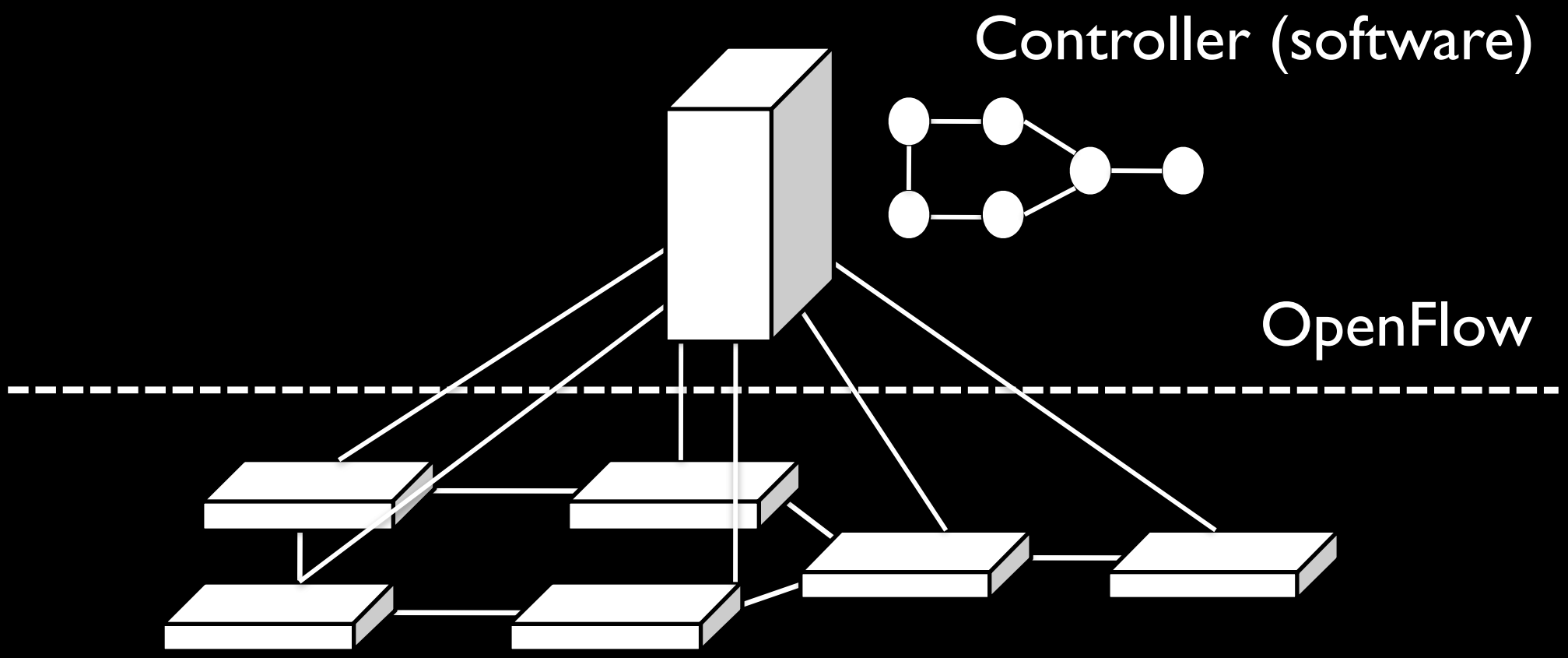
Transaction Defined Networking

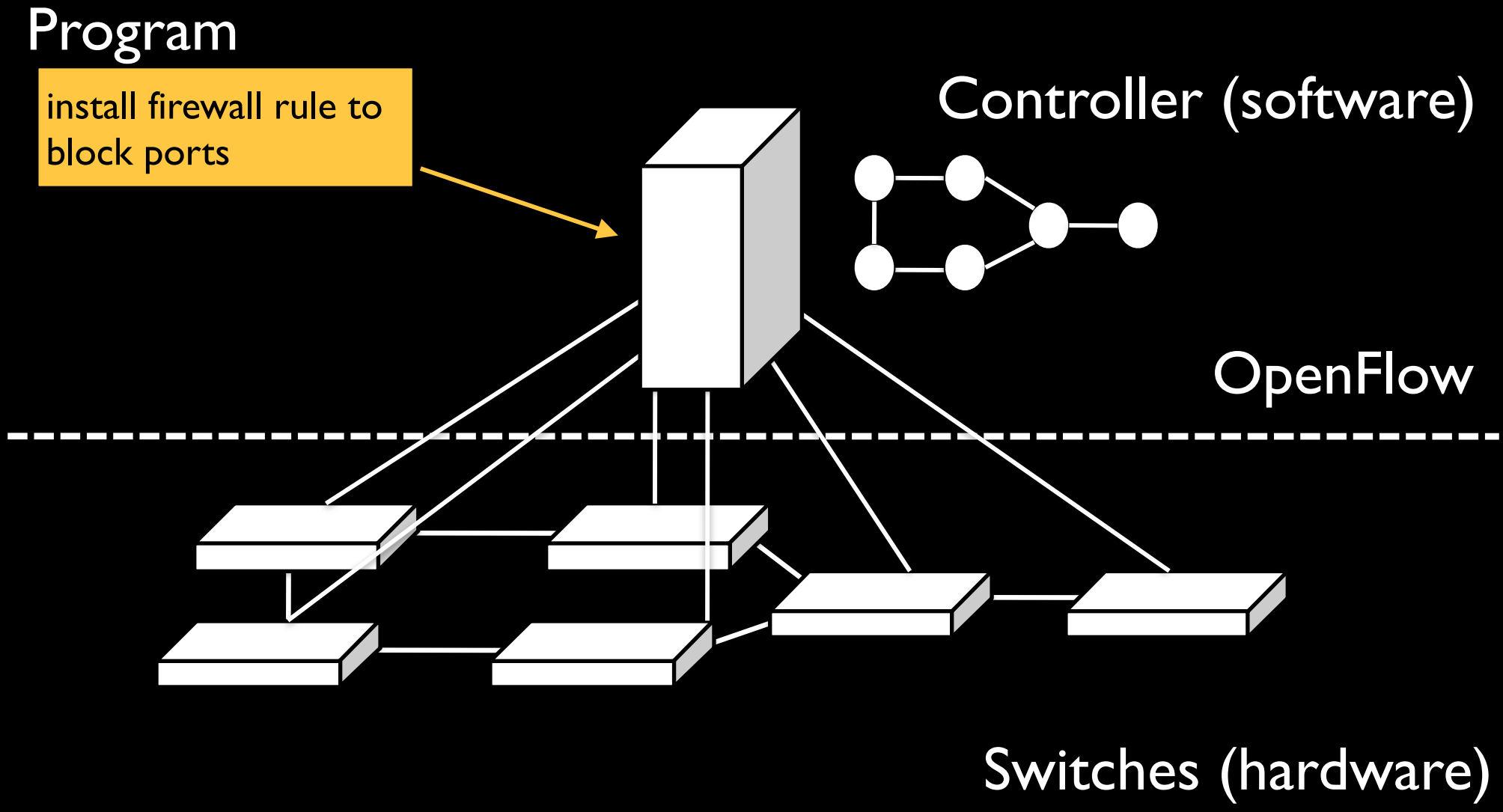
Behram Mistree and Philip Levis
Stanford University
ONRC, 5/12/2015

OpenFlow is the wrong low-level abstraction (compiler target) to program networks.

