NOX, POX, and lessons learned

James “Murphy” McCauley

Organization

- A Bit of History
- Lessons Learned
  - Part 1: Two little lessons
  - Part 2: Thinking big
- Ongoing Work
- Wrap-Up

Current NOX and POX Collaborators

- Murphy McCauley *(ICSI, UC Berkeley)*
- Aurojit Panda *(UC Berkeley)*
- Colin Scott *(UC Berkeley)*
- Amin Tootoonchian *(ICSI, U of Toronto)*
- Andreas Wundsam *(ICSI)*
- Kyriakos Zarifis *(ICSI)*

- Anyone with a github account

NOX: A Bit of History

- NOX was the first SDN controller
- Developed at Nicira at the same time OpenFlow was being developed
  - Highly synergistic relationship
- Released under GPL in 2008
  - Extensively used in research
- Now maintained by research community

NOX Highlights

- Linux
- C++ and Python
- Cooperative multithreading
- Component system
- Event-based programming model
- OpenFlow interface
- Packet construction/dissection libraries
- Applications:
  - Forwarding (reactive), topology discovery, host tracking, …

Lessons Learned

- Part 1: Two small lessons
  - Deployability matters
  - Language choice matters
- Part 2: One bigger lesson
  - Thinking big

Deployability matters
Language choice matters

Lessons Learned Part 1

Observation 1: Deployability

NOX is difficult to deploy

- A fairly large number of users have trouble building and running NOX in their environment
- Relatively complex build with a fair number of dependencies

- New users are mostly researchers
- Experienced users are mostly researchers

Observation 2: C++ and Python

NOX was programmable in C++ and Python

- *Expectation:* Python would be "glue" for more substantial C++ components
- *Actuality:* Significant applications entirely in Python
  - We think more than in C++
  - Very few really used C++ *and* Python
- *Results:*
  - Python API wasn't as good as one might hope for doing full applications
  - Python support added a fair amount of maintenance and build complexity – unnecessary for those just using C++

What We Learned

- Deployability matters to us
  - We need to pick our dependencies very carefully
- Pick a language
  - Integrating two languages takes effort
  - .. and nobody cares anyway

Applying What We Learned

- Remove Python from NOX: “New” NOX
  - Immediate simplification of NOX code and deployment (less code; fewer dependencies)
  - Change of threading model possible
  - Makes NOX a better platform for those who want to use C++

Applying What We Learned

- Build a new platform in pure Python: POX
  - Pick our dependencies very carefully
  - Take things we liked from NOX
  - Target Linux, Mac OS, and Windows
  - Use this as the basis for as much of our own research going forward as possible

- Goal: Good for research
- Non-goal: Performance

A Sidenote on Performance

- We don't have great SDN benchmarks yet
  - Ones we have focus on purely reactive
  - Many controllers outpace many hardware OpenFlow switches
- If performance across the board matters to you:
  - Research controller probably isn't a good fit

POX

Choosing our dependencies:

1. Python 2.7
   - Expected to have a long life
   - System Python on Ubuntu and Mac OS
     - Probably will be for a while
   - Lots of nice new stuff
   - Supported by PyPy
     - Alternative Python runtime
     - Great performance
     - Easy: download, decompress, run POX with it

