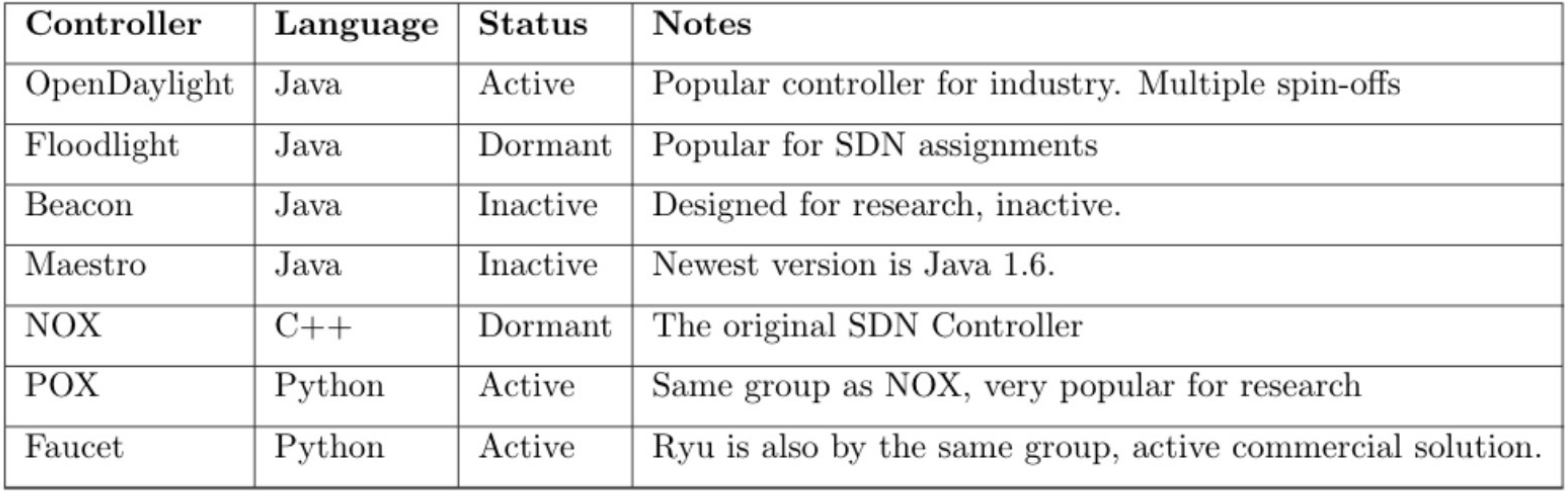
SDN Tutorial

Based on Chen Liang's floodlight tutorial @ Duke University

Colorado State University
Department of Computer Science

SDN Controllers



Controller	Language	Status	Notes
OpenDaylight	Java	Active	Popular controller for industry. Multiple spin-offs
Floodlight	Java	Dormant	Popular for SDN assignments
Beacon	Java	Inactive	Designed for research, inactive.
Maestro	Java	Inactive	Newest version is Java 1.6.
NOX	C++	Dormant	The original SDN Controller
POX	Python	Active	Same group as NOX, very popular for research
Faucet	Python	Active	Ryu is also by the same group, active commercial solution.