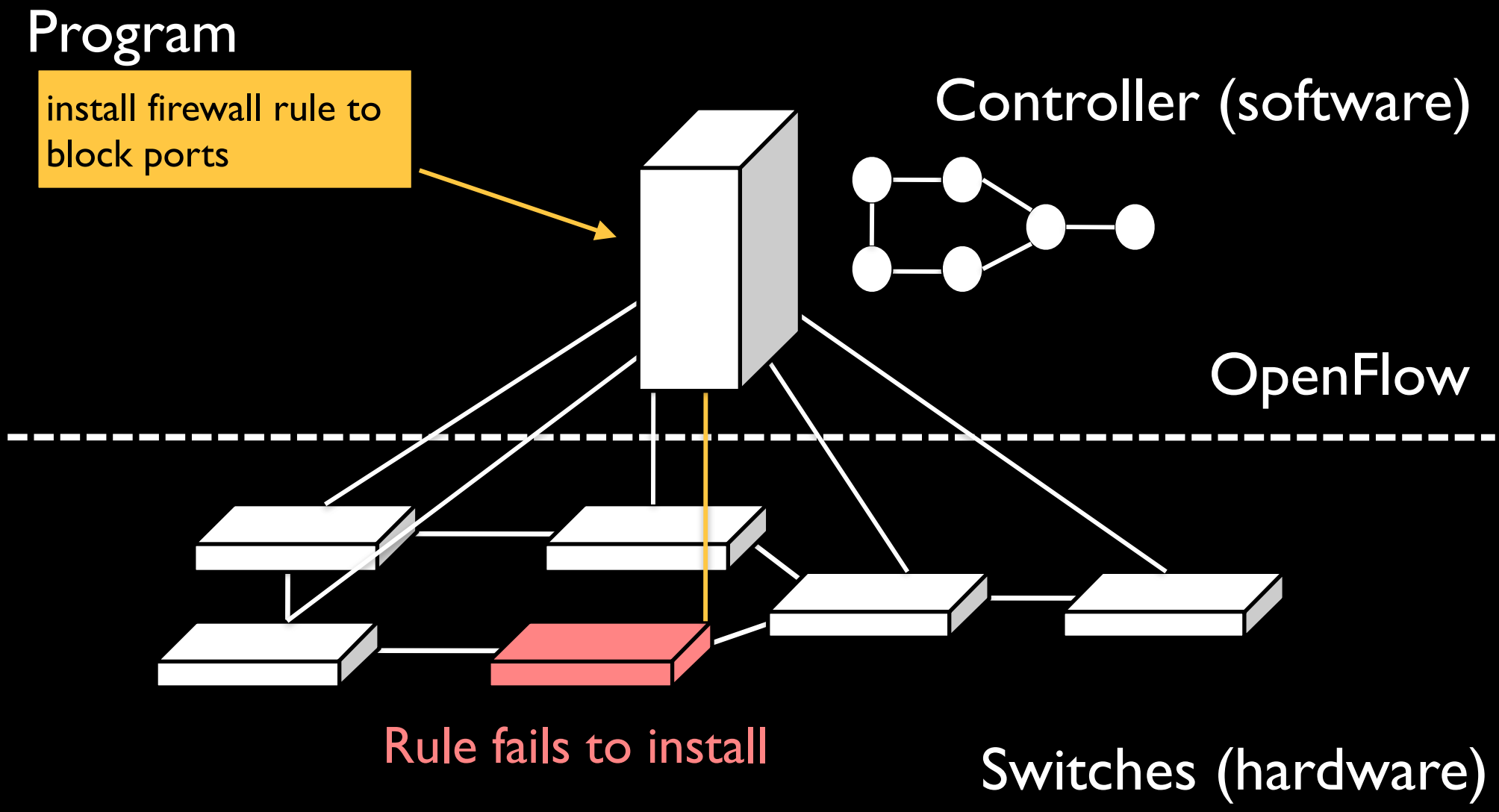
ONOS, Pyretic, Floodlight



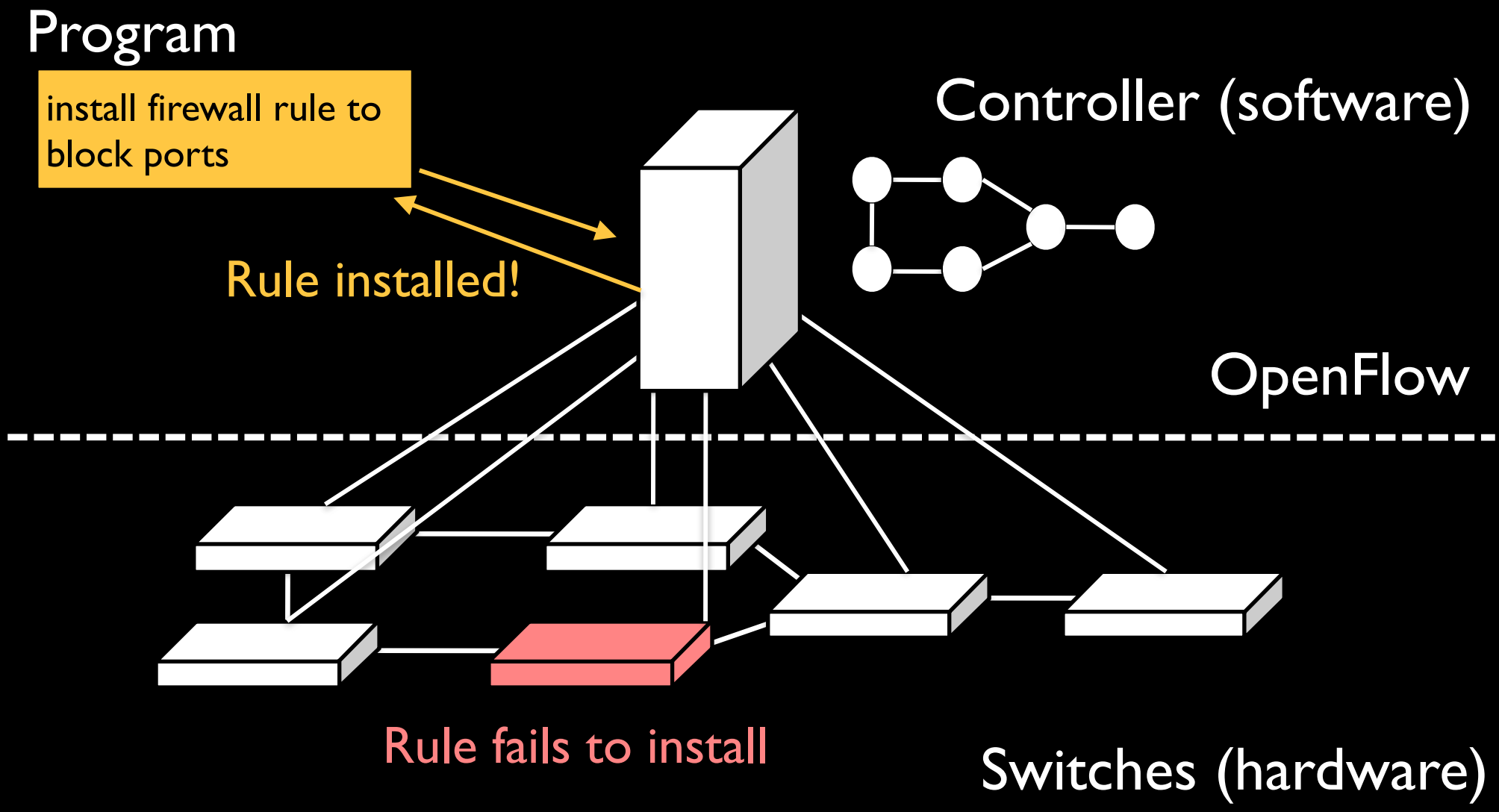
ONOS, Pyretic, Floodlight



ONOS, Pyretic, Floodlight



ONOS, Pyretic, Floodlight



The Problem

- Exists in ONOS, Pyretic, Floodlight, Onix
- Controller executes SDN program
  - Software state updates, program changes seem to have been installed
  - Controller fails to install changes properly
- Network state is permanently inconsistent with software state
  - Software says firewall rule is installed, but it's not, and never will be

The Reason

- OpenFlow's execution model is very difficult for a program or programmer to manage
  - Reordered commands
  - Rejected commands
- "Higher level" or "declarative" languages do not solve this problem
  - They can make it worse: why does a program fail?
- What compiler target should an SDN programming system provide?

First Principles

- A network's configuration transitions between known, well-defined states
  - Moving a flow does not result in a black hole
- A network does what software says it does
  - The firewall rule is actually installed
- Software sees well-defined network states
  - Don't have to cover every conceivable edge case
- Network is robust to controller crashes

First Principles

- A network's configuration transitions between known, well-defined states
  - Atomicity
- A network does what software says it does
  - Consistency
- Software sees well-defined network states
  - Isolation
- Network is robust to controller crashes
  - Durability

First Principles

- A network's configuration transitions between known, well-defined states
  - Atomicity
- A network does what software says it does
  - Consistency
- Software sees well-defined network states
  - Isola
- Netw s
  - Dura

ACID: transactions

Transaction Defined Networks

- A low-level transactional interface to an SDN
  - Programs execute with ACID semantics
  - Provide low-level resource access: flow tables
  - Prevents bugs encountered in existing systems
- A two-phase execution model and state sharding give high performance
  - 13,000 transactions/s on a single controller; 3,000 transactions/s on a distributed controller network
- Transactions enable new applications
  - Network version control and replay debugging

Outline

- Bugs in SDN controllers today
- Transaction defined networks
- Sergeant, a TDN system
- New network applications
- Conclusion

Reordering and Rejection

v1.4, §6.2: In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance;

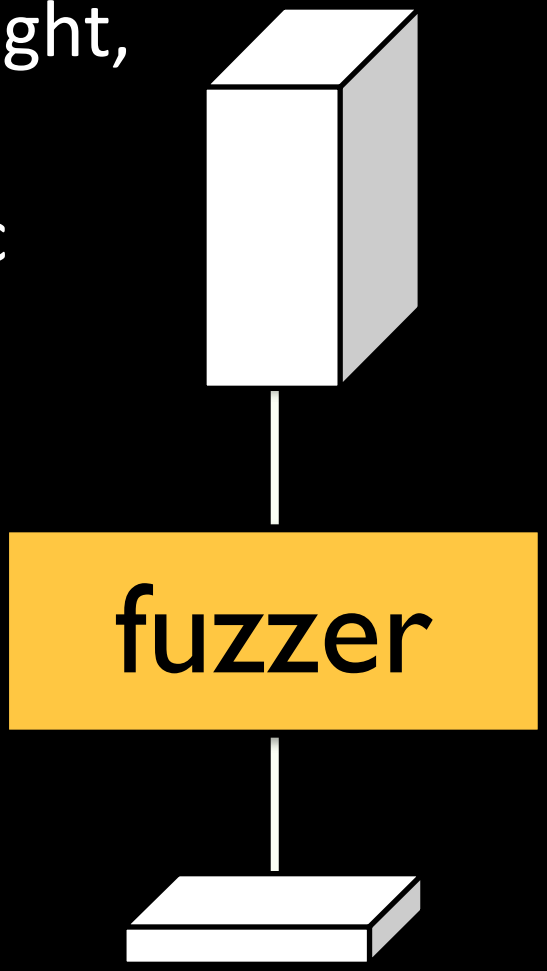
v1.4, §6.2: If a switch cannot completely process a message received from a controller, it must send back an error message.