3. There is no #2! No other dependencies.

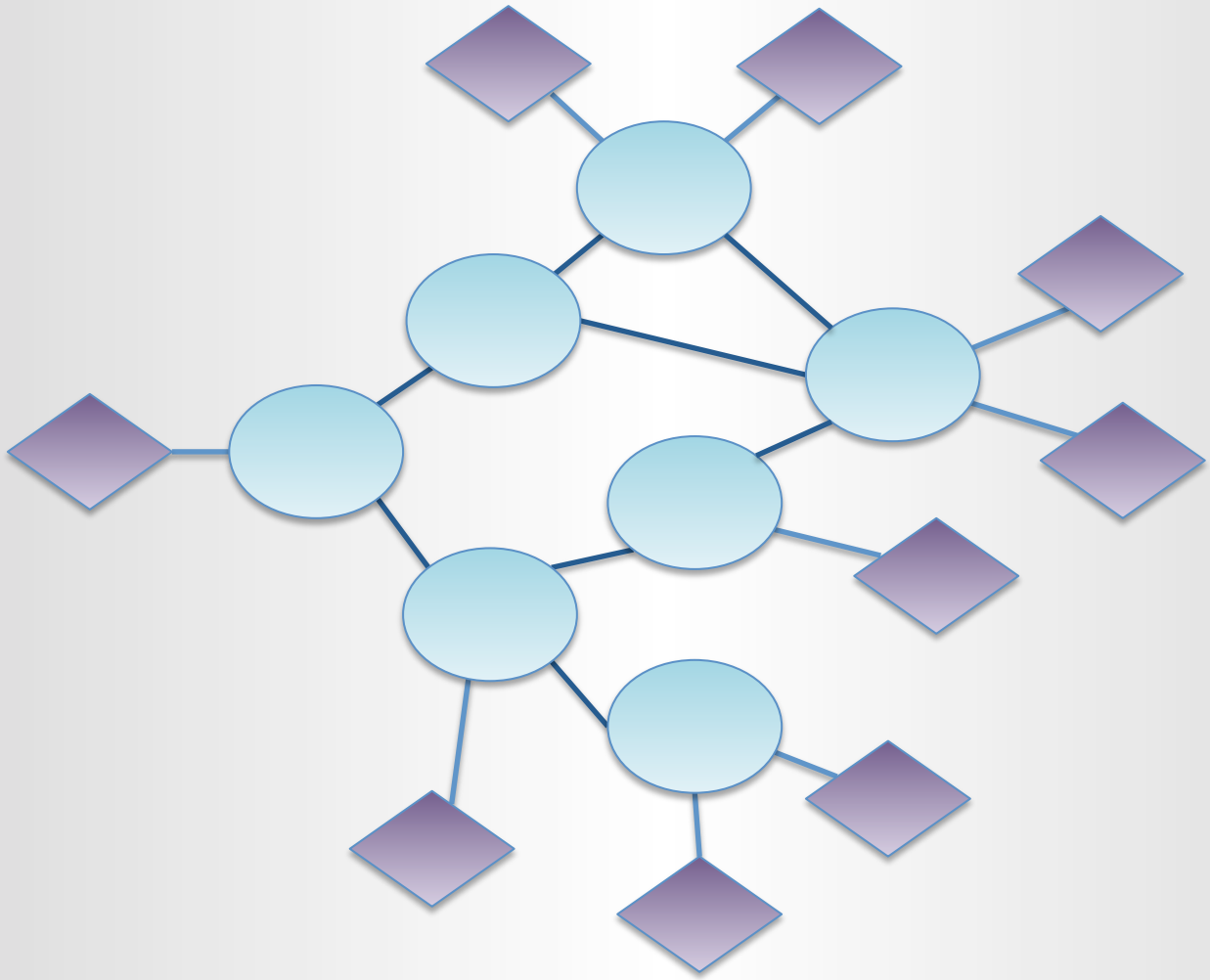
POX

- Borrowed ideas from NOX:
  - Cooperative multitasking
  - Component system
  - OpenFlow interface (much improved)
  - Messenger
- Borrowed code from NOX:
  - Packet construction/dissection
  - GUI

