchronous events



- Frenetic simplifies the programmer's life
  - A language that raises the level of abstraction
  - A run-time system that handles the gory details

Limited Controller Visibility

- Example: MAC-learning switch
  - Learn about new source MAC addresses
  - Forward to known destination MAC addresses
- Controller program is more complex than it seems
  - Cannot install destination-based forwarding rules
  - … without keeping controller from learning new sources



1 sends to 2 → learn 1, install
3 sends to 1 → never learn 3
1 sends to 3 → always floods

- Solution: rules on <inport, src MAC, dst MAC>

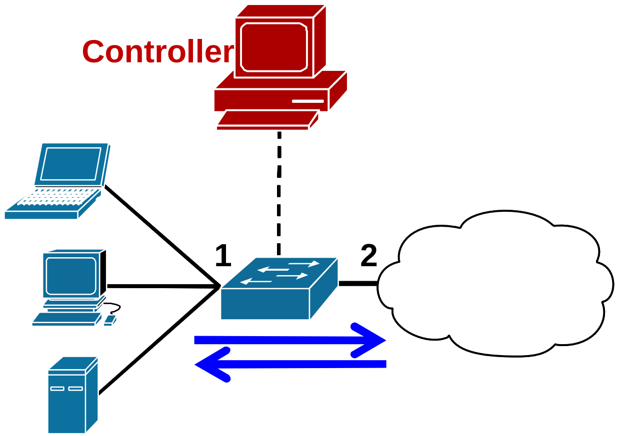
Must think about *reading* and *writing* at the same time.

Composition: Simple Repeater

Simple Repeater

```
def switch_join(switch):
  # Repeat Port 1 to Port 2
  p1 = {in_port:1}
  a1 = [forward(2)]
  install(switch, p1, DEFAULT, a1)

  # Repeat Port 2 to Port 1
  p2 = {in_port:2}
  a2 = [forward(1)]
  install(switch, p2, DEFAULT, a2)
```



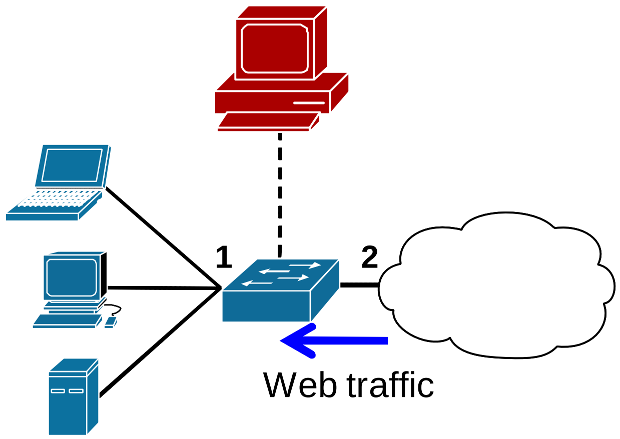
When a switch joins the network, install two forwarding rules.

Composition: Web Traffic Monitor

Monitor "port 80" traffic

```
def switch_join(switch)):
  # Web traffic from Internet
  p = {inport:2,tp_src:80}
  install(switch, p, DEFAULT, [])
  query_stats(switch, p)

def stats_in(switch, p, bytes, …)
  print bytes
  sleep(30)
  query_stats(switch, p)
```



When a switch joins the network, install one monitoring rule.

Composition: Repeater + Monitor

Repeater + Monitor

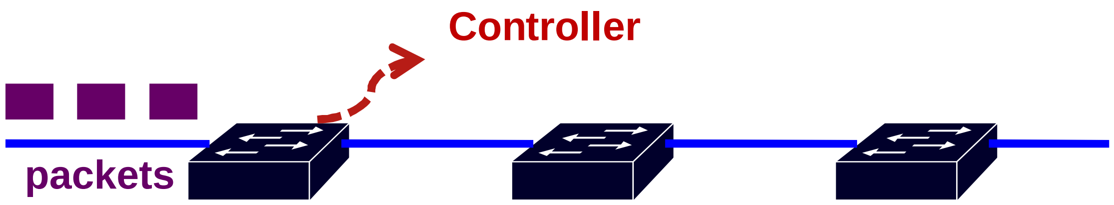
```
def switch_join(switch):
  pat1 = {inport:1}
  pat2 = {inport:2}
  pat2web = {in_port:2, tp_src:80}
  install(switch, pat1, DEFAULT, None, [forward(2)])
  install(switch, pat2web, HIGH, None, [forward(1)])
  install(switch, pat2, DEFAULT, None, [forward(1)])
  query_stats(switch, pat2web)

def stats_in(switch, xid, pattern, packets, bytes):
  print bytes
  sleep(30)
  query_stats(switch, pattern)
```

Must think about both tasks at the same time.