Table 1: Side by Side Comparison of Popular SDN Controllers

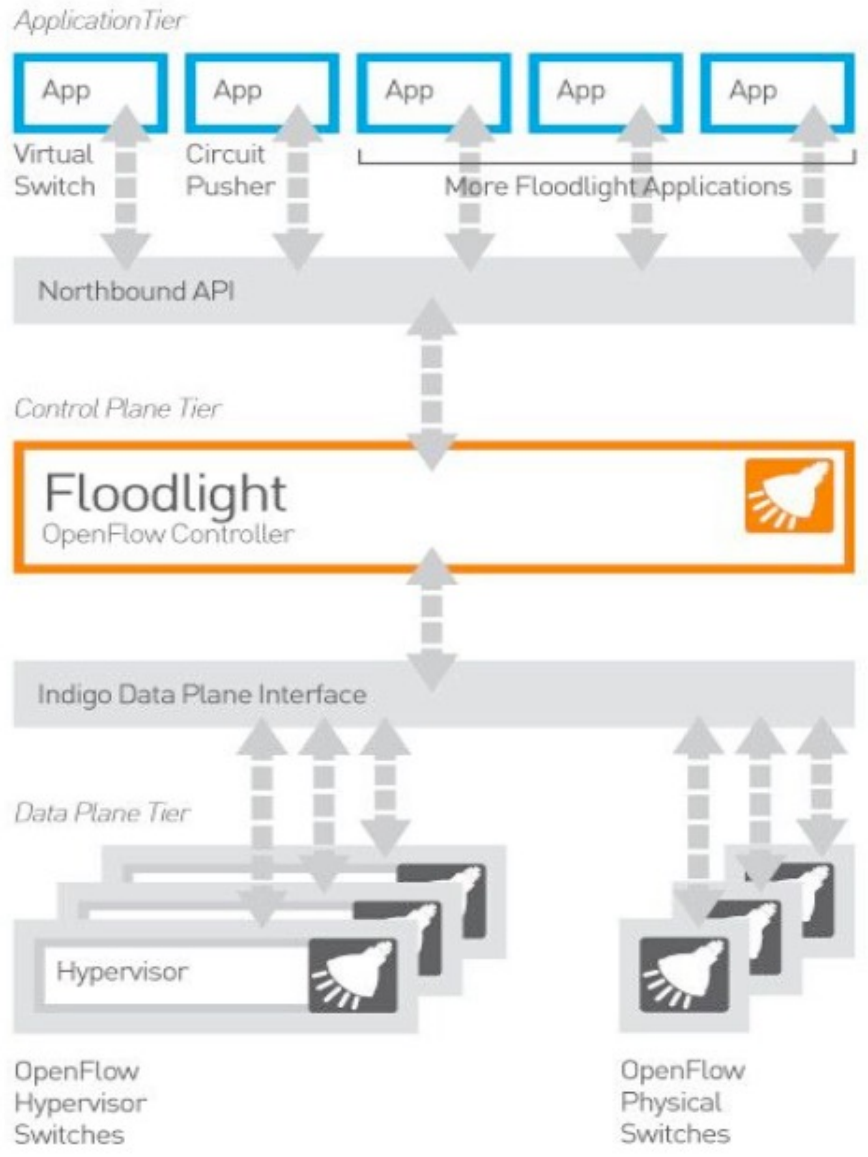
Assignment Controllers

- The CS 557 Assignment may be done in Python or Java
  - For consistency, it was narrowed down to using Floodlight or POX
- Why Floodlight?
  - OpenDaylight while more popular comes with a lot of industrial “baggage”
    - Really into writing SDN controllers? get involved in the open-source development!
  - Floodlight is using the Java 1.6 version due to capability issues with recent OpenJDK releases and Openflow13
    - Make sure to use the provided virtual machine! Don’t update it.
- Why POX?
  - Comes with mininet!
  - Heavy focus in academia and research
  - Also has industrial uses

Details: Floodlight

- All Controllers help determine
  - which port a switch should send traffic
  - based on packet information
    - destination IP, MAC, packet contents, etc
- Floodlight
  - Java based
  - Needs updating, but works for this assignment
    - Very specific in helping understand the assignment goals
  - Step 1: Set up a Match
    - what type of packet is this?
  - Step 2: Install an Action on a *switch*
    - specifies which port on the switch to send packet
    - if match is true, do action

Floodlight overview



Step 1: Setup a Match

- Example matches:
  - <src ip: 10.0.0.2, dst ip 10.0.0.3, src port: 90>
    <src mac addr: 00:0a:95:9d:68:16>
    <vlan tag: 4000, protocol: ipv4>
- In Floodlight, each match is an object of org.openflow.protocol.OFMatch
  - The match can match anything (both destination and source identifiers) that is
    - port
    - MAC Address
    - IP
    - VLAN
    - ethernet
  - Be careful!
    - A lot of flexibility
    - But enough to cause unexpected issues (IP and MAC address matching that conflict when IP changes)

