NOSIX

A Lightweight Portability Layer for the SDN OS

ONS 2013 - Research Track

Andreas Wundsam (Big Switch Networks) · Minlan Yu (USC)·
Muruganantham Raju (USC)

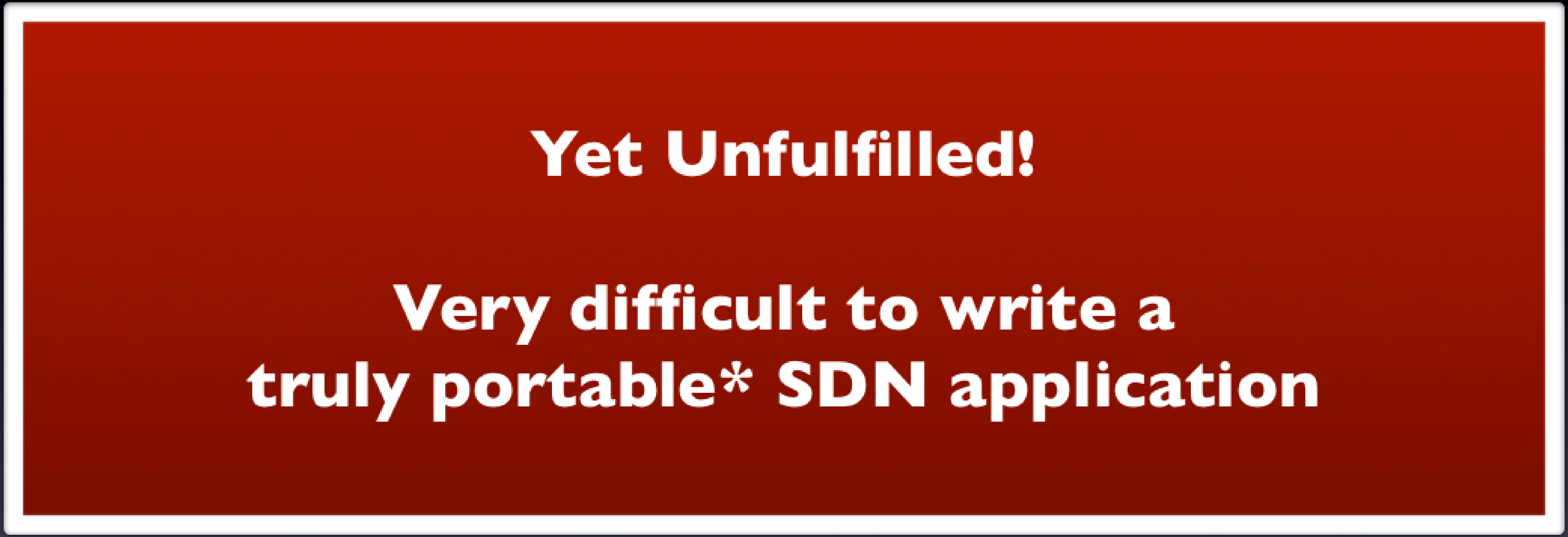
Motivation

Core SDN promise:

Freedom from Vendor Lock-In

mix + match switches
reuse your SDN application

Motivation



Yet Unfulfilled!

Very difficult to write a truly portable* SDN application

(*) **correct** and **efficient** forwarding
over a **wide range of switches**

Switch Diversity

Heterogenous Switch landscape!

- **Data Plane:**
- Hardware vs. software
- # Flow Tables, Flow Table sizes
- Supported matches + actions

- **Control Plane:** OpenFlow version + vendor extensions
- Rule updates (consistency, churn rate)Counters

Diversity is intrinsic:

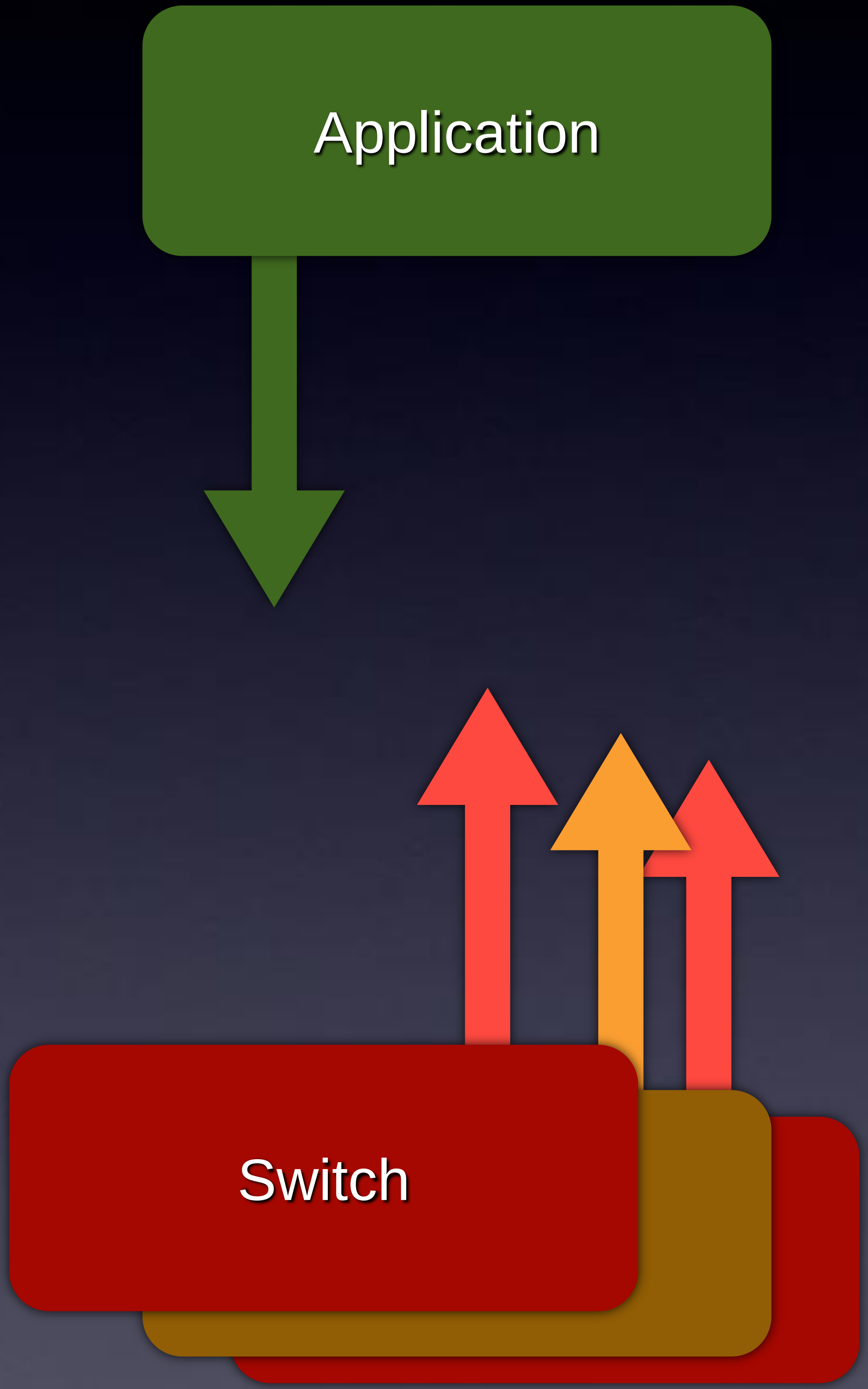
Usage Scenarios, Price Points, Diversification

The Gap

Expectations of the application

?