Asynchrony: Switch-Controller Delays

- Common OpenFlow programming idiom
  - First packet of a flow goes to the controller
  - Controller installs rules to handle remaining packets



- What if more packets arrive before rules installed?
  - Multiple packets of a flow reach the controller
- What if rules along a path installed out of order?
  - Packets reach intermediate switch before rules do

Must think about all possible event orderings.

Wouldn't It Be Nice if You Could...

- Separate reading from writing
  - Reading: specify queries on network state
  - Writing: specify forwarding policies
- Compose multiple tasks
  - Write each task once, and combine with others
- Prevent race conditions
  - Automatically apply forwarding policy to extra packets

This is what Frenetic does!

Our Solution: Frenetic Language

- Reads: query network state
  - Queries can see any packets
  - Queries do not affect forwarding
  - Language designed to keep packets in data plane
- Writes: specify a forwarding policy
  - Policy separate from mechanism for installing rules
  - Streams of packets, topology changes, statistics, etc.
  - Library to transform, split, merge, and filter streams
- Current implementation
  - A collection of Python libraries on top of NOX

Example: Repeater + Monitor

```
# Static repeating between ports 1 and 2
def repeater():
 rules=[Rule(inport:1, [forward(2)]),
        Rule(inport:2, [forward(1)])]
 register(rules)
```

Repeater

```
# Monitoring Web traffic
def web_monitor():
 q = (Select(bytes) *
      Where(inport:2 & tp_src:80) *
      Every(30))
 q >> Print()
```