Step 1: Match Code

```
OFMatch addMatch(Host host) {
    OFMatch ofMatch = new OFMatch();
    OFMatchField etherType = new
           OFMatchField(OFOXMFieldType.ETH_TYPE,
           Ethernet.TYPE_IPv4);
    OFMatchField macAddr = new
           OFMatchField(OFOXMFieldType.ETH_DST,
           Ethernet.toByteArray(host.getMACAddress()));
    ofMatch.setMatchFields(
           Arrays.asList(etherType, macAddr));
    return ofMatch;
}
```

- For this assignment:
  - MAC Address is critical
  - Host is found in project **util** folder

- Algorithm:
  - Create Match object
  - setup one type of match field (IPv4)
  - setup another type of match field (mac address)
  - Add fields to the Match object
  - returns Match object for later use

Step 2: Build the Action

- Actions say “for the current switch, use port X” based on your destination
- In floodlight
  - setup an OFActionOutput object
  - Set the port for that OFActionOutput
- Notice, we are not saying anything about the destination, just the port to use on the action
- We then install the command on the switch, by associating the Match with the Action
- Essentially
  - Build Match
  - Build Action
  - Link them by installing command on the switch

Step 2: Simplified Code

```
OFActionOutput getActionOutput(Host host, IOFSwitch currentSwitch) {
    IOFSwitch hostSwitch = host.getSwitch();
    OFActionOutput ofActionOutput = new OFActionOutput();
    if (currentSwitch.getId() == hostSwitch.getId()) {
       ofActionOutput.setPort(host.getPort());
    } else {
      // do something fancy, with shortest path
      // knowing both the next switch to go to,
      //and then using the “links” will help
    }
    return ofActionOutput;
}
```

- Recall – host is the location you seek to *go to*.
- If the host’s switch and current switch are the same
  - you just need to know which port the host is on

Step 2: Installing the action continued

```
void installCommand(Host host, IOFSwitch currentSwitch, OFMatch match, byte table) {
    OFInstructionApplyActions actions = new
              OFInstructionApplyActions(Collections.<OFAction>singletonList(
                                        getActionOutput(host, currentSwitch)));

        SwitchCommands.installRule(currentSwitch, table,
                                  SwitchCommands.DEFAULT_PRIORITY, match,
                                  Collections.<OFInstruction>singletonList(actions));
}
```

- The above code says:
  - first get the action (port) to use, based on the target Host (**host**) and the switch you want to write the rule to
  - Then write the rule to the switch, using the passed in Match (also based on the host).

Student TODO

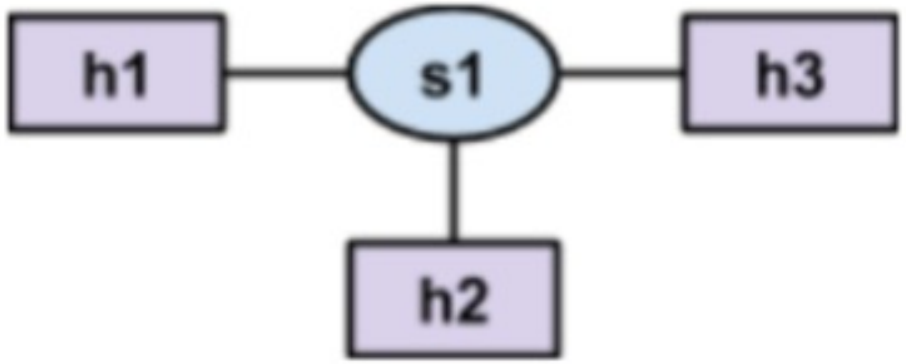
- First step: get the following topology working!
  - Only one switch
  - the Action is always just setting the port for the host.

- After this is working, you can focus on shortest path.