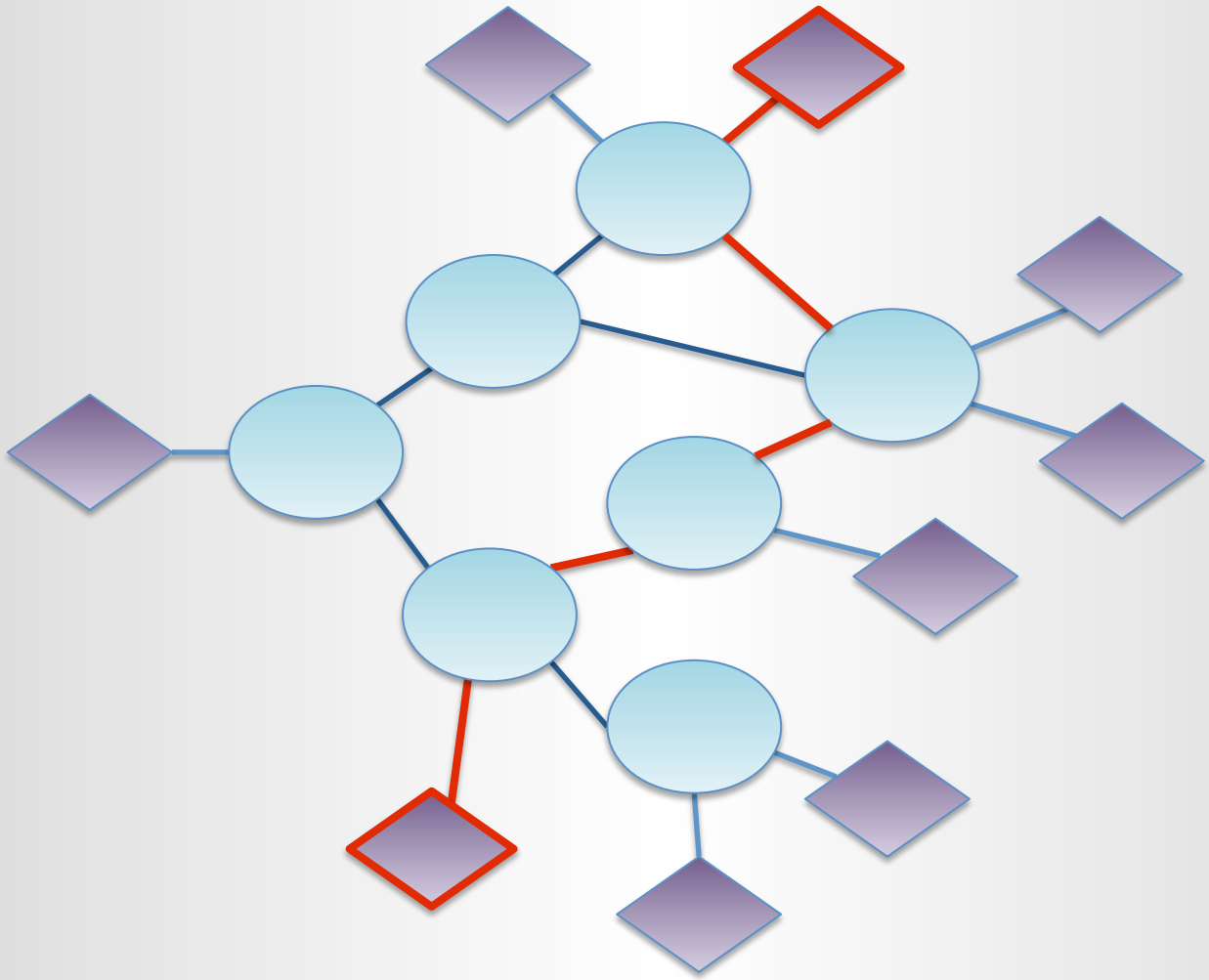
Thinking Big

Lessons Learned Part 2

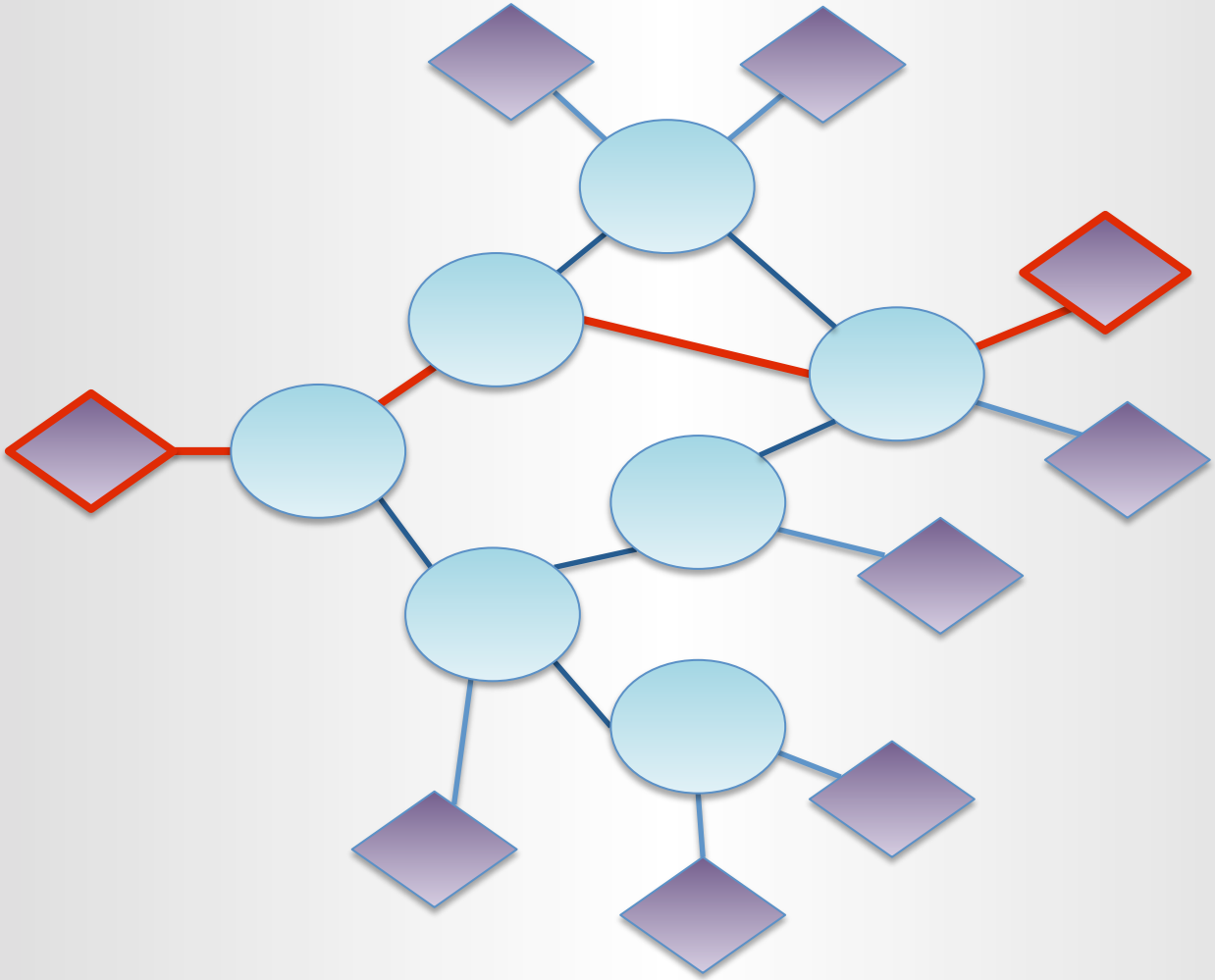
A Simple Example



A Simple Example

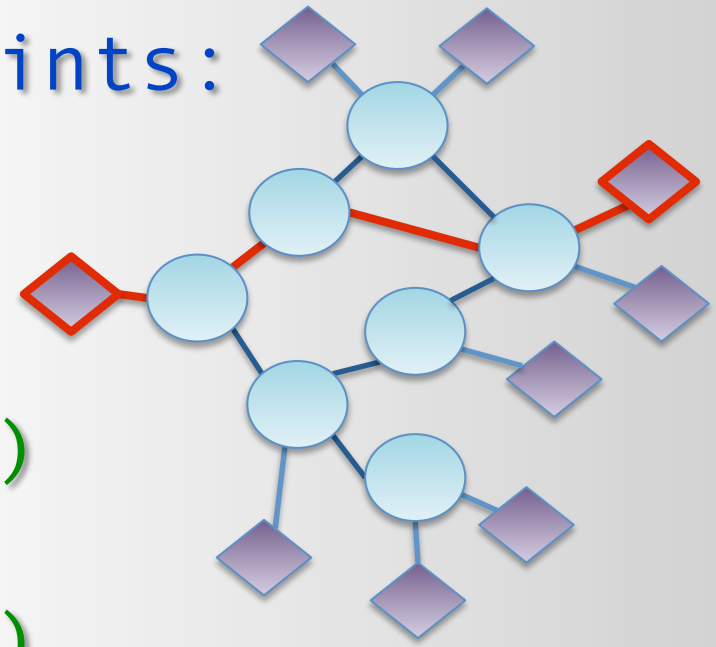


A Simple Example