Reordering/Rejection in Practice

- Reordering: Indigo2 switch firmware delays delete and modify flow commands to run after add commands
  - Reasoning: take longer to run, lower latency
- Rejection: switch TCAM is full
  - Mapping between flow mods and TCAM entries is non-trivial, requires careful management by software

Fuzz Testing

Floodlight,
ONOS,
Pyretic



1) Reorders, obeying barriers
2) Rejects

SDN Guarantees

System	Atomicity	Consistency	Isolation	Durability
Floodlight	No	No	No	No
ONOS	No	No	No	No
Pyretic	No	No	No	No
Onix	No	No	Yes	Yes
TDN	Yes	Yes	Yes	Yes

Floodlight, ONOS, Pyretic, Sergeant results from fuzz testing
Onix results from discussion with authors

Outline

- Bugs in SDN controllers today
- Transaction defined networks
- Sergeant, a TDN system
- New network applications
- Conclusion

Transaction Defined Networks

- Imperative programs modify flow tables
- These programs execute transactionally
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- When a TDN system reports a program has executed successfully, the change is now active in the network

Two Techniques

- Two-phase execution
  - Software program executes transactionally
  - When software completes, system compiles set of commands, pushes them transitionally to switches
- Sharded state architecture
  - Each controller in a large system is responsible for a subset of the network state
  - Distributed transactions across multiple controllers can make large changes, but small local changes only require one controller

Two Techniques

- Two-phase execution
  - Software program executes transactionally
  - When software completes, system compiles set of commands, pushes them transitionally to switches
- Sharded state architecture
  - Each controller in a large system is responsible for a subset of the network state
  - Distributed transactions across multiple controllers can make large changes, but small local changes only require one controller