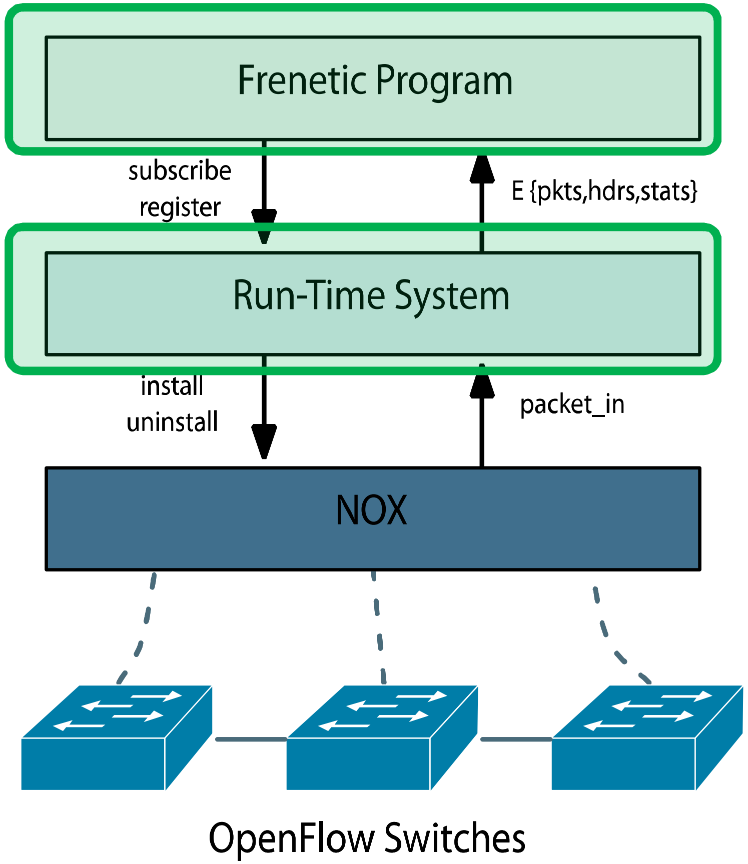
Monitor

Repeater + Monitor

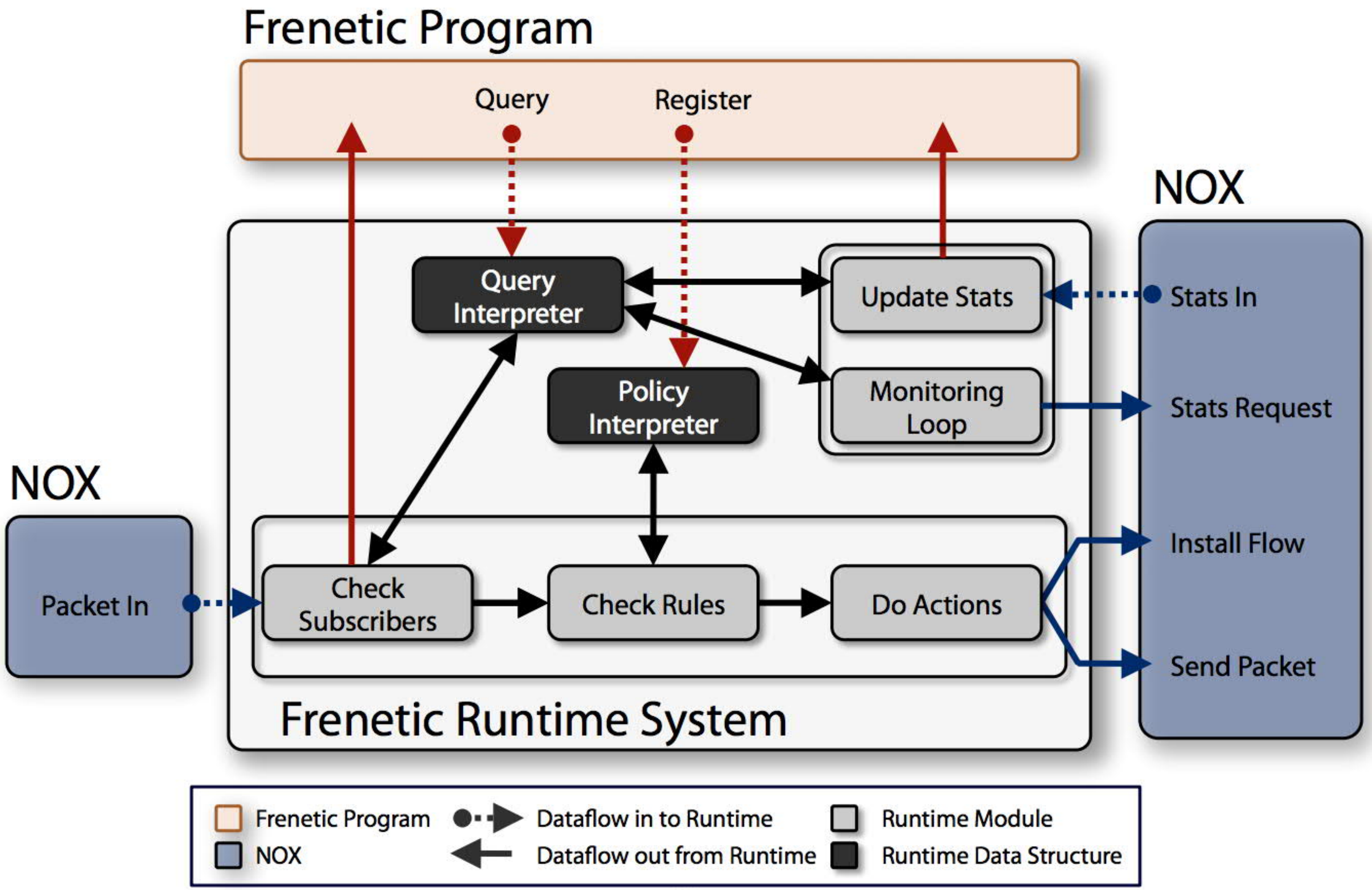
```
# Composition of two separate modules
def main():
 repeater()
 web_monitor()
```

Frenetic System Overview

- High-level language
  - Query language
  - Composition of forwarding policies
- Run-time system
  - Interprets queries and policies
  - Installs rules and tracks statistics
  - Handles asynchronous events



Run-time Activities



Outline

- Arch
  - Scalability
  - Programming
- App
  - Security

Exemplar SDN Security Apps

- Security functions can be applications of SDN
  - Firewall
  - DDoS detection
  - Scan detection
  - Reflector net
  - Tarpit
  - Dynamic quarantine
  - and more…