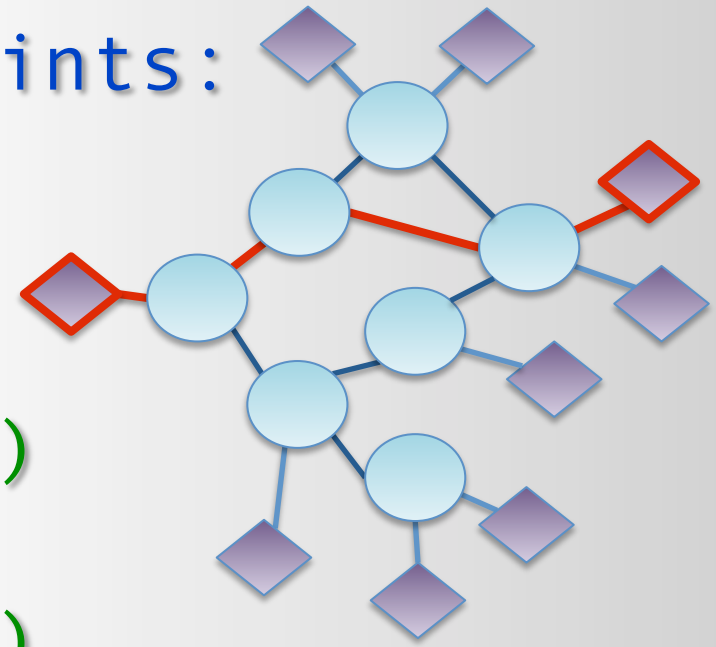
A Simple Example

```
func initialize ():
  clear_all_switches()
  for each A,B in endpoints:
    new_path(A, B)

func new_path (A, B):
  path = best_path(A, B)
  path.install()
  all_paths.append(path)
```