switch feature-sets and performance characteristics

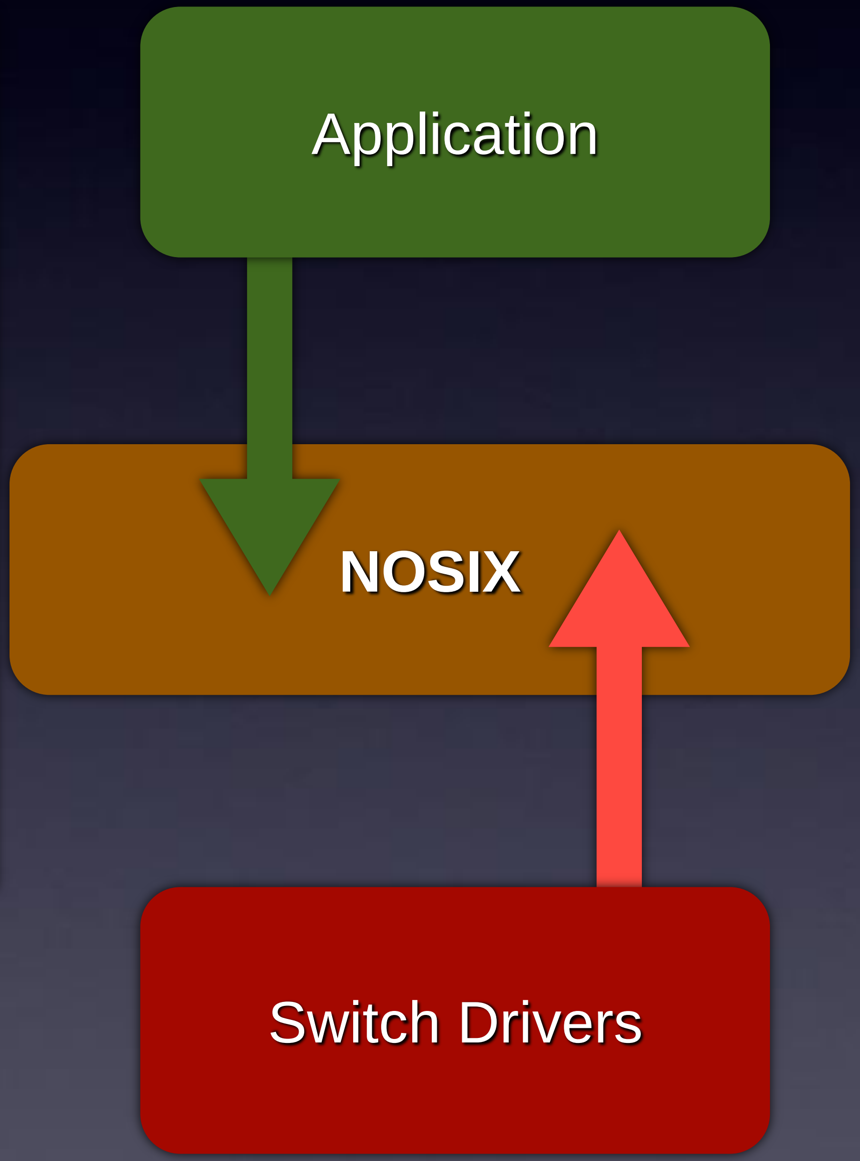


NOSIX

a lightweight portability API
in the **controller**

Applications
express
expectations

leverage
application
knowledge

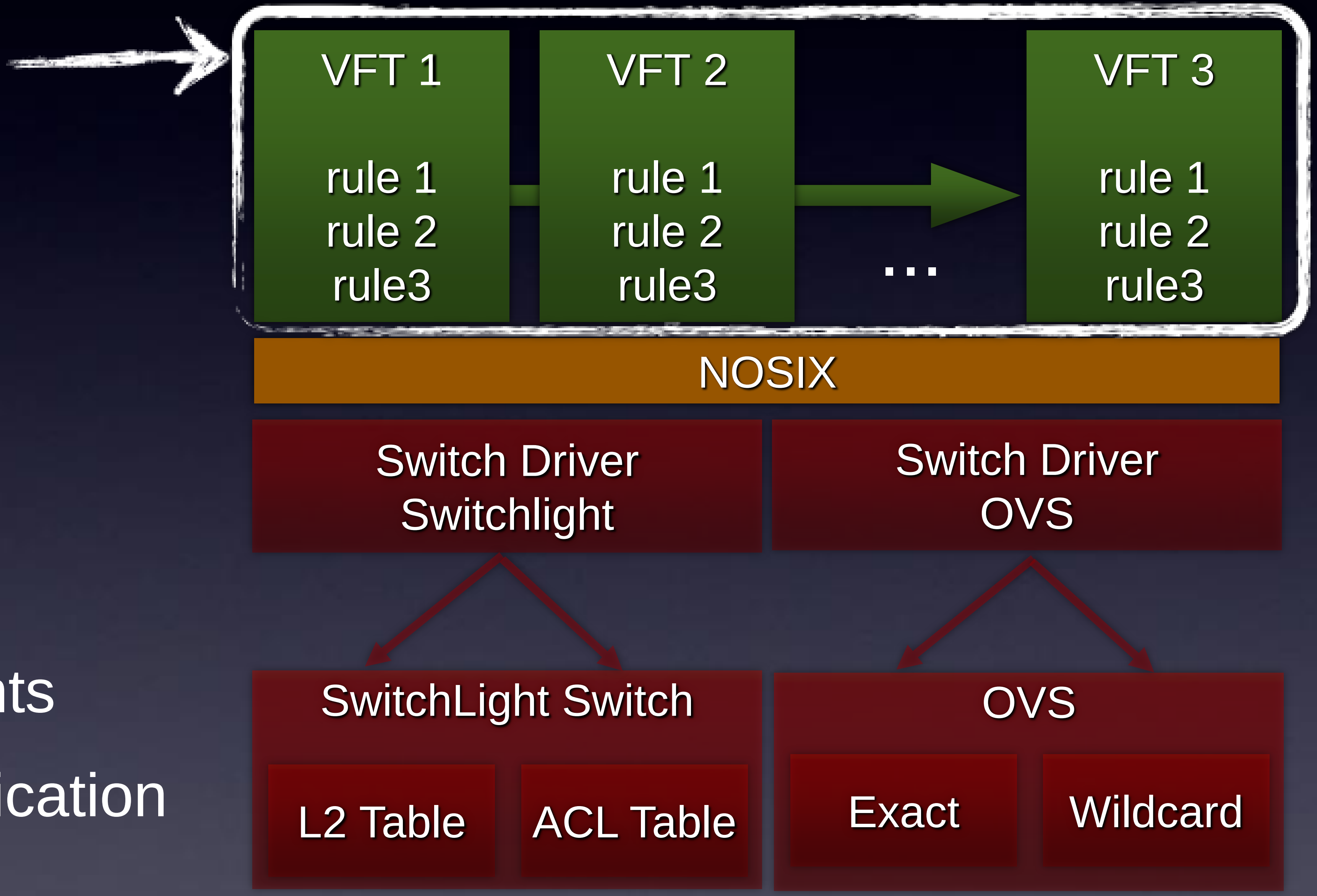


Drivers **map**
to available
features

leverage
vendor
knowledge

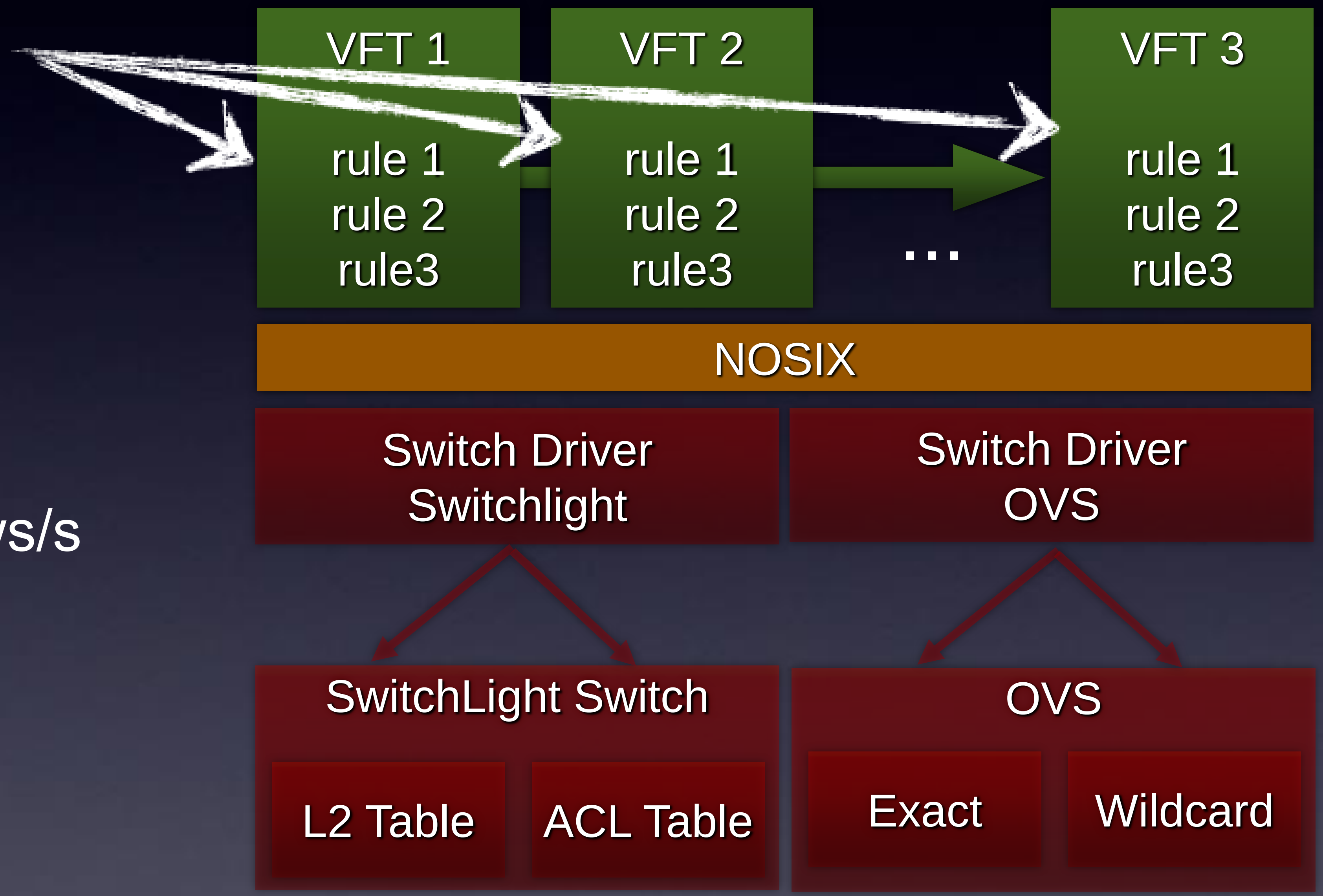
Core Concepts: Top Down

- **Pipeline of VFTs**
  Virtualized Flow Tables