- Before moving on: Make sure to run

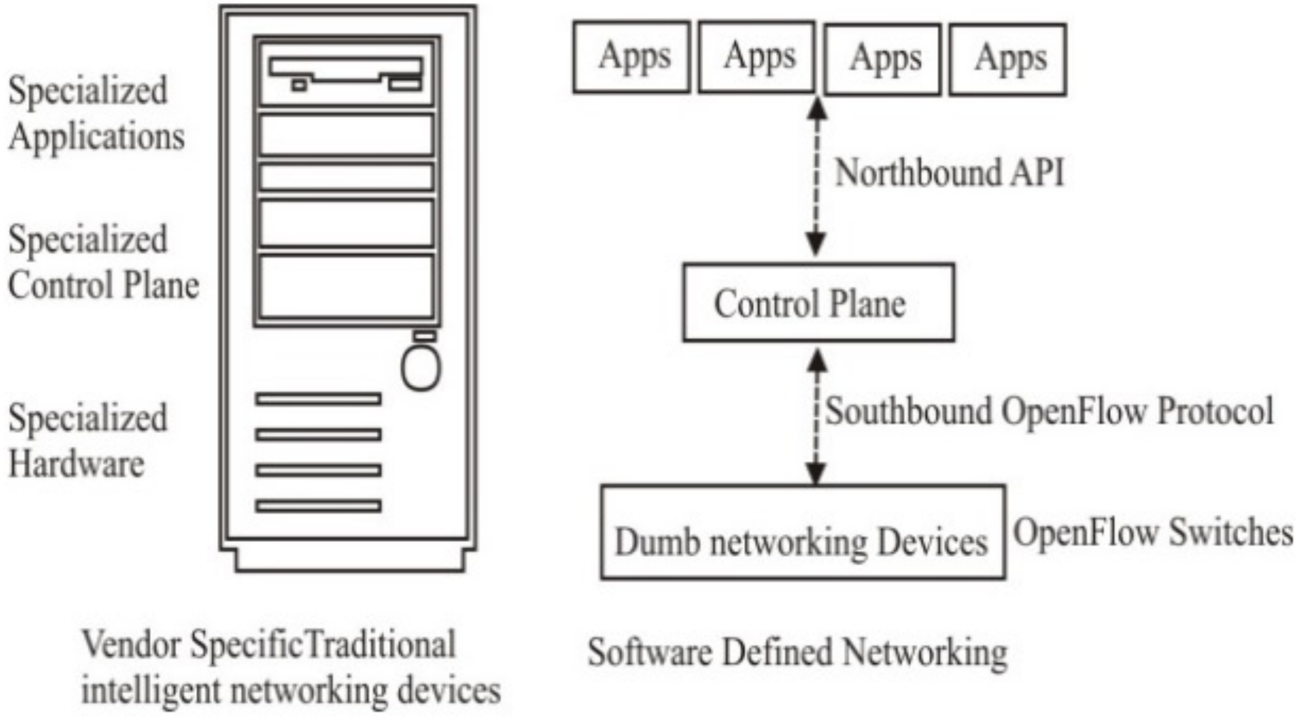
```
sudo ovs-ofctl -O OpenFlow13 dump-flows s1
```

  - This will help you see the actions on the switch
  - You should see three of them