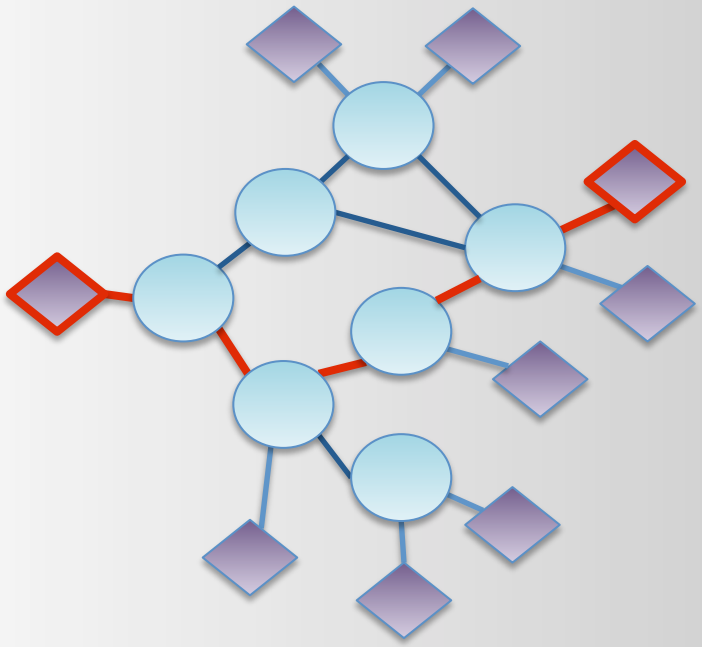
A Simple Example

```
func initialize ():
  clear_all_switches()
  for each A,B in endpoints:
    new_path(A, B)

func new_path (A, B):
  path = best_path(A, B)
  path.install()
  all_paths.append(path)
```