  - Created by the Application
  - Pipelined
  - Default setting: 'portability'
    - Full Feature Set
    - No resource constraints
  - **Annotations** describe application expectations



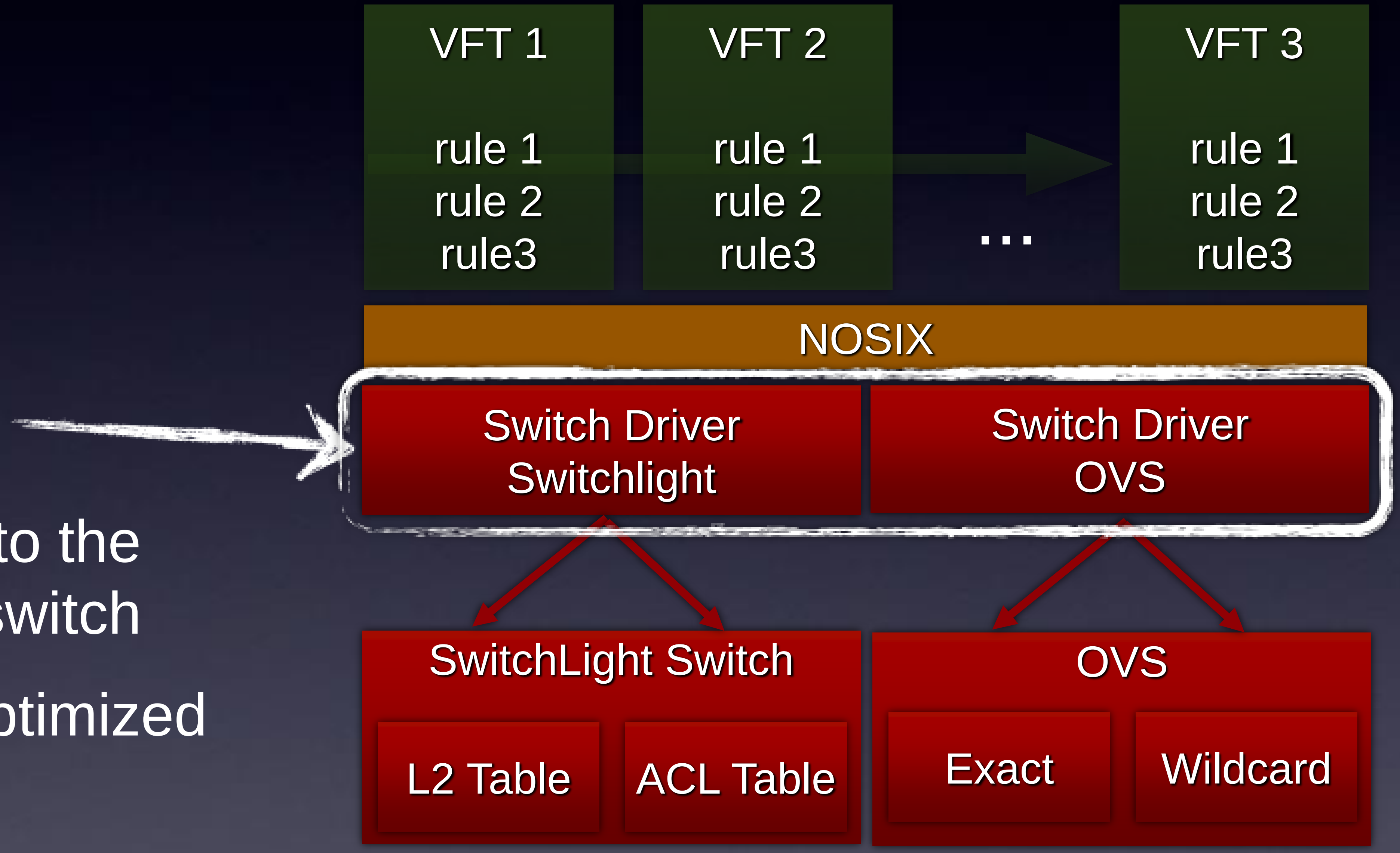
Core Concepts: Top Down

- **VFT Annotations**
  - **Requirements**
  - **throughput**
  - ≥ 500 Mbit/s
  - **churn**
  - ≥ 1000 flows/s
  - **Promises**
  - **only L2 matches**
  - **<= 100 Flows/s**
  - **Consistency**