Separate *What* from *How*

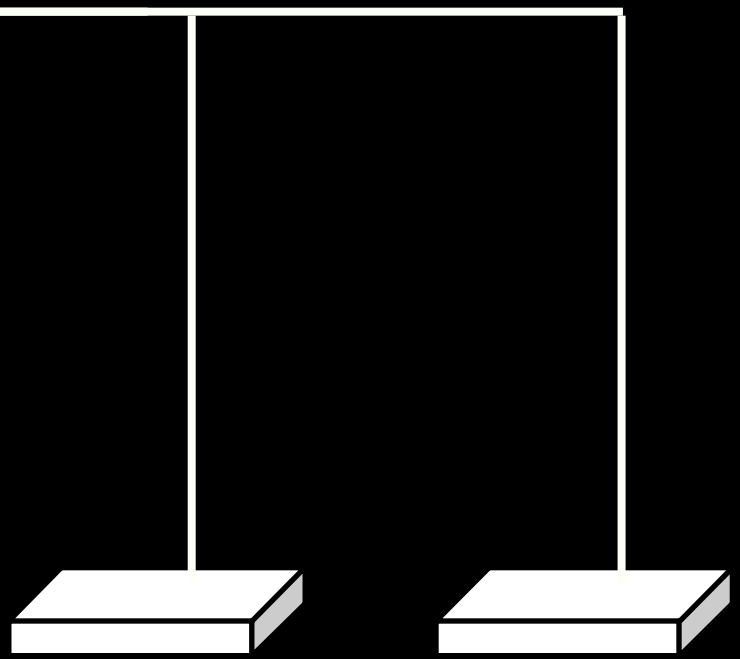
Program arrives

Controller

Match	Action	Priority

install firewall rules to block ports

Match	Action	Priority



Match	Action	Priority

Match	Action	Priority

Separate *What* from *How*

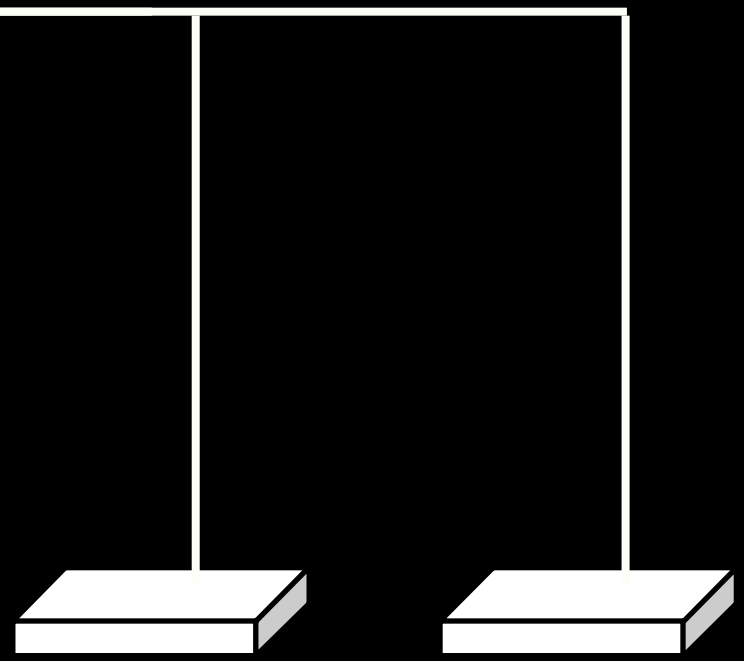
Controller

Match	Action	Priority

Match	Action	Priority

install firewall rules to block ports

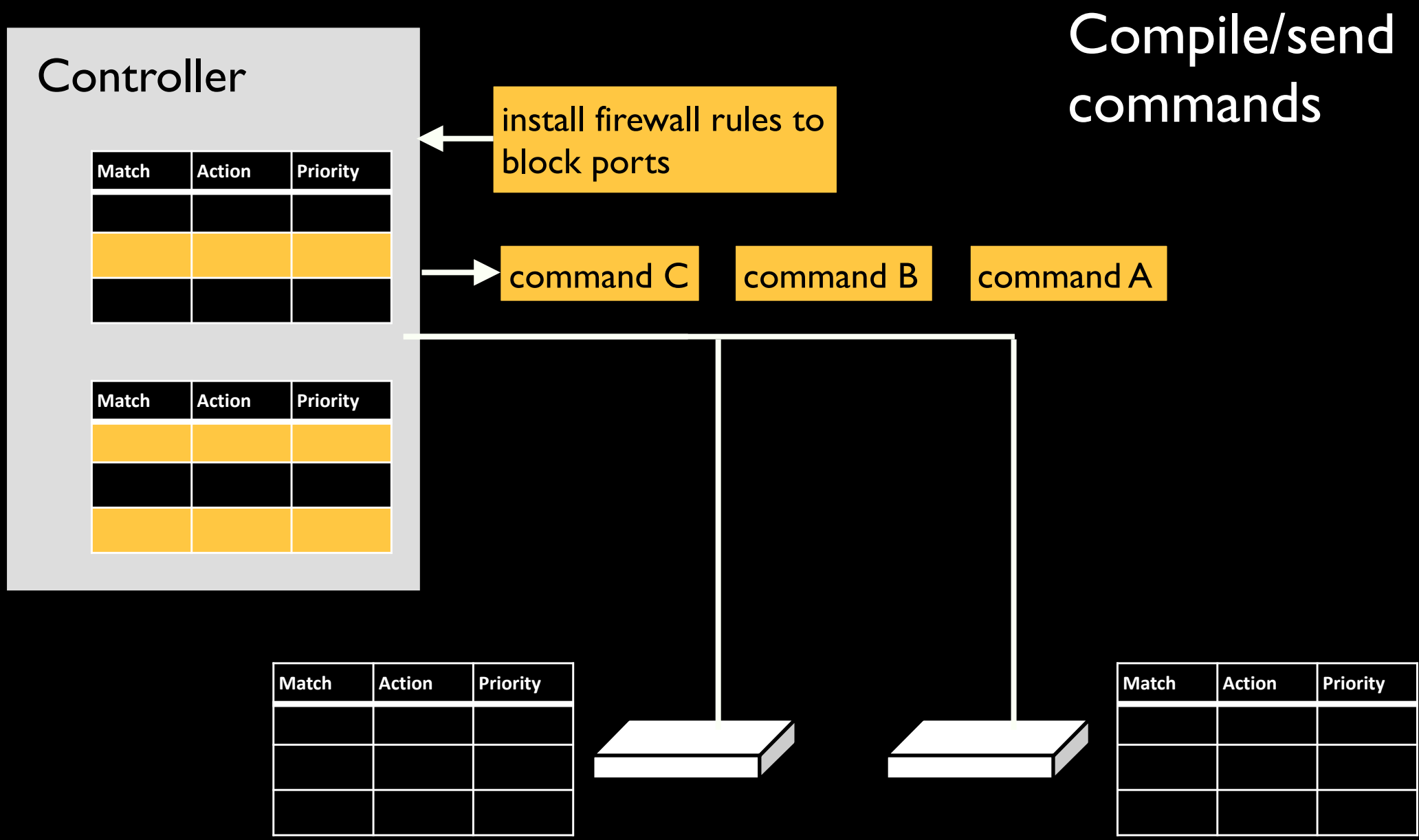
Execute software



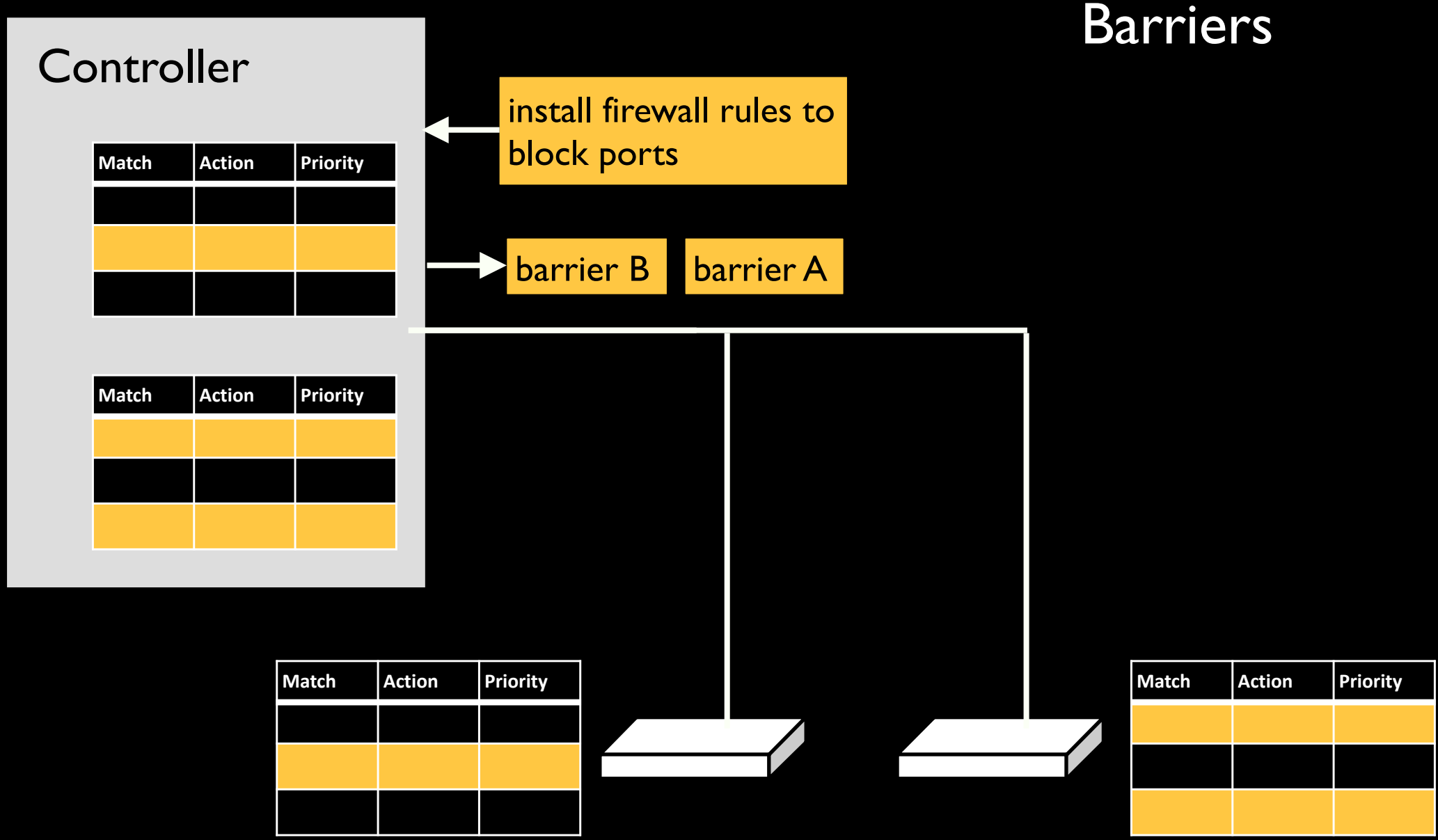
Match	Action	Priority

Match	Action	Priority

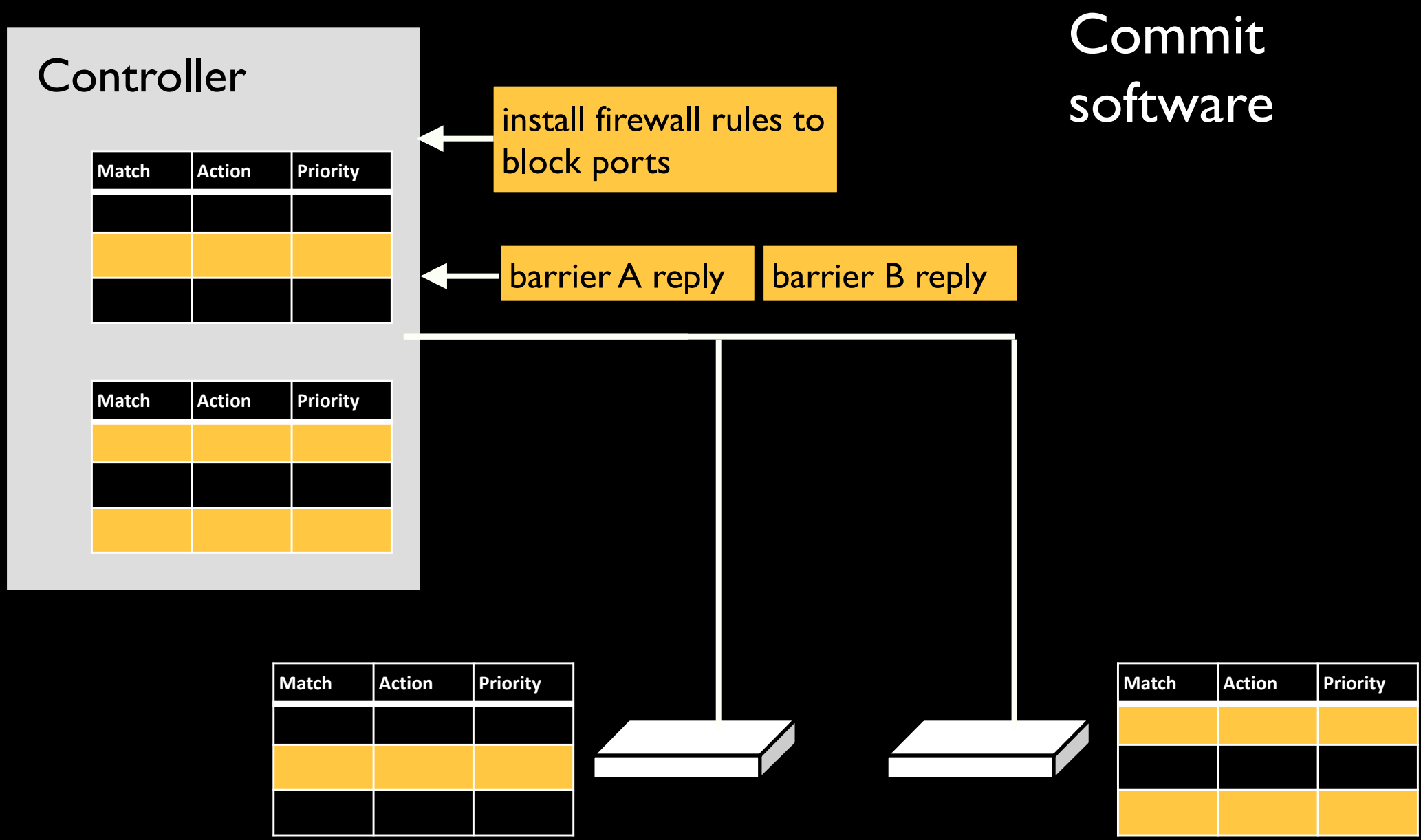
Separate *What* from *How*



Separate *What* from *How*



Separate *What* from *How*



Separate *What* from *How*

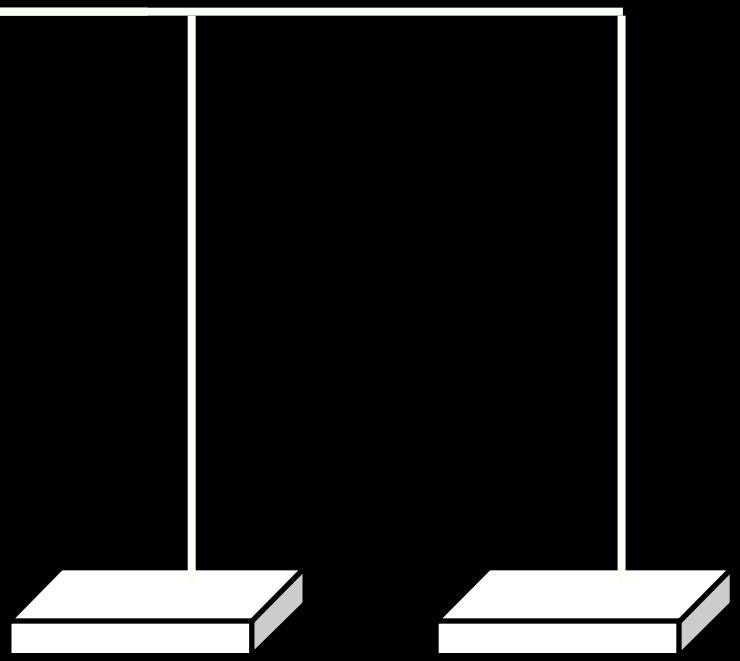
Complete

Controller

Match	Action	Priority

Match	Action	Priority

install firewall rules to block ports



Match	Action	Priority

Match	Action	Priority

Software Execution

- Borrows from database literature
- Software acquires locks as it executes
  - Per-software object locks, each switch is one object
- Deadlock detection/avoidance
  - Programs can preempt one another and steal locks
  - Stolen-from program rolls back
  - More later on lock stealing policy
- Dynamic lock acquisition enables fine-grained, precise locking, important in SDN context