A Simple Example

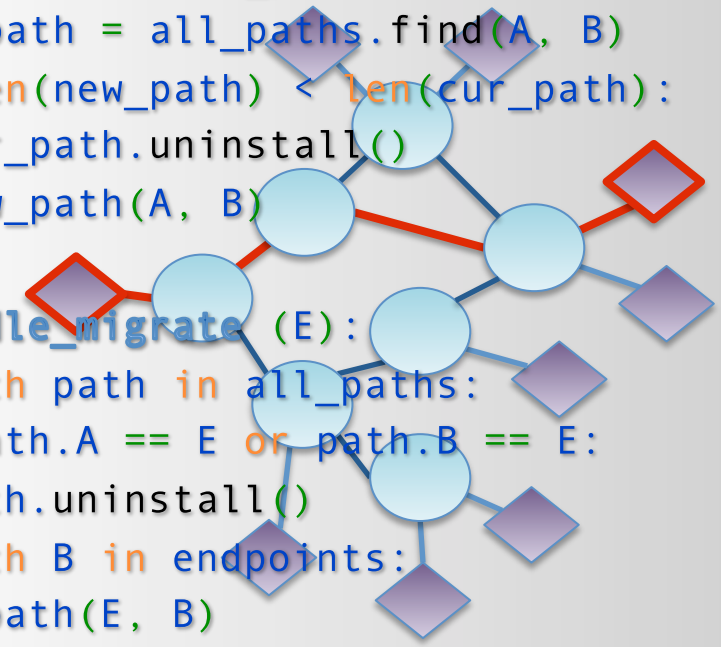
```
func initialize ():
  clear_all_switches()
  for each A,B in endpoints:
    new_path(A, B)

func new_path (A, B):
  path = best_path(A, B)
  path.install()
  all_paths.append(path)

func handle_switch_down (switch):
  lost = new List()
  for each path in all_paths:
    if switch in path:
      all_paths.remove(path)
      lost.append(path.endpoints)
      path.uninstall()
  for each A,B in lost:
    new_path(A, B)
```



A Simple Example

```
func initialize ():
  clear_all_switches()
  for each A,B in endpoints:
    new_path(A, B)

func new_path (A, B):
  path = best_path(A, B)
  path.install()
  all_paths.append(path)

func handle_switch_down (switch):
  lost = new List()
  for each path in all_paths:
    if switch in path:
      all_paths.remove(path)
      lost.append(path.endpoints)
      path.uninstall()
  for each A,B in lost:
    new_path(A, B)
```

```
func handle_switch_up (switch):
  for each A,B in endpoints:
    new_path = best_path(A, B)
    cur_path = all_paths.find(A, B)
    if len(new_path) < len(cur_path):
      cur_path.uninstall()
      new_path(A, B)

func handle_migrate (E):
  for each path in all_paths:
    if path.A == E or path.B == E:
      path.uninstall()
  for each B in endpoints:
    new_path(E, B)
```



A Simple Example

- Still need to handle:
  - Link up
  - Link down
  - Adding endpoints
  - Removing endpoints

What have we been doing?

```
func event_handler (some_event):
  send_commands_to_switches()
```

$$f : \Delta state \rightarrow \Delta config$$

What have we been doing?

$$f : \Delta state \rightarrow \Delta config$$

- It's a natural way to write control logic
- OpenFlow protocol is largely deltas:
  - Switch-to-Controller: changes of network state
  - Controller-to-Switch: changes of configuration
- Most example SDN code works like this
- **IT'S HARD TO GET THIS RIGHT**

Issues

- Some state is actually stored on the switches
  - Distributed systems problem
  - Not entirely reliable connections to this state
  - Easy to accidentally assume ordering which does not actually exist (e.g., due to differing latencies)
  - *Errors are cumulative*
- Some of the state is held on the controller(s)
  - Some by the platform (topology info in example)
  - Some by the application (paths in example)
- You're juggling three kinds of state and they have very different properties

Issues

- The code is fairly complex
  - The example code had three event handlers with three different algorithms to respond!
  - Every event type → another algorithm ?