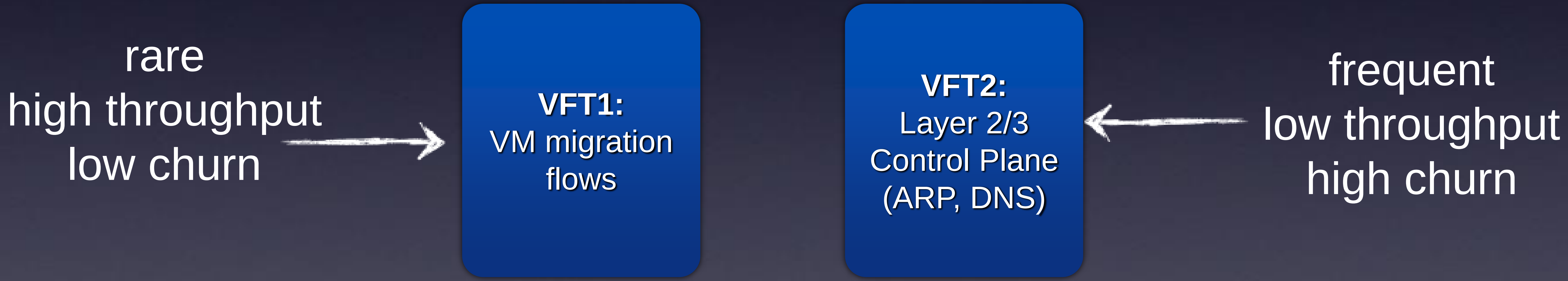
Core Concepts: Bottom Up

- **Switch Drivers**
  - Map the annotated VFTs to the physical flow tables in the switch
  - Use the annotations for optimized placement



Intuition

- Flows fall in natural groups
- Apps have information about the characteristics / allowable tradeoffs



rare
high throughput
low churn

VFT1:
VM migration
flows

VFT2:
Layer 2/3
Control Plane
(ARP, DNS)

frequent
low throughput
high churn

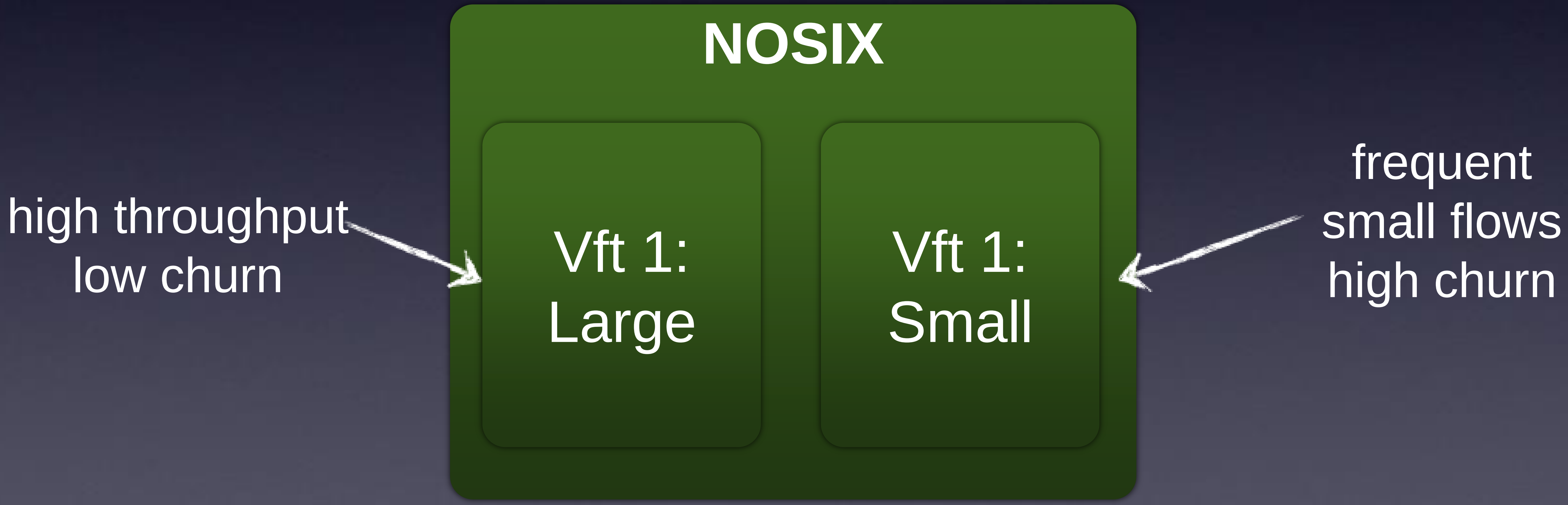
Case Study:

Flow Table Size Limit in a Simulated P-Switch

Access Control ➞ Microflows

80% small flows, 20 % large flows

grow # flows > flow table size



NOSIX

high throughput
low churn

Vft 1:
Large

Vft 1:
Small

frequent
small flows
high churn

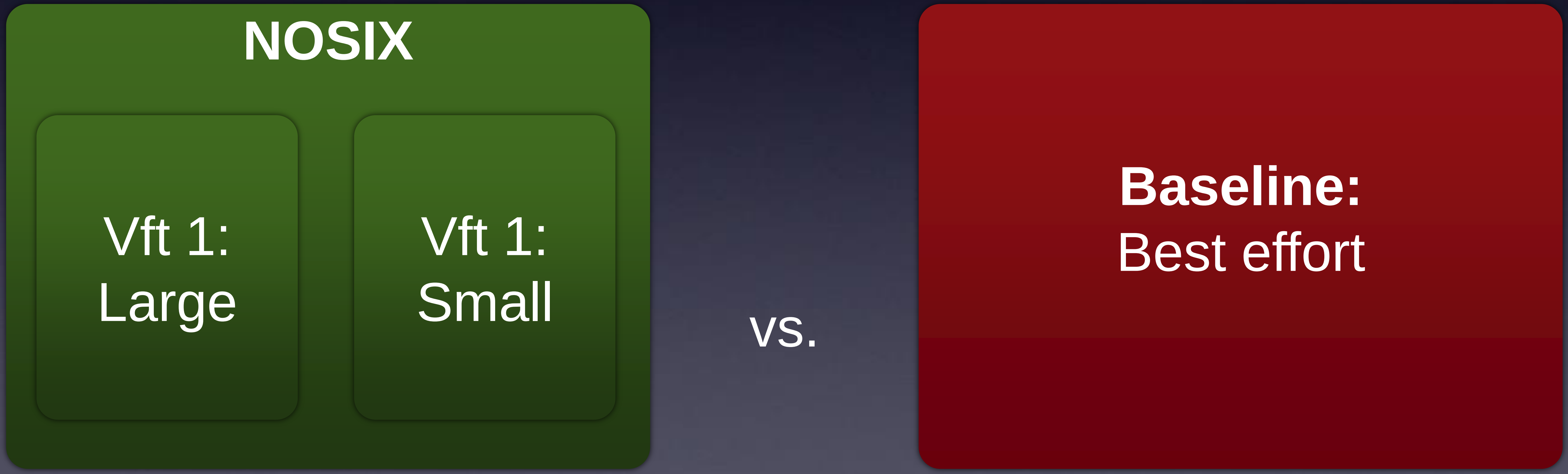
Case Study:

Flow Table Size Limit in a Simulated P-Switch

Access Control → Microflows

80% small flows, 20 % large flows

grow # flows > flow table size



NOSIX

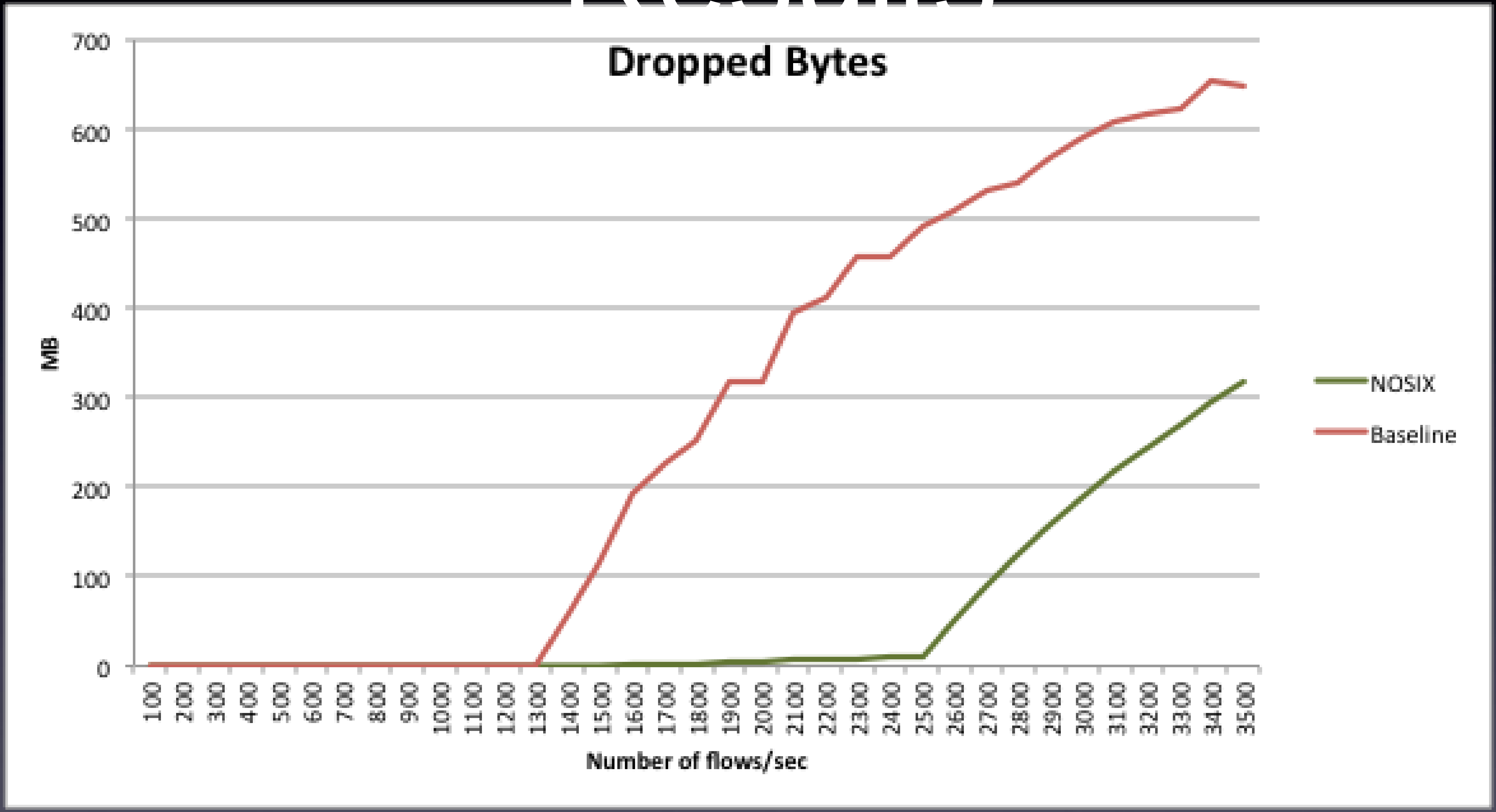
Vft 1:
Large

Vft 1:
Small

vs.