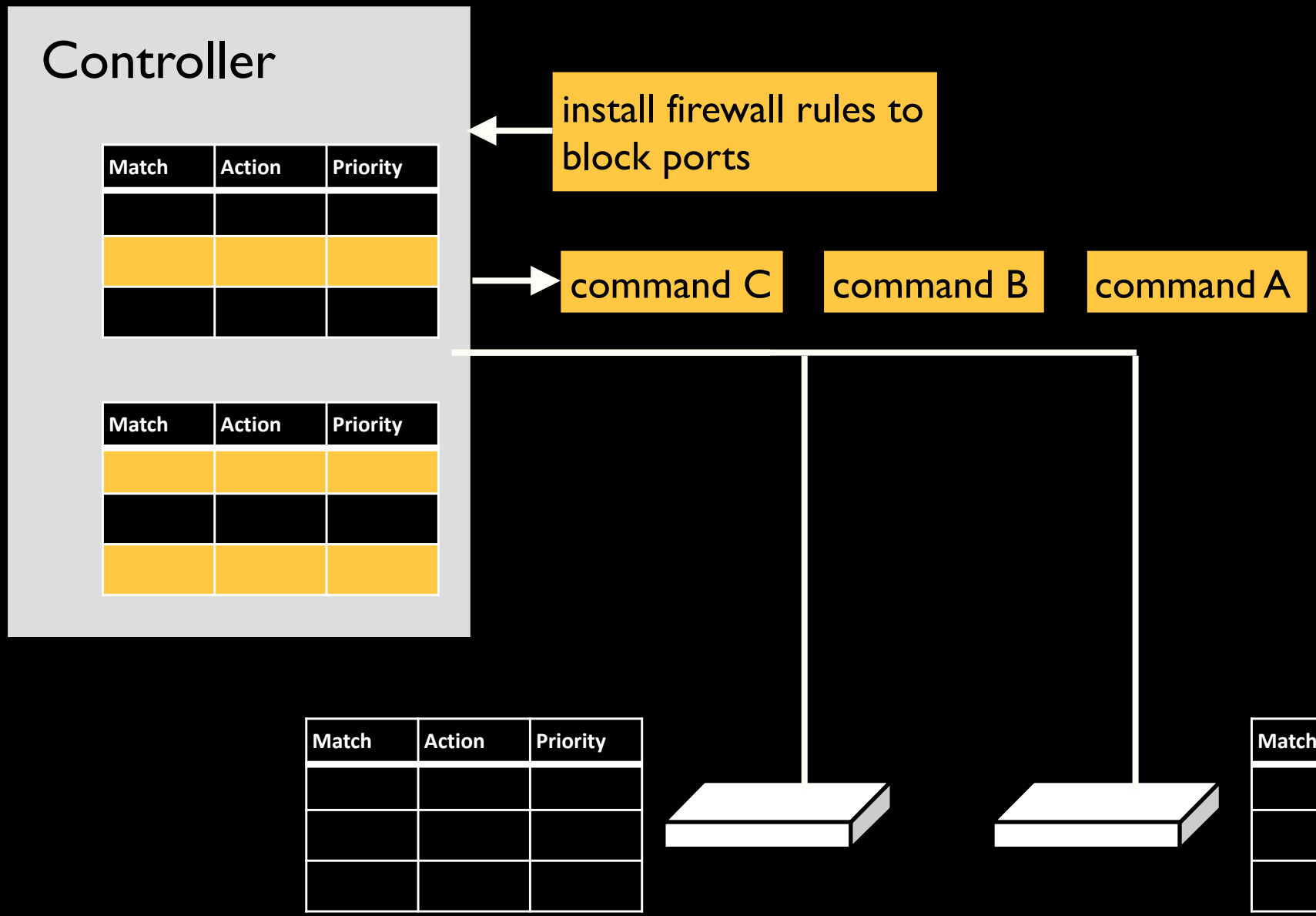
Compilation/Execution

- Runtime compiles software changes to OpenFlow reordering-robust commands
  - Separates *what* from *how*
- Issues commands to switches
- Issues a barrier command
- When barrier complete command received, commit software phase and return to caller

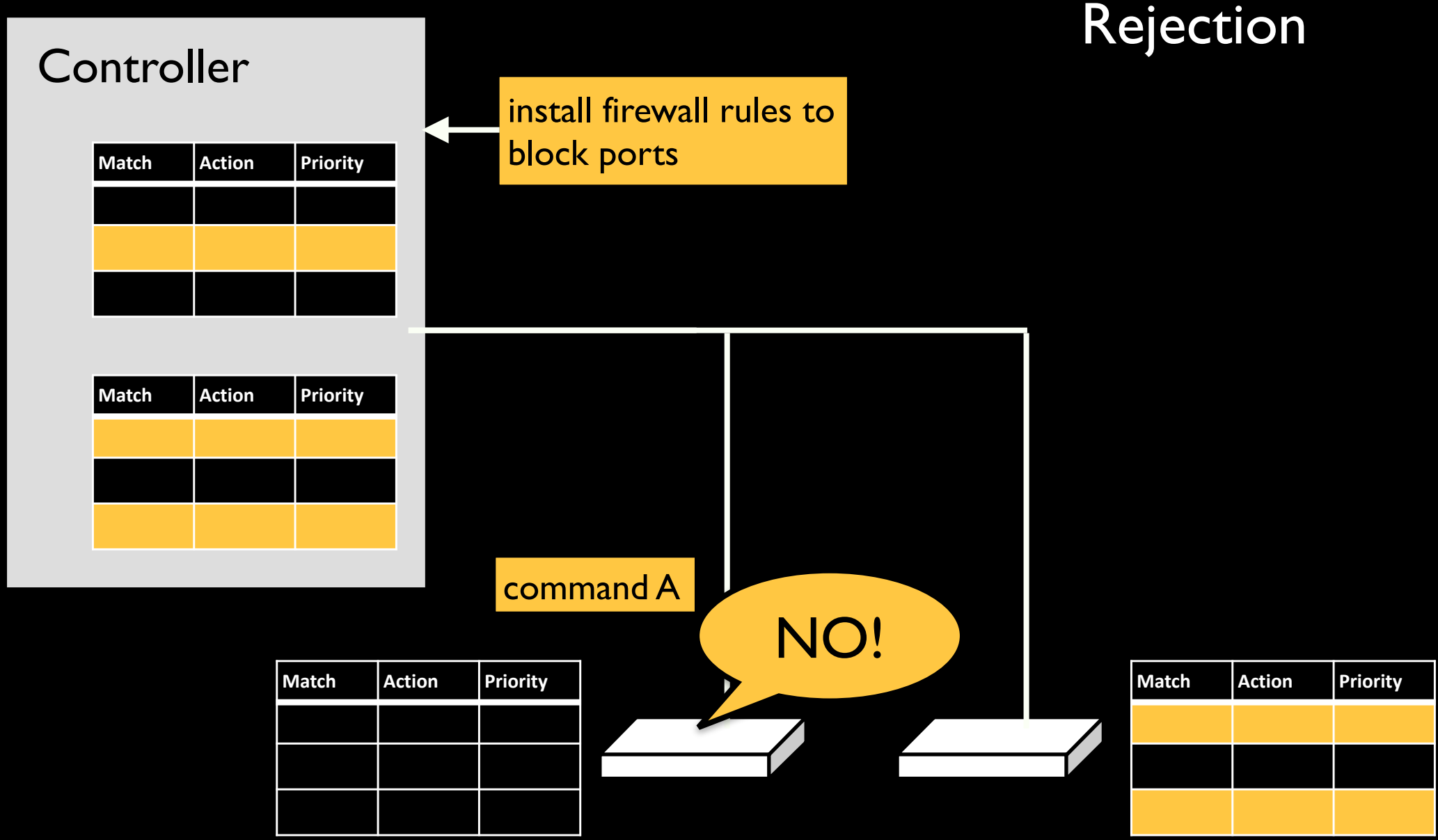
Handling Rejection

- Controller keeps record of generated commands for each switch
- If any switch rejects a command, controller generates an *undo set* of commands to roll back changes
- Rolls back software program, retries, eventually times out and returns error