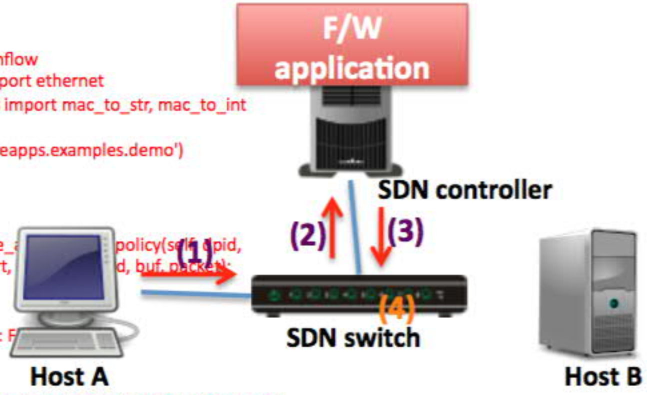
```
import logging

from nox.lib.core import *
import nox.lib.openflow as openflow
from nox.lib.packet.ethernet import ethernet
from nox.lib.packet.packet_utils import mac_to_str, mac_to_int

log = logging.getLogger('nox.coreapps.examples.demo')

class demo(Component):
def __init__(self, ctxt):def create_        policy(self, dpid,
policy_type, outport_find, inport,         d, buf, packet):

if outport_find == 0:
print 'DBG: No Specific Out Port: F
if policy_type == 'ARP':
print 'DBG: ARP packet'
self.send_openflow(dpid, bufid, buf, openflow.OFPP_FLOOD, inport)
elif policy_type == 'REQ':
attrs = extract_flow(packet)
attrs = {core.IN_PORT:inport,
core.DL_TYPE:ethernet.IP_TYPE,
core.NW_PROTO:ipv4.ipv4.TCP_PROTOCOL
core.NW_SRC:'10.0.0.2'}
```