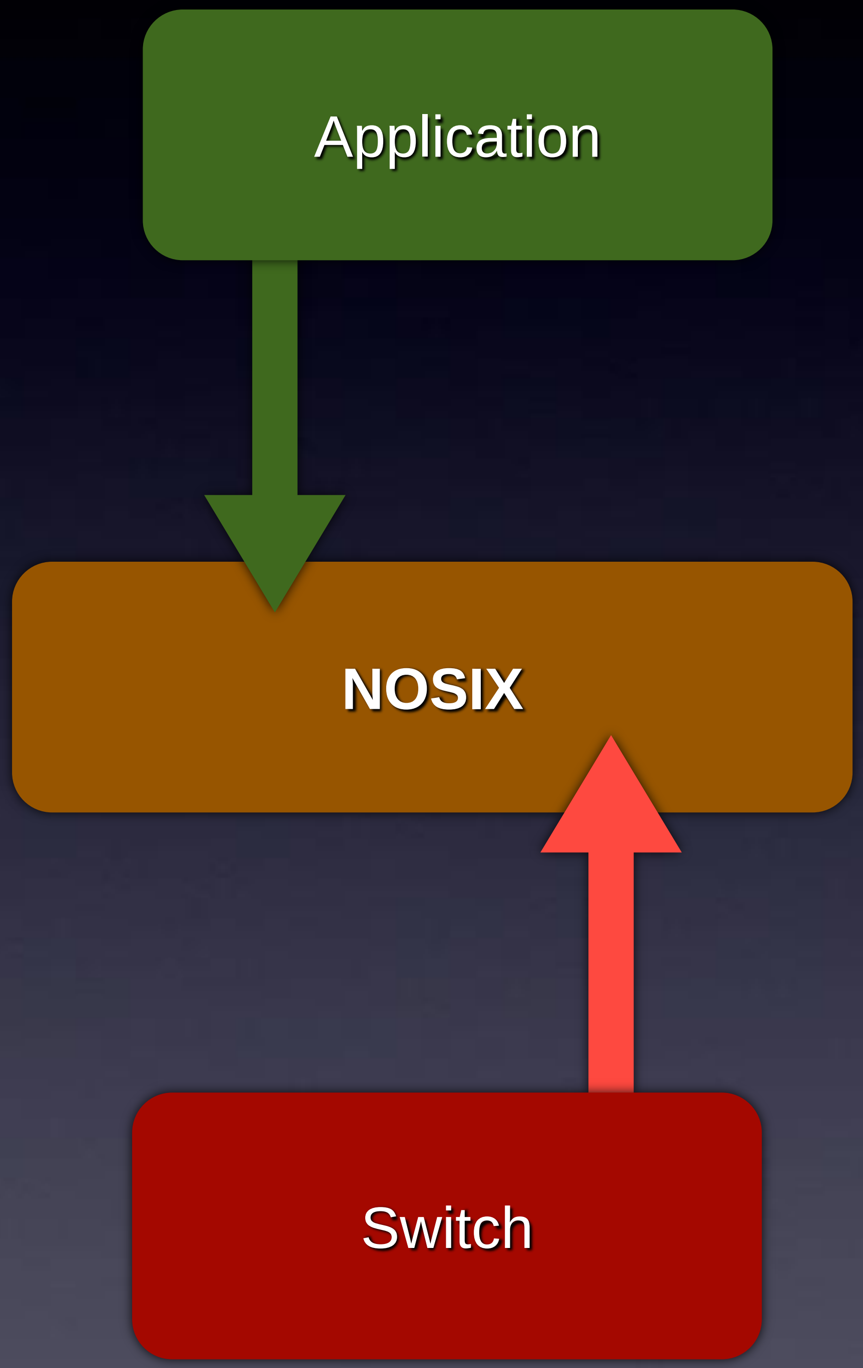
Baseline:
Best effort

Case Study: Simulation Results



Summary

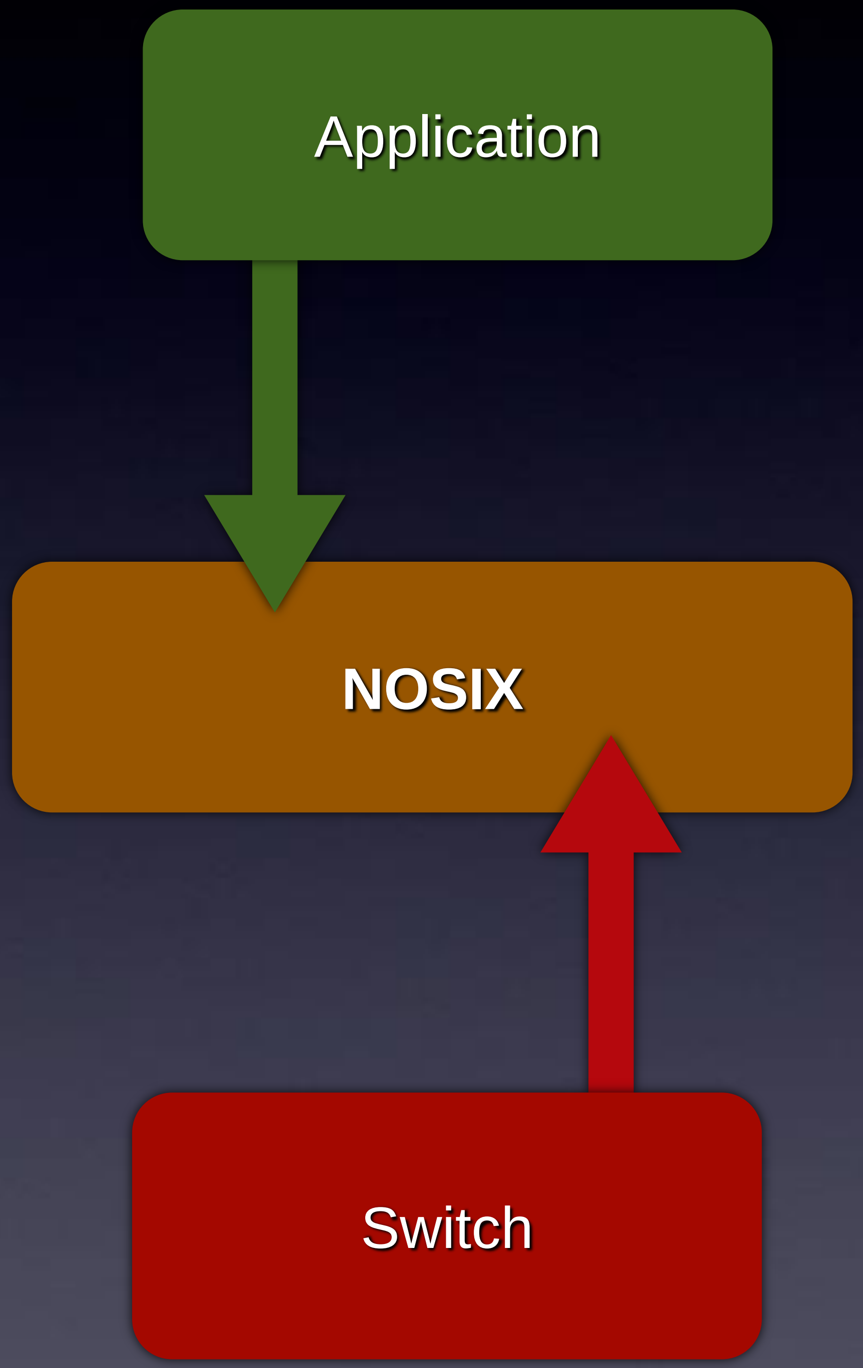
- Lightweight portability API in the **controller**
  - Applications express expectations
  - Switch drivers implement them
- Addresses portability challenges in SDN
- Building block for higher abstraction level controllers



Thank you.

Summary

- Lightweight portability API in the **controller**
  - Applications express expectations
  - Switch drivers implement them
- Addresses portability challenges in SDN
- Building block for higher abstraction level controllers



Summary

- lightweight portability API in the controller
- addresses portability challenges in SDN
- rendevous-point between
    - Application knowledges and Switch-Vendor Knowledge