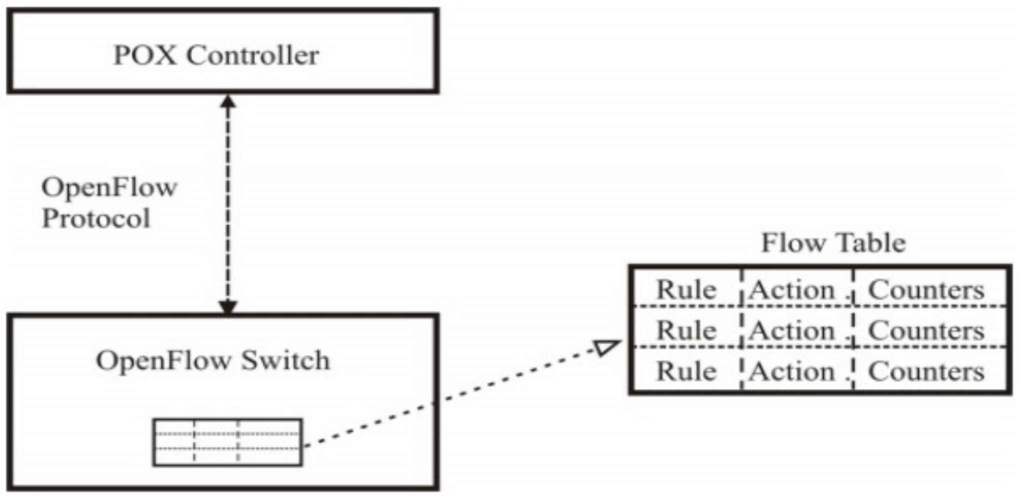
Details: POX

- All Controllers will determine
  - The flow for a switch to send packets through
  - Based on openflow matches, IPs, ports, etc.
- POX
  - Python based
  - Open Source, included with Mininet
    - Hides some details, useful for demonstrating SDN
  - Step 1: Set up a Flow
    - what does the switch need to know?
  - Step 2: Install a Flow on a *switch*
    - specifies which port on the switch to send packet
    - if match is true, do action



Vendor SpecificTraditional intelligent networking devices

Software Defined Networking



Rule	Action	Counters
Rule	Action	Counters
Rule	Action	Counters

Step 1: Flow Code

```
def install_flow (self, con_or_dpid, priority = None):
    if priority is None:
       priority = self._flow_priority
    if isinstance(con_or_dpid, int):
       con = core.openflow.connections.get(con_or_dpid)
    if con is None:
       log.warn("Can't install flow for %s", dpid_to_str(con_or_dpid))
       return False
    else:
       con = con_or_dpid
    match = of.ofp_match(dl_type = pkt.ethernet.LLDP_TYPE,
                         dl_dst = pkt.ETHERNET.NDP_MULTICAST)
    msg = of.ofp_flow_mod()
    msg.priority = priority
    msg.match = match
    msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
    con.send(msg)
    return True
```

- For this assignment:
  - Details tend to be hidden
  - openflow.py has all implentation details for flow, match
- Algorithm:
  - Assign priority, connection/datapath ID
  - Create a match using LLPD, multicast
  - Create a message for the switch
  - Add fields to the message object
  - Send the message

Step 2: Build the Action

- Flows say “for the current switch, use port X” based on your destination
- In POX
  - Set up match, message objects
  - Set the port for the action within the message
  - Notice, we are not saying anything about the destination, just the port to use on the action
- We then install the command on the switch, by sending the message via the connection
- Essentially
  - Build Match
  - Build Message
  - Link them by sending a message over the connection with the switch

Step 2: Entry Install Code

```
def install (self, entries=[]):
    self._mod(entries, OFSyncFlowTable.ADD)

def _mod (self, entries, command):
    if isinstance(entries, TableEntry):
        entries = [ entries ]

    for entry in entries:
        if(command == OFSyncFlowTable.REMOVE):
            self._pending = [(cmd,pentry) for cmd,pentry in self._pending
                if not (cmd == OFSyncFlowTable.ADD
                    and entry.matches_with_wildcards(pentry))]
        elif(command == OFSyncFlowTable.REMOVE_STRICT):
            self._pending = [(cmd,pentry) for cmd,pentry in self._pending
                if not (cmd == OFSyncFlowTable.ADD
                    and entry == pentry)]
        self._pending.append( (command, entry) )
    self._sync_pending()
```

- Entry objects are created to be put into the flow table, or commands are sent to delete them

Step 2: Resync the switch with Entries

- When the switch resyncs with the entries:
  - Build list of TODO operations (could be adding or removing flows)
    - Option to clear all entries or to perform specific ADD/DELETE ops
  - For operations in the TODO, send flow_mod objects to the switch and perform changes

```
for op in todo:
    fmod_xid = self.switch._xid_generator()
    flow_mod = op[1].to_flow_mod(xid=fmod_xid, command=op[0],
                                flags=op[1].flags | of.OFPFF_SEND_FLOW_REM)
    self.switch.send(flow_mod)
```

Student TODO

- First step: get the following topology working!
  - Only one switch
  - the Action is always just setting the port for the host.

- After this is working, you can focus on spanning tree

- Before moving on: Make sure to run

```
sudo ovs-ofctl -O OpenFlow13 dump-flows s1
```

  - This will help you see the actions on the switch
  - You should see three of them