Alternative: Think Big

$$f: \text{link down} \rightarrow \Delta\text{config}$$

$$f: \text{switch up} \rightarrow \Delta\text{config}$$

$$\cdots$$

$$f: \Delta\text{state} \rightarrow \Delta\text{config}$$

$$f: \text{state} \rightarrow \text{config}$$

Alternative: Think Big

$$f : state \rightarrow config$$

- Said another way:
  - Always recalculate the complete configuration based on the complete state

- Falls out of "Shenker [Casado,Koponen,...] view"

Alternative: Think Big

```
func update_state ():
  for each A,B in endpoints:
    path = best_path(A, B)
    path.install()
```

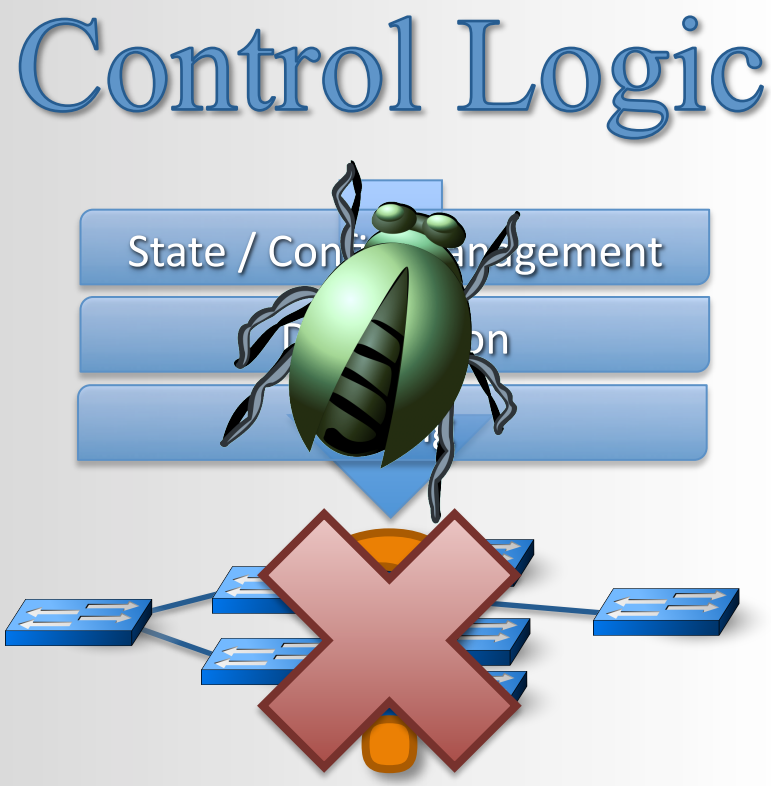
That's it!

Implications

- Requires *way* less control logic
  - A single (deterministic!) algorithm
  - No cumulative errors
- Platform gets more complex
  - Must build local model of state from deltas
  - Must build deltas from local configuration
  - But platform is cleanly separated from control logic
    - Reusable
    - Less complex than weaving this all together!
- Easier to reason about control logic
  - A single input – not a sequence of events
- Downside: More computation

Ongoing Work

Ongoing work: Troubleshooting



- Control Logic determines configuration of network
- Intermediate platform functionality makes it harder to reason about final configuration
- Platform itself may contain bugs!

Ongoing work: Troubleshooting

- SDN is all about software, so…
- **You need a debugger!**

- Approach based on correspondence checking

A Quick Example

A super-simple POX learning switch

Quick Example: Overview

1. `git clone http://noxrepo.org/git/pox`
2. `cd pox`
3. `vim ext/switch.py # Write a learning switch`
4. `./pox.py switch`

Quick Example: ext/switch.py

```
from pox.core import core
from pox.openflow.libopenflow_01 import *

def handle_PacketIn (event):
  msg = ofp_flow_mod()
  msg.match.dl_dst = event.parsed.src
  msg.actions.append(ofp_action_output(port = event.port))
  event.connection.send(msg)

  msg = ofp_packet_out()
  msg.actions.append(ofp_action_output(port = OFPP_FLOOD))
  msg.buffer_id = event.ofp.buffer_id
  msg.in_port = event.port
  event.connection.send(msg)

def launch ():
  core.openflow.addListenerByName("PacketIn", handle_PacketIn)
```

Wrap-Up

- NOX Classic
  - Still available. C++ and Python.
- NOX (New fork)
  - Available *this week!* C++ only. Cleaner all over.
- POX
  - Work in progress; Available now. Python only.
  - More stuff becoming available.
- SDN Debugger
  - Work in progress; Available now.
  - Framework for finding bugs across control plane layers.

- Find it all starting from http://noxrepo.org

Thanks for listening!