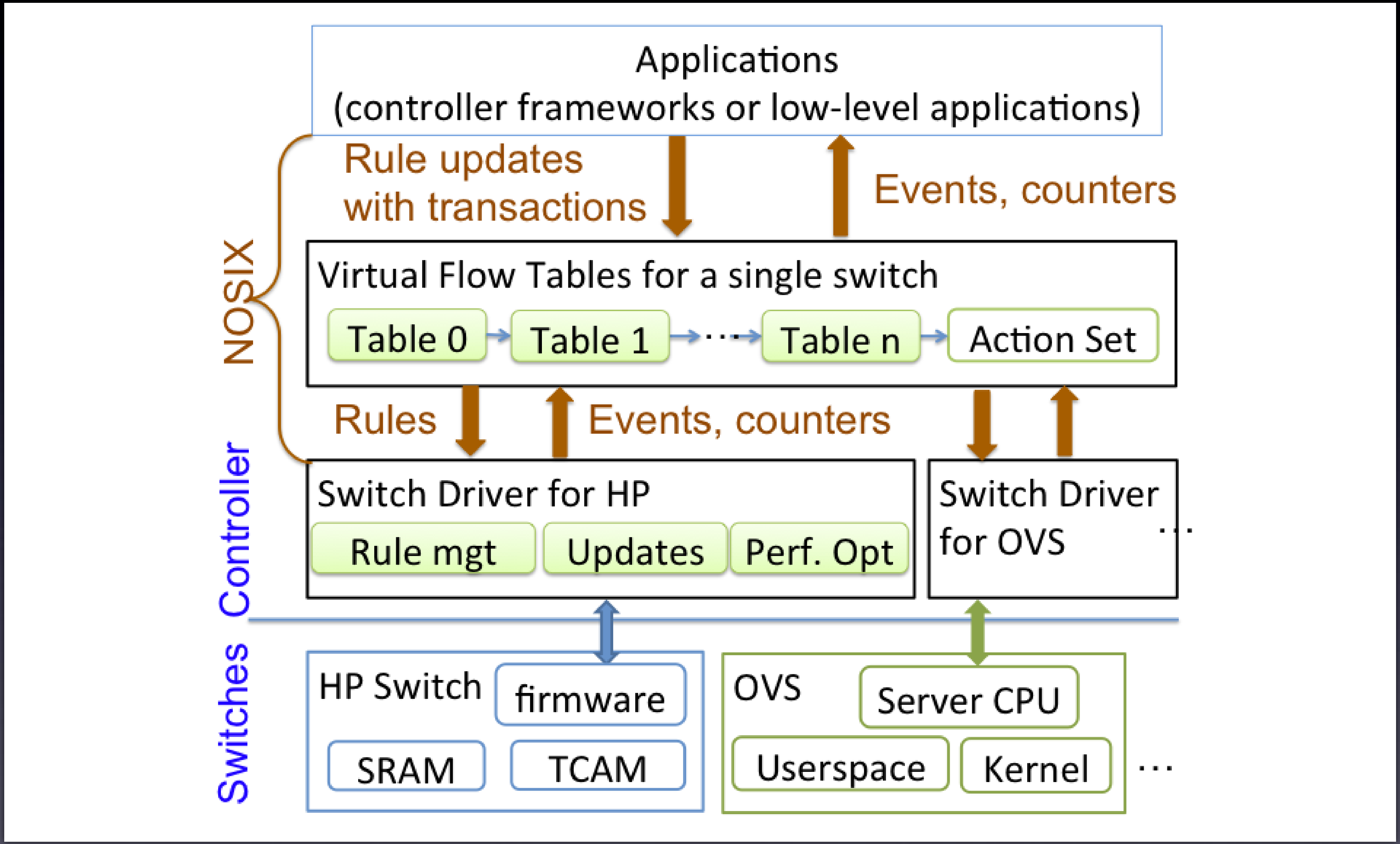
Backup

Implementation

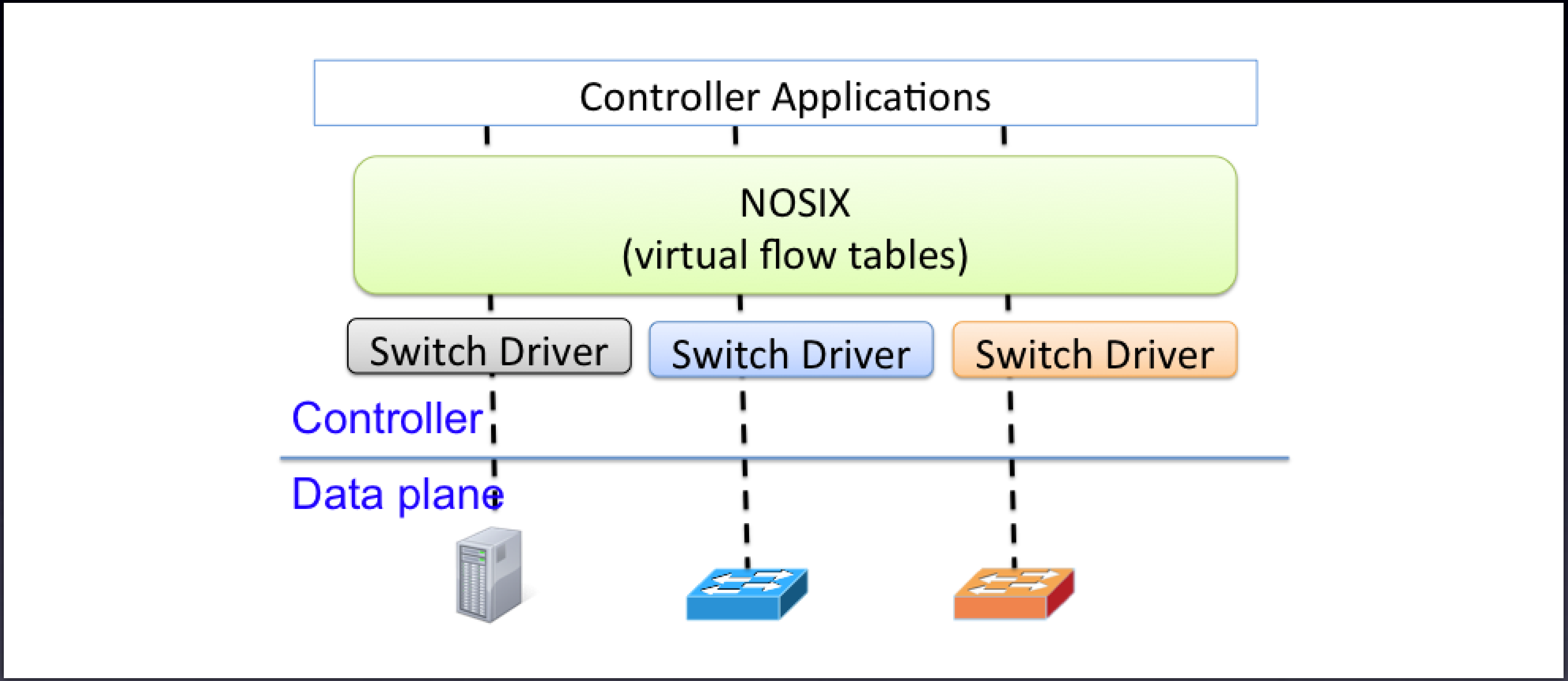
- **NOSIX**
  Generic Layer
  - Matches Annotations and Requirements to Switch Offerings
  - Can **virtualize** resource constraints
- E.g., rule paging to map 50k rules to 2k table entries

- **Switch Drivers**
  *Vendor provided*Provide Vendor/Switch Specific Knowledge Optimize for switch **specifics**E.g., knowledge of exact BARRIER Semantics
- Vendor extensions

Architecture



Architecture



Usage

Building block for higher level controller frameworks

Enables direct, portable development of low-level apps

Benefits

- Application-specific and switch-specific performance optimizations
- Enable protocol innovations by the vendors, e.g.,
  - built-in transactions for updates
  - efficient ruleset reconciliation after disconnect
- Annotations
  - provide a knob to choose between portability and performance

Use Case: Middlebox Loadbalancing

- 1 Switch, 2 Middleboxdes
- Reconfigure:
  - Consistency: Each (Pkt|Flow) handled by exactly 1 MB
- How to?
  - JRex (Overhead!)
  - Switch-specific (requires knowledge of BARRIER sem)
  - Vendor Extension?

Use Case: Middlebox Loadbalancing

```
vft = nosix.create_vft( requirements: { churn: >=10k },
                              promises: { rate <= 100k/s })

vft2 = nosix.create_vft( requirements: { churn: >=10k },
                              promises: { rate <= 100k/s })


nosix.transaction_mode(pkt_consistent)
vft.clear_flows()
for match, device in recalculate_flows():
vft.add_flow(match, output: device)
nosix.commit()
```

Use Case: Middlebox Loadbalancing

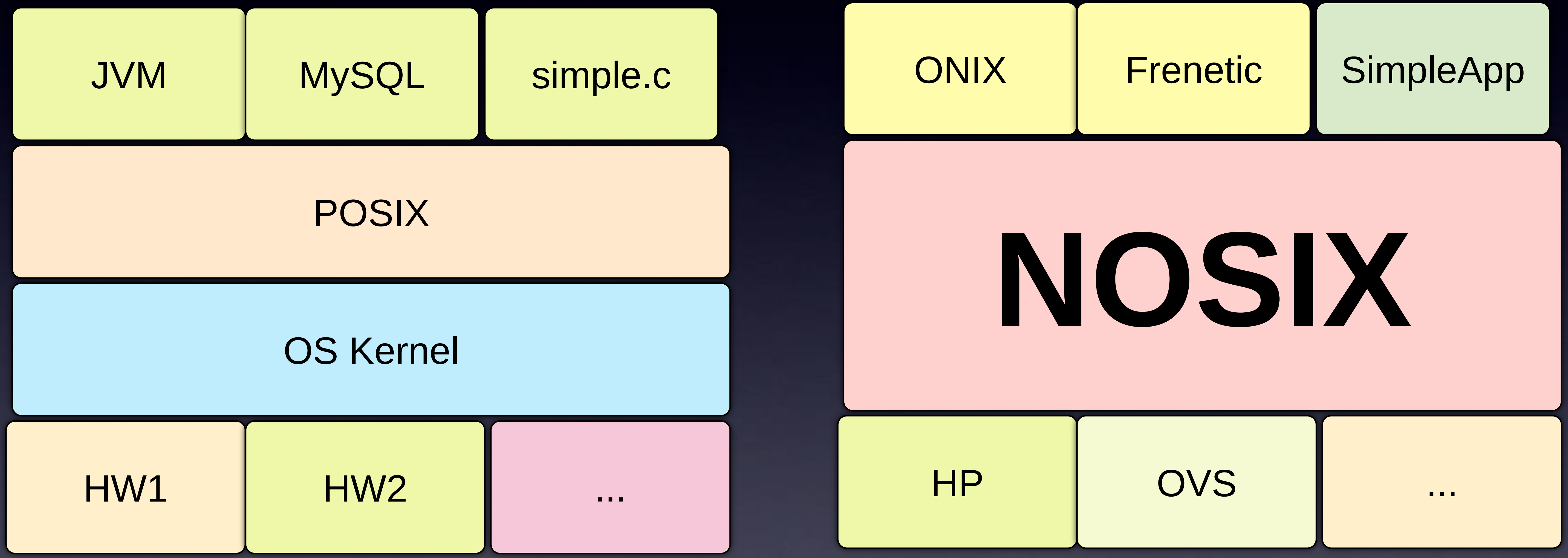
Optimization Options

Rule versioning *à la JREX*

Rule Reordering + Barriers

Shadow Flow Tables

OS vs. NOS



JVM | MySQL | simple.c

POSIX

OS Kernel

HW1 | HW2 | ...