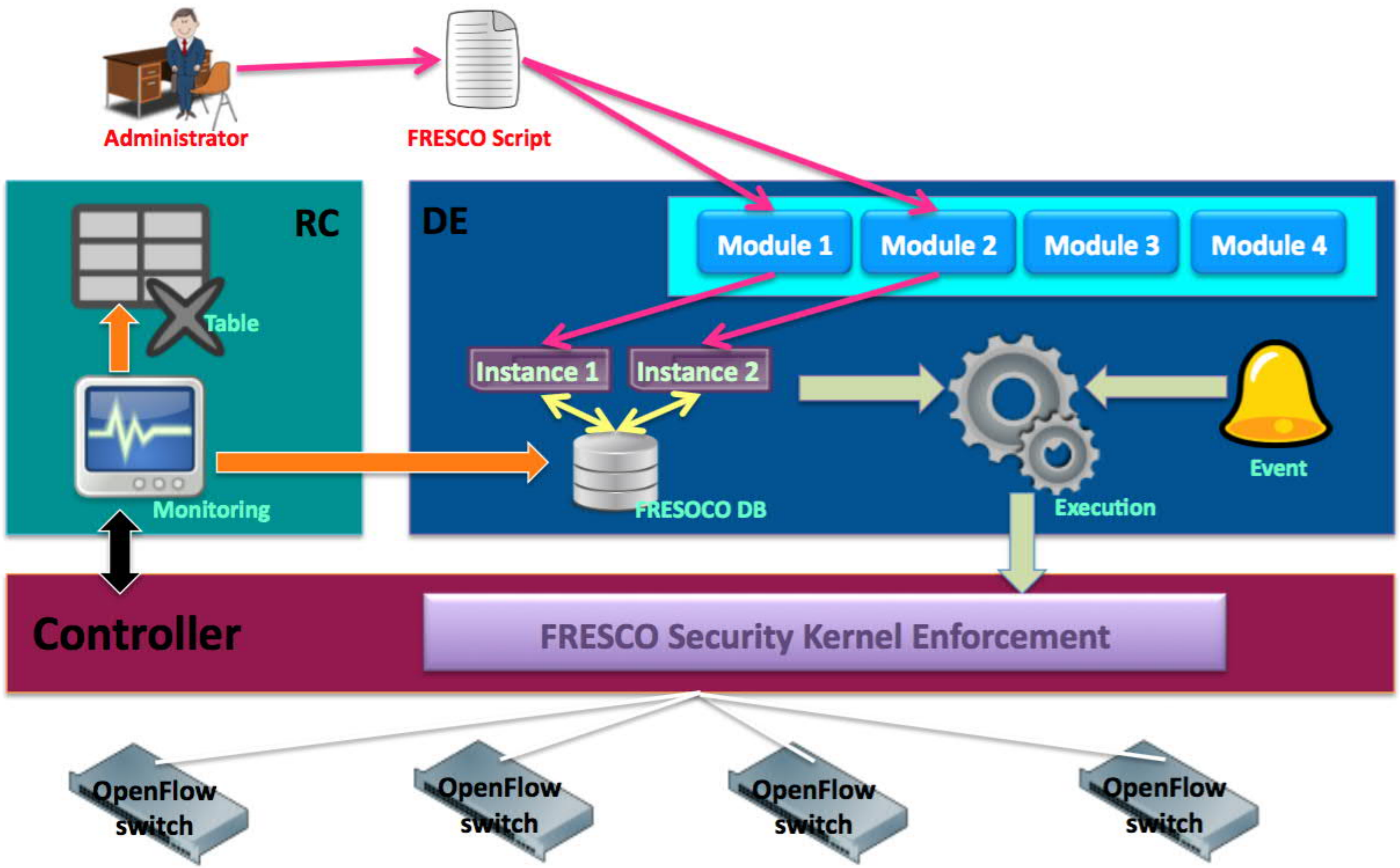


(1) Host A sends packet to Host B
(2) Switch asks a controller form a flow rule
(3) F/W application decides to block the packet
(4) Switch drops this packet

FRESCO

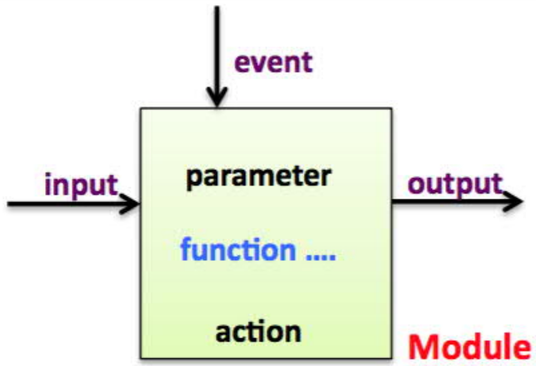
- FRESCO is a new frame work that
  - Provides a new development environment for security applications
  - Effectively manages shared resources among security applications
  - Simplifies deployment of security policies
    - provides a set of 7 new intelligent security action primitives
      - E.g., block, deny, allow, redirect, and quarantine

Operational Scenario

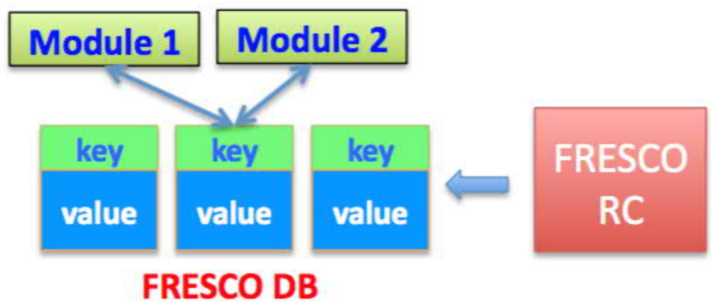


Development Environment

- FRESCO Module
  - Basic operation unit



- FRESCO DB
  - Simple database
    - (key,value) pairs



- FRESCO script
  - Define interfaces
  - Connect multiple modules

Development Environment

- FRESCO script
  - Format
    - **Instance name (# of input) (# of output)**
    - **type**: class of this module
    - **input**: input for this module
    - **output**: output of this module
    - **parameter**: define some variables
    - **event**: trigger a module
    - **action**: conduct this action