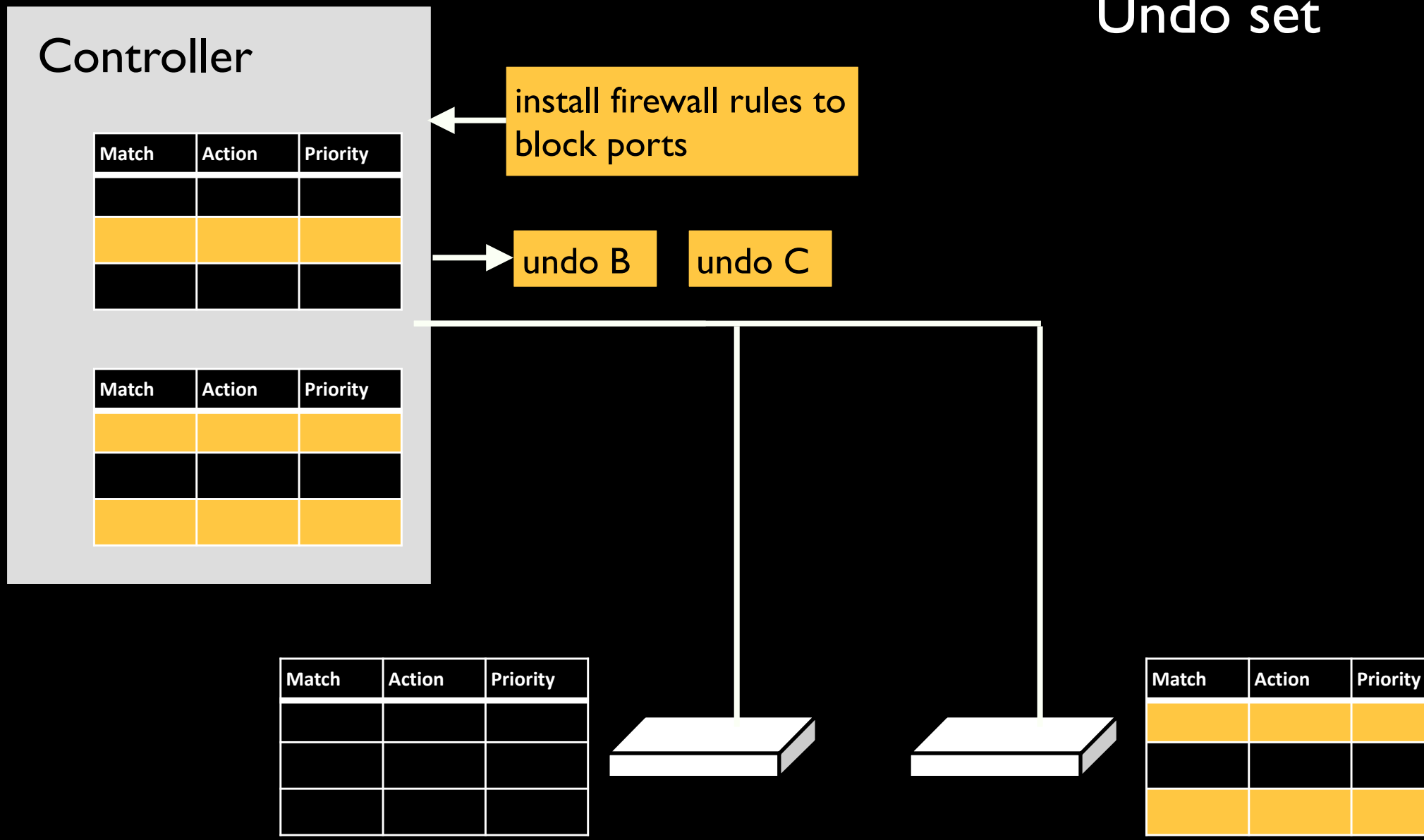
Handling Rejection



Handling Rejection



Handling Rejection



Handling Rejection

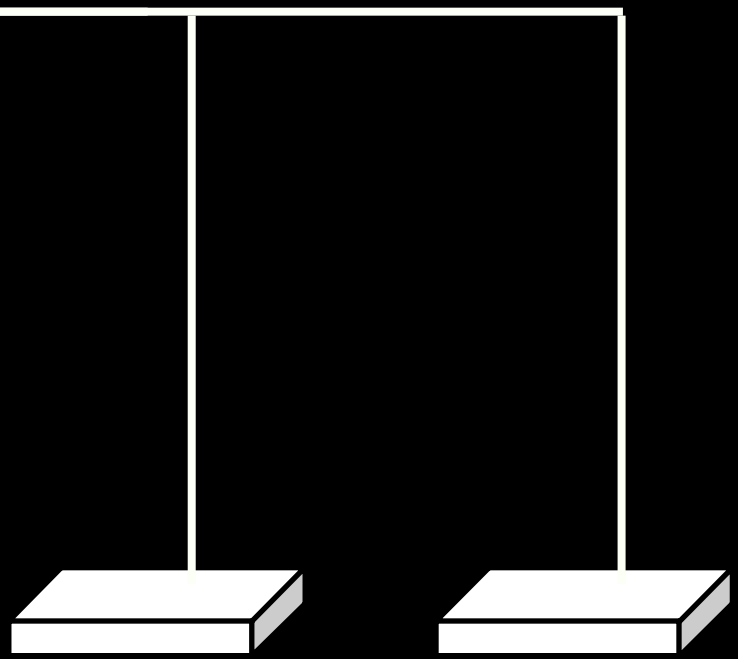
Execute undo

Controller

install firewall rules to block ports

Match	Action	Priority

Match	Action	Priority



Match	Action	Priority

Match	Action	Priority

Handling Rejection

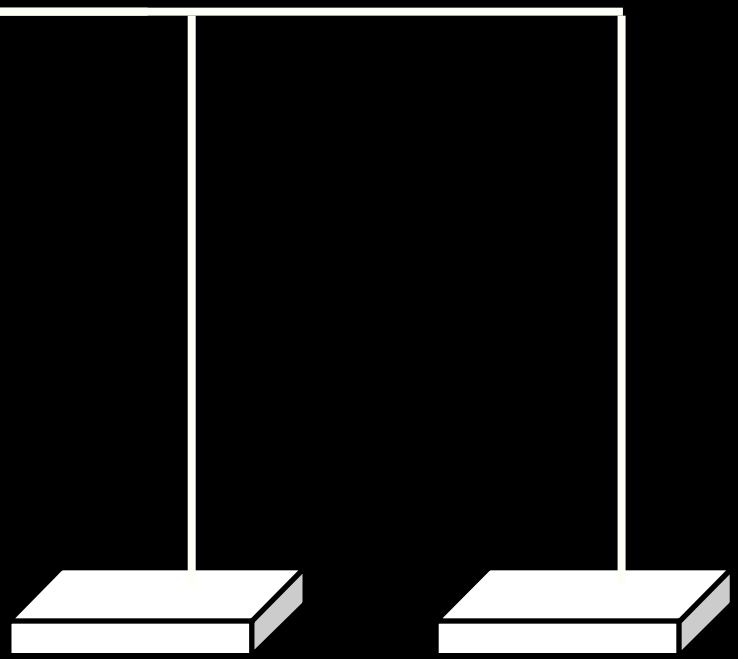
Roll back software

Controller

install firewall rules to block ports

Match	Action	Priority

Match	Action	Priority



Match	Action	Priority

Match	Action	Priority

Handling Rejection

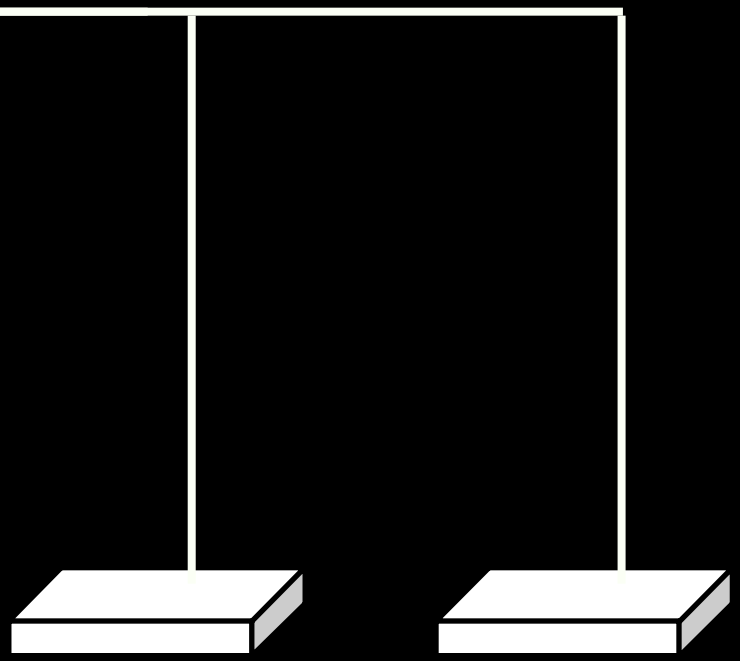
Retry

Controller

Match	Action	Priority

Match	Action	Priority

install firewall rules to block ports — retry



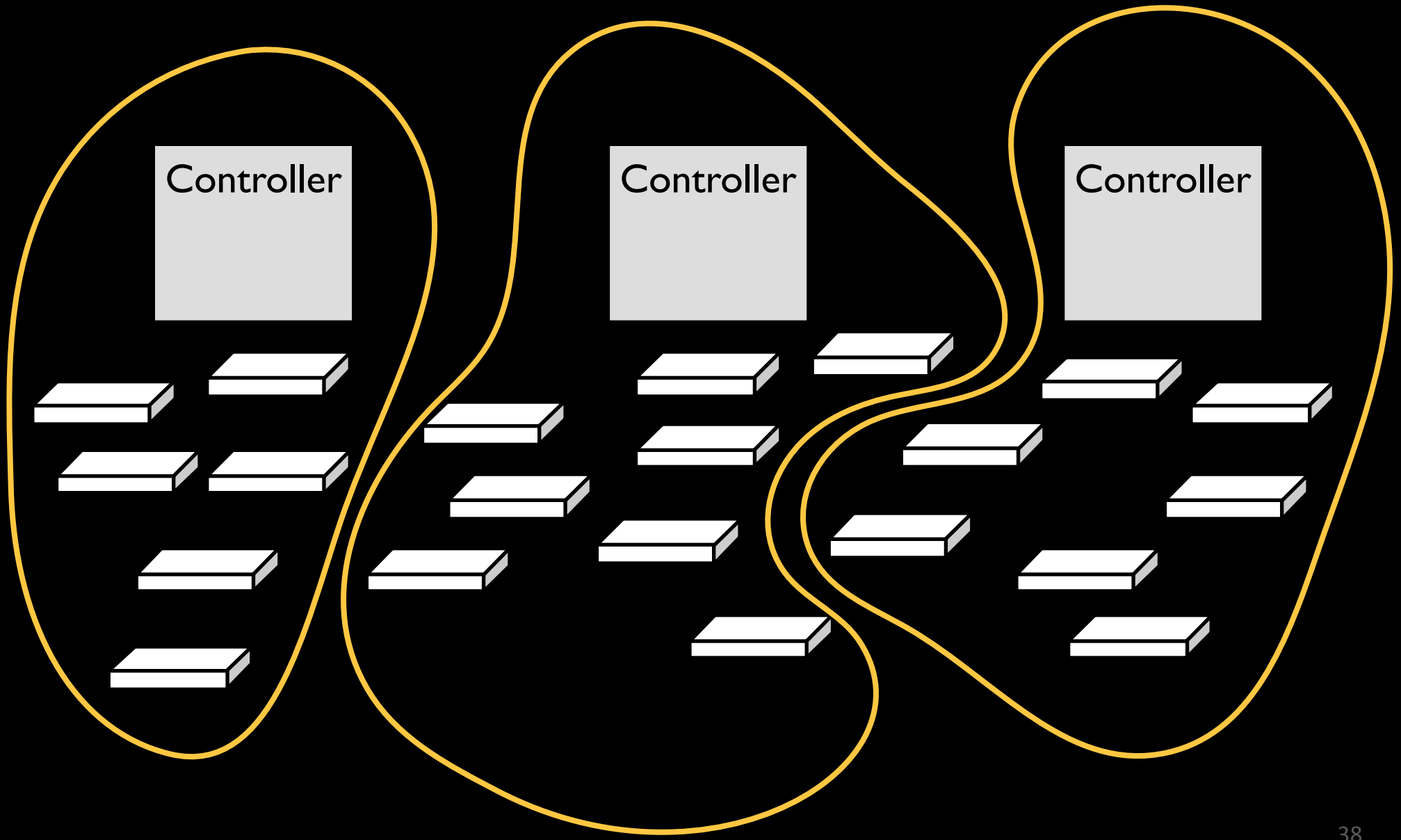
Match	Action	Priority

Match	Action	Priority

Two Techniques

- Two-phase execution
  - Software program executes transactionally
  - When software completes, system compiles set of commands, pushes them transitionally to switches
- Sharded state architecture
  - Each controller in a large system is responsible for a subset of the network state
  - Distributed transactions across multiple controllers can make large changes, but small local changes only require one controller

Sharded Architecture



Distributed Transactions

- A program can issue a remote procedure call to start a transaction on another controller
- Transaction operates identically to single-controller, case with one exception: completion
  - When transaction completes, waits for "commit" message from its invoker
- When the root transaction completes, it issues commit messages to its sub-transactions

Sharding Tradeoffs

- Parallelism: programs touching one controller can run in parallel with no coordination
  - Transactions on opposite coasts do not need cross-country RTT
- Scale-out: network size and complexity not constrained by RAM or database size
- Large, global operations involve large transactions across many controllers
  - This is true regardless — it is a global operation!