ONIX | Frenetic | SimpleApp

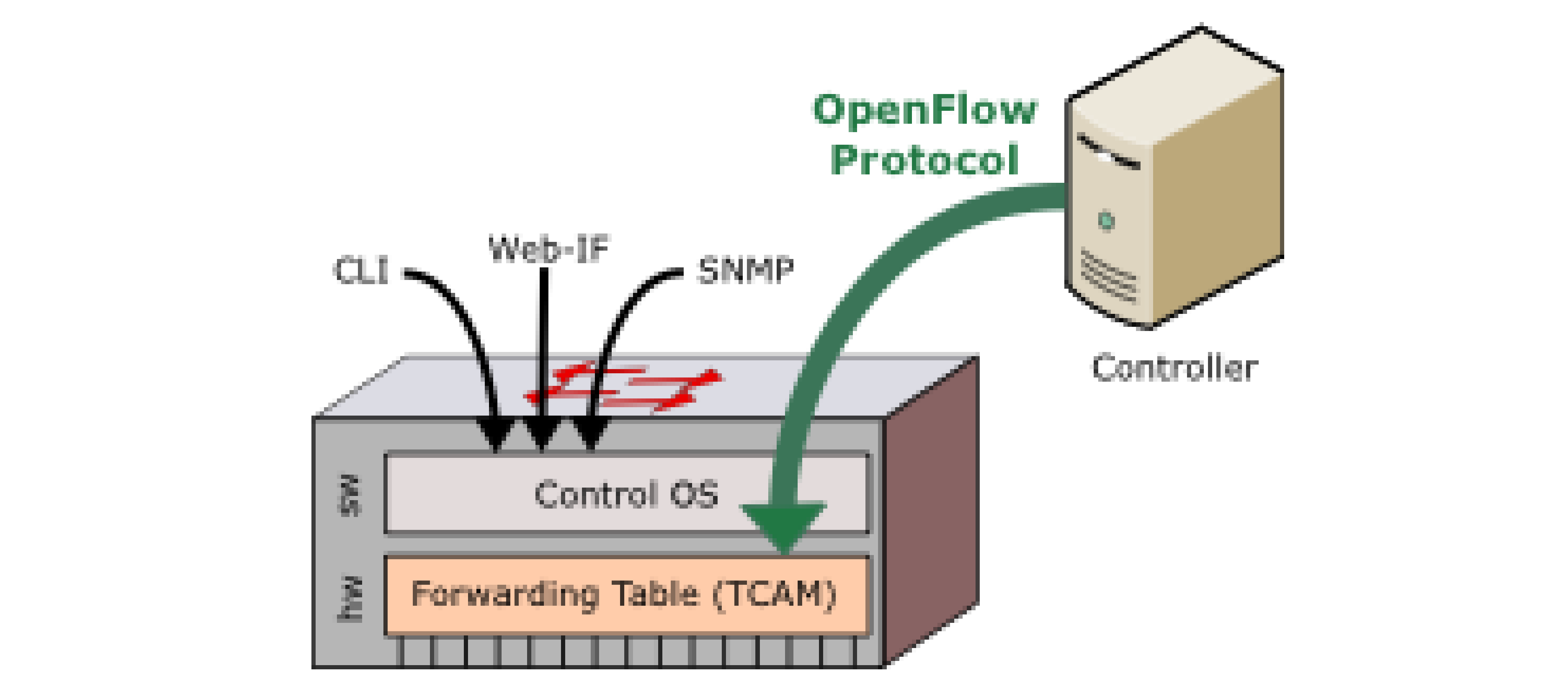
NOSIX

HP | OVS | ...

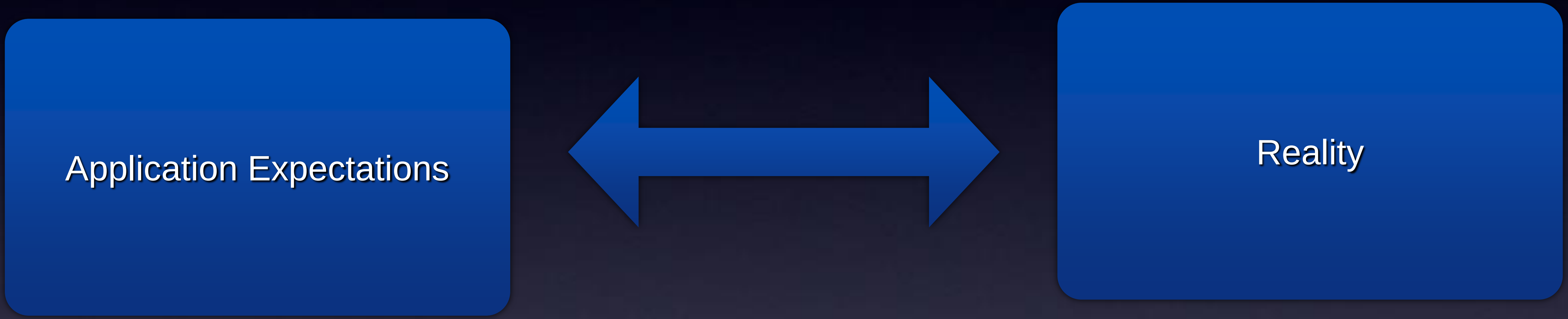
That is the idea. Start the flame throwers :)

Background

- OpenFlow enables control plane programmability...



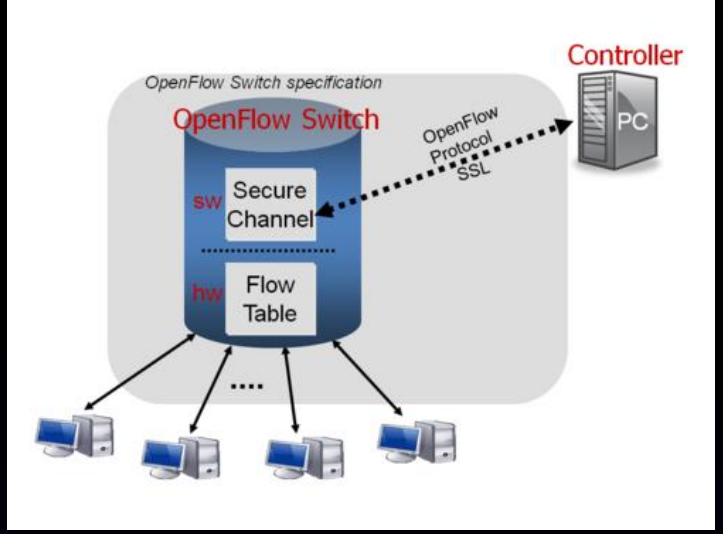
Mismatch between



Application Expectations

Reality

Expectations



- Homogeneous forwarding model
- Sufficiently large flow tables
- Predictable feature set and performance
- Switch state known / deltas efficiently reconcilable
- Support for fail-over

Reality

Heterogenous Switch landscape!

- **Data Plane:**
- Hardware vs. software
- Supported matches + actions
- Table count and sizes

- **Control Plane:**Rule updates (consistency, churn rate)CountersOpenFlow version + vendor extensions

Also: OF idiosyncracies

- With switch-side flow-expirations, flow table state is unknown
- Spurious PACKET_INs
- **Barrier** semantics switch dependent
- No efficient reconciliation of changes after disconnect

So far: Onix, POX, Frenetic...

- Manage the entire network
- Provide a simplified network-wide programming model, controller distribution, consistent updates, composability,...
- This requires making assumptions → optimize for a particular programming model
- All have to be adapted for each individual switch [class]
- **Duplication of effort**

Principles

- Applications expose expectations to the switch
- Vendors provide switch drivers in the controller

A Missing Piece in the Stack?