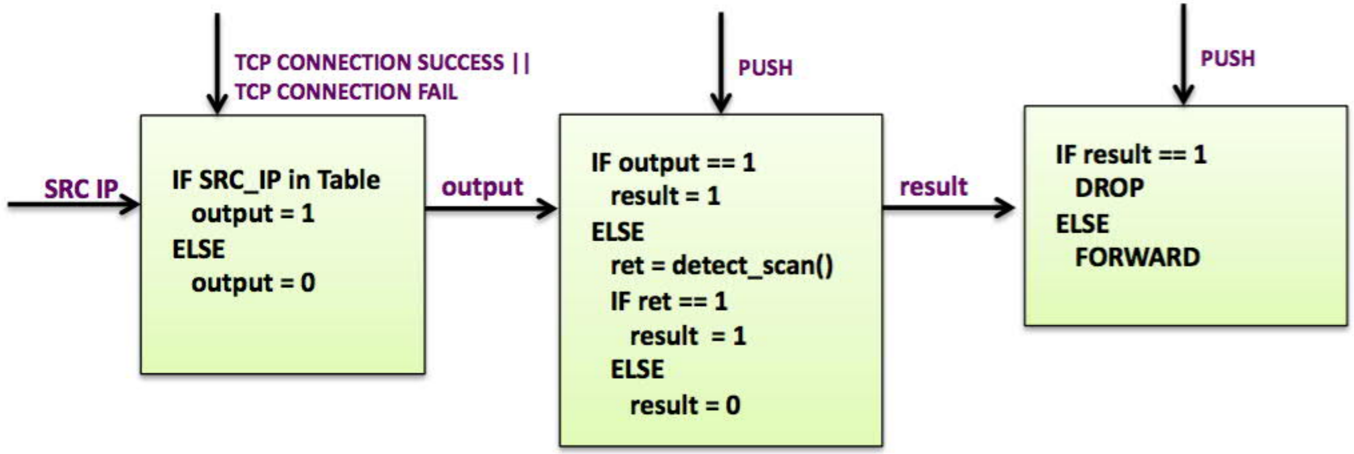
```
port_comparator (1)(1) {
   type: Comparator
   event: PUSH
   input: destination_port
   output: comparison_result
   parameter: 80
   action: -
}
```

Inspired by Click Modular Router

Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. Proceedings of SOSP '99

Example: Scan Detection

- Steps
  - Check blacklist → Threshold based scan detection → Drop or Forward



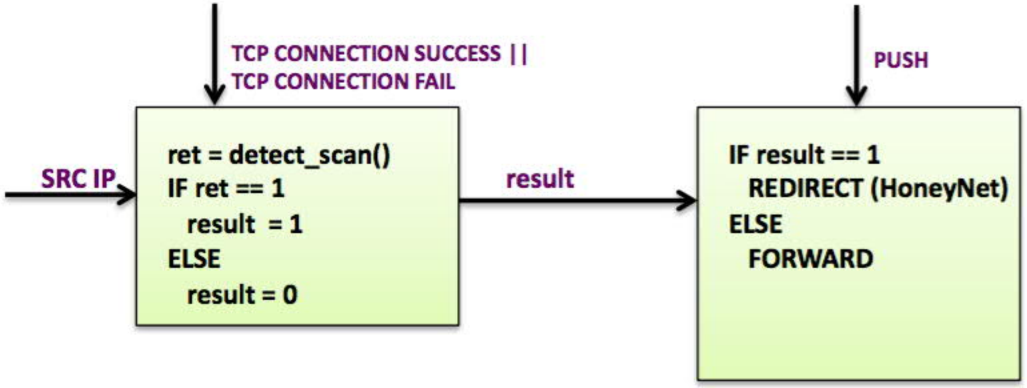

```
blacklist_check (1)(1) {
  type: TableLookup
  event: TCP_CONNECTION_FAIL,
        TCP_CONNECTION_SUCCESS
  input: SRC_IP
  output: blacklist_out
  parameter: NONE
  action: NONE
}
```

```
scan_detect (1)(1) {
  type: ScanDetect
  event: PUSH
  input: blacklist_out
  output: detect_result
  parameter: NONE
  action: NONE
}
```

```
final_action (1)(0) {
  type: ActionHandler
  event: PUSH
  input: detect_result
  output: NONE
  parameter: NONE
  action: detect_result == 1
          ?DROP:FORWARD
}
```

Example: Reflector Net

- Confuse network scan attackers
- Steps
  - Threshold based scan detection → Reflect or Forward




```
scan_detect (1)(1) {
   type: ScanDetect
  event: TCP_CONNECTION_FAIL,
            TCP_CONNECTION_SUCCESS
input: blacklist_out
   output: detect_result
   parameter: NONE
   action: NONE
}
```

```
final_action (1)(0) {
   type: ActionHandler
   event: PUSH
   input: detect_result
   output: NONE
   parameter: NONE
   action: detect_result ==  ?REDIRECT(10.0.0.3):FORWARD
}
```

Example: Reflector Net

- Test result