Outline

- Bugs in SDN controllers today
- Transaction defined networks
- Sergeant, a TDN system
- New network applications
- Conclusion

SERGEANT

- A transaction defined network implementation
- Written in Java, 17,000 lines of code
- Programs written in a Javascript-like language with support for transactions
- Implements two phase execution, distributed transactions, full ACID semantics
- Two implementation details
  - Speculative execution to improve throughput
  - Distributed scheduling algorithm

Speculative Execution

- Hardware switches can be *slow*
- Rejections occur, but are rare
- Idea: assume second phase will succeed, allow transactions to speculatively execute on the not-yet committed software state
  - If speculated-on transaction aborts, roll back dependent transactions
  - Allows first phase of new transactions to make progress while second phase of transaction completes

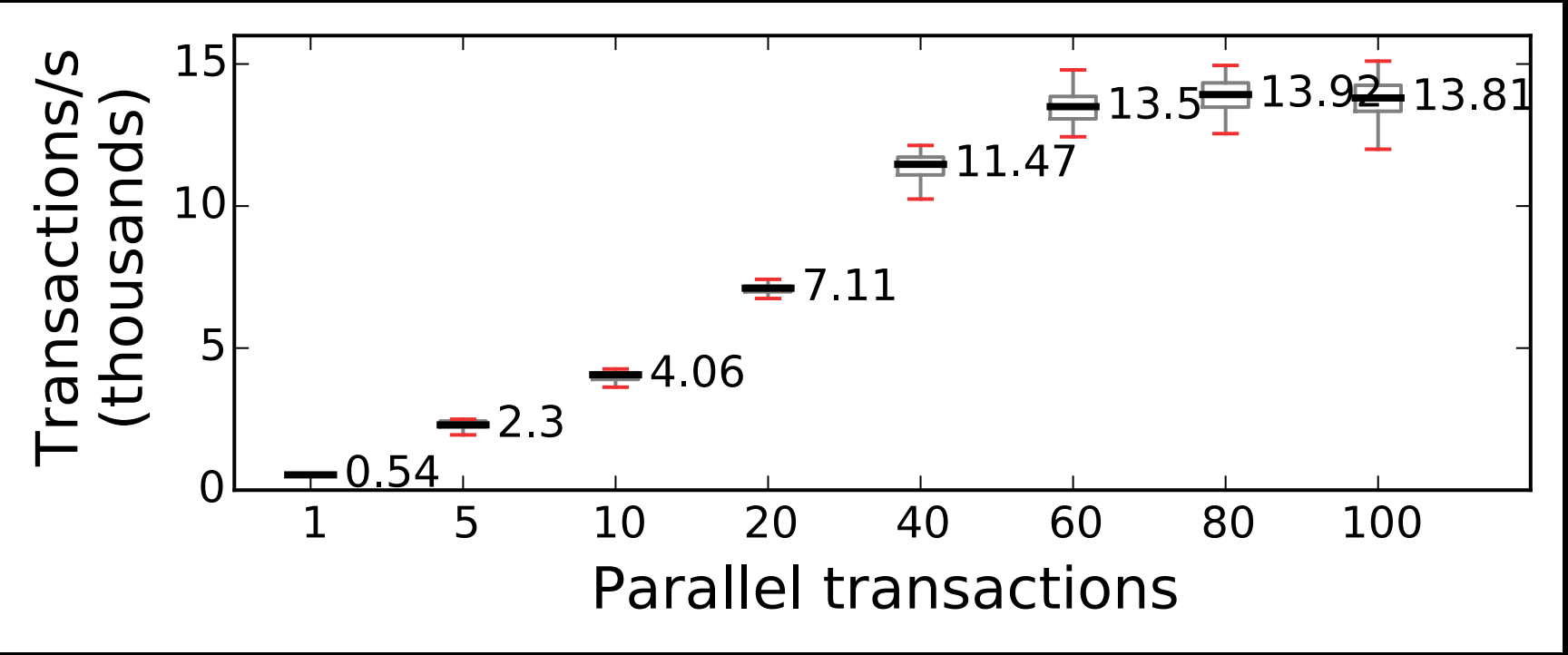
Distributed Scheduling

- Large networks will have many administrators and people issuing transactions
- Sergeant needs to schedule these transactions
  - Starvation-free: every transaction will eventually run
  - Deadlock-free: system will not hang
  - Scales out: no global state/coordination

Sergeant Evaluation

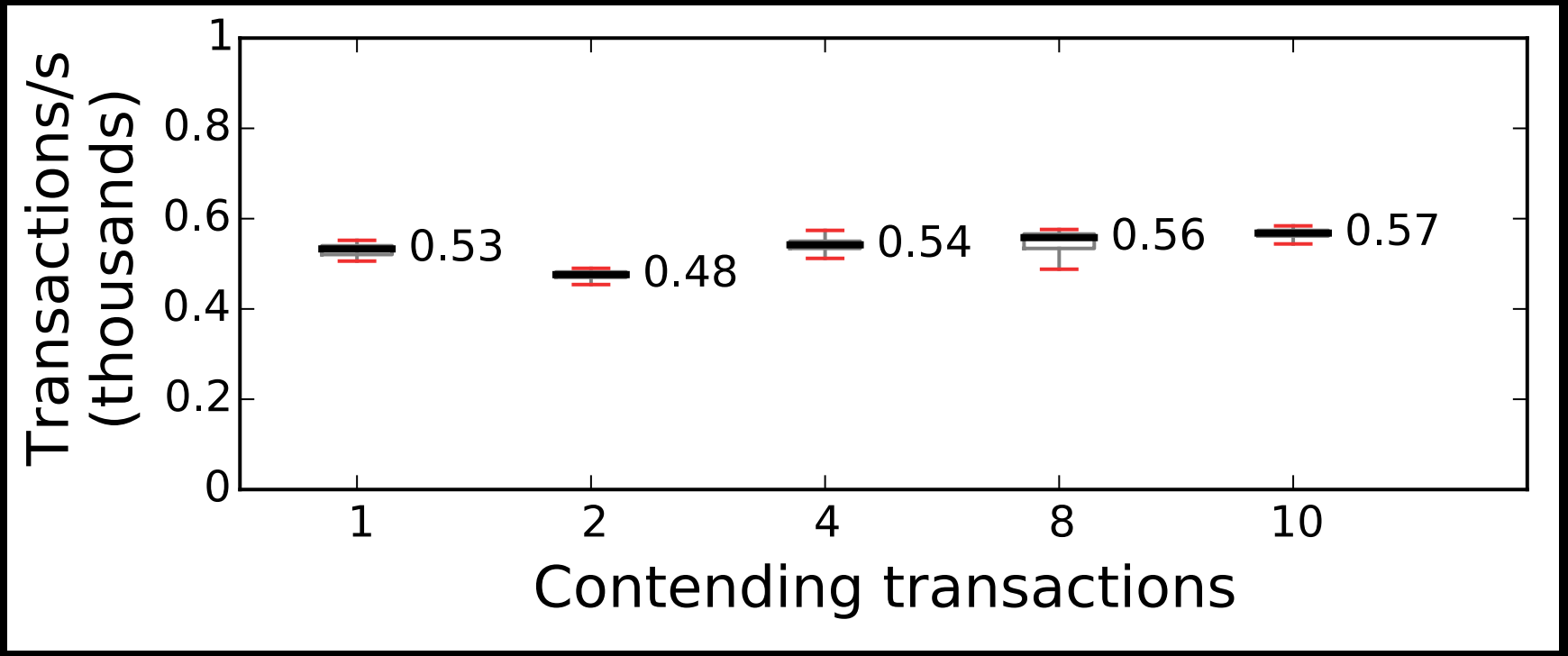
- Amazon EC2, c3.4x large instances
  - 16 vCPUs, 30GB memory
  - Single controller: SSD, provisioned to 15k IO ops/sec
  - Multi controller: SSD, provisioned to 6k IO ops/sec
- Mininet, OpenVSwitch 2.0.2
- Hardware switch: Pronto 3290, connected to 3.4GHz quad-core, 16GB RAM, 10G Ethernet

