Towards Distributed and Reliable Software Defined Networking

Marco Canini (TU Berlin & T-Labs & UCL)
Petr Kuznetsov (TU Berlin & TU Berlin & Paris Tech)
Dan Levin (TU Berlin)
Stefan Schmid (TU Berlin & T-Labs)



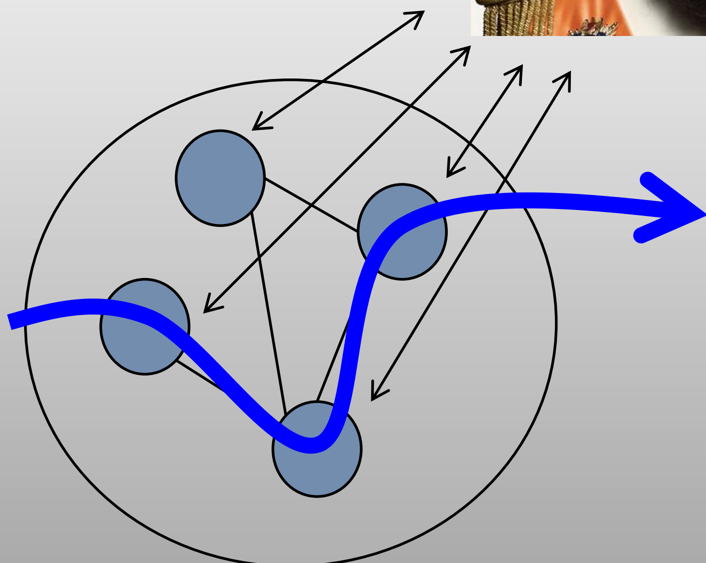

Towards Distributed and Reliable Software Defined Networking

The Case for Software Transactional Networking?

Marco Canini (TU Berlin & T-Labs & UCL)
Petr Kuznetsov (TU Berlin & TU Berlin & Paris Tech)
Dan Levin (TU Berlin)
Stefan Schmid (TU Berlin & T-Labs)