Single Controller



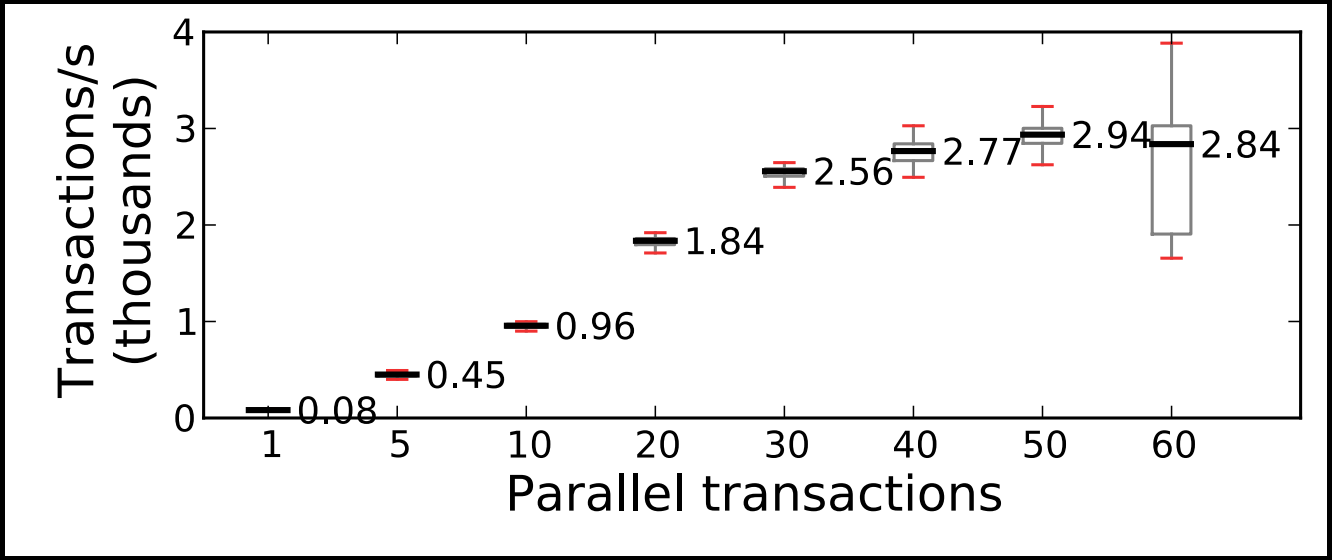
- A single controller can execute 13,000 parallel transactions/second
- Limited by SSD/durability

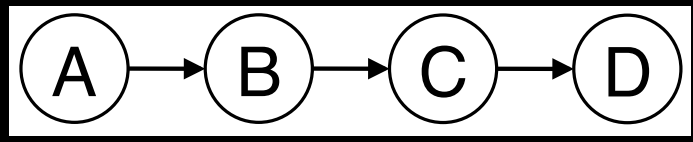
Single Switch



- A single switch can handle 500 transactions/ second
- Limited by I/O latency for durable logs

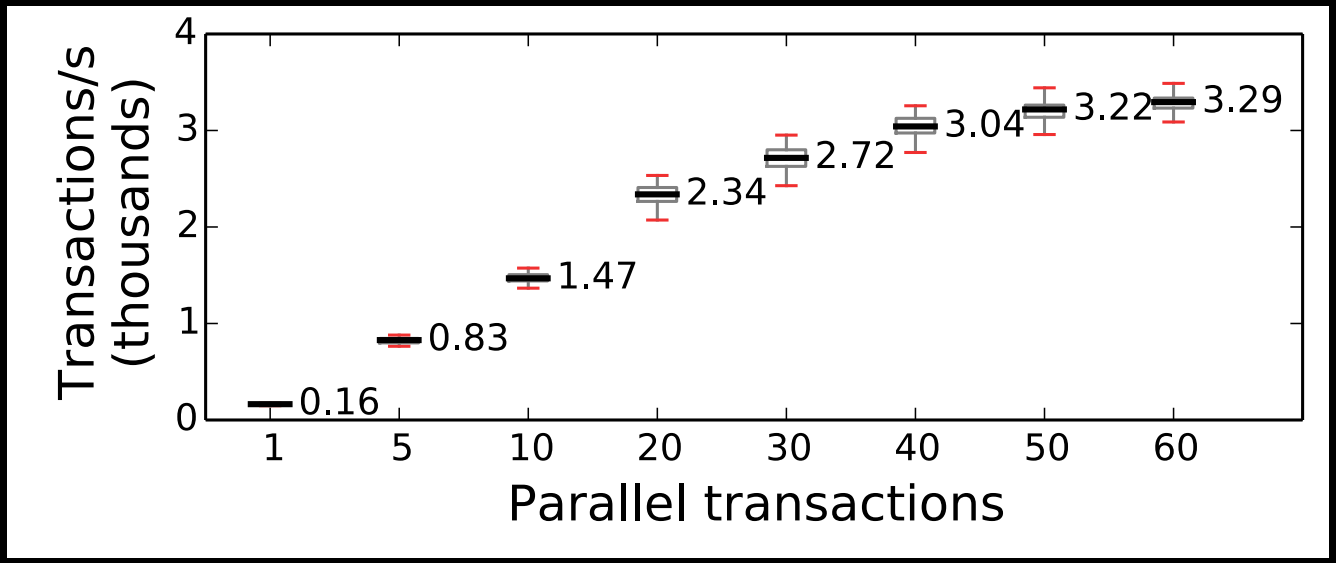
Distributed Controllers: Line

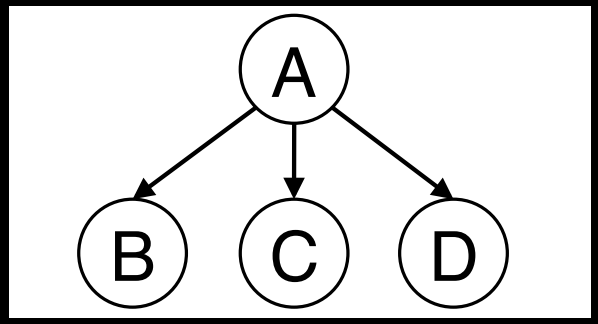




- 2,800 transactions/second across 4 controllers
- Limited by latency of two-phase commit

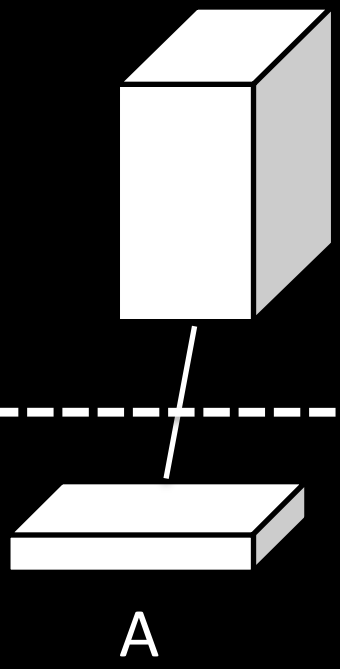
Distributed Controllers: Tree





- 3,000 transactions/second across 4 controllers
- Limited by latency of two-phase commit
- Faster than line due to parallel commit

Read-only throughput



App 1: size(A's ftable)
App 2: size(A's ftable)
App 3: size(A's ftable)
…

> 100k programs/s

TDN Overview

- Transactions solve persistent bugs and errors in software defined network controllers
- Sergeant demonstrates that transaction defined networks can be high performance
  - 13,000 transactions/second on a single controller
  - 500 transactions/second on a single switch
  - > 2,800 transactions/second in distributed controller networks
  - Transactions are scheduled fairly, deadlock free, do not starve

Outline

- Bugs in SDN controllers today
- Transaction defined networks
- Sergeant, a TDN system
- New network applications
- Conclusion

*"The problem is always knowing what the network is doing so I can find the issue… that can take **hours**, but after finding the problem I can fix it in **15 seconds**"*

- Stanford network admin

What if…

- You could ask questions about your network?
  - "When did node A stop being able to connect to B?"
  - "Who made that change?"
  - "Why did that change happen?"
  - "What would happen if I made this change?"
- Treat transactions on network like commits to a software repository: network version control

tdndb: TDN debugger

- Uses mininet to simulate state of a network
- Can step through network history at transaction granularity (forward and back)
  - Can probe state of network (e.g., ping, connect, etc.)
- Automated features to find points of interest
  - "When did X happen?", "Who made that change?"
- Replay program execution
  - "Why did the program do this?"
- Takes 20s to replay 100k transactions across 100 switches

Outline

- Bugs in SDN controllers today
- Transaction defined networks
- Sergeant, a TDN system
- New network applications
- Conclusion

SDN Guarantees

System	Atomicity	Consistency	Isolation	Durability
Floodlight	No	No	No	No
ONOS	No	No	No	No
Pyretic	No	No	No	No
Onix	No	No	Yes	Yes
TDN	Yes	Yes	Yes	Yes

Floodlight, ONOS, Pyretic, Sergeant results from fuzz testing
Onix results from discussion with authors

Transactions

- OpenFlow gives switches tremendous latitude in program execution
  - Imagine a processor that could arbitrarily re-order instructions and reject any instruction
- TDN provides an intermediate abstraction that hides complexity without sacrificing power
  - Complexity: execution model; power: flow tables
  - Analogy: C provides a stack but otherwise provides explicit memory management and layout
- Other systems should build on top of TDN

Why Do These Problems Occur?

- Mininet has enabled fast, rapid experimentation and prototyping of SDN systems
- Mininet does not re-order or reject commands
  - Systems tested on Mininet can fail on real switches

- Short term patch: developing a fuzzer extension for Mininet
- Long term solution: use a transactional execution model

Long Term Implications

- The Internet was originally a loose confederation of separate parties
  - Distributed algorithms, eventual consistency
  - But switches have similar performance to old disks
- This ethos and mentality has continued into networks today… but it doesn't have to
  - Single administrative domains
  - Networks are many orders of magnitude faster
  - We much better understand how to build highly reliable, highly available, high performance systems

Questions

(S++)DN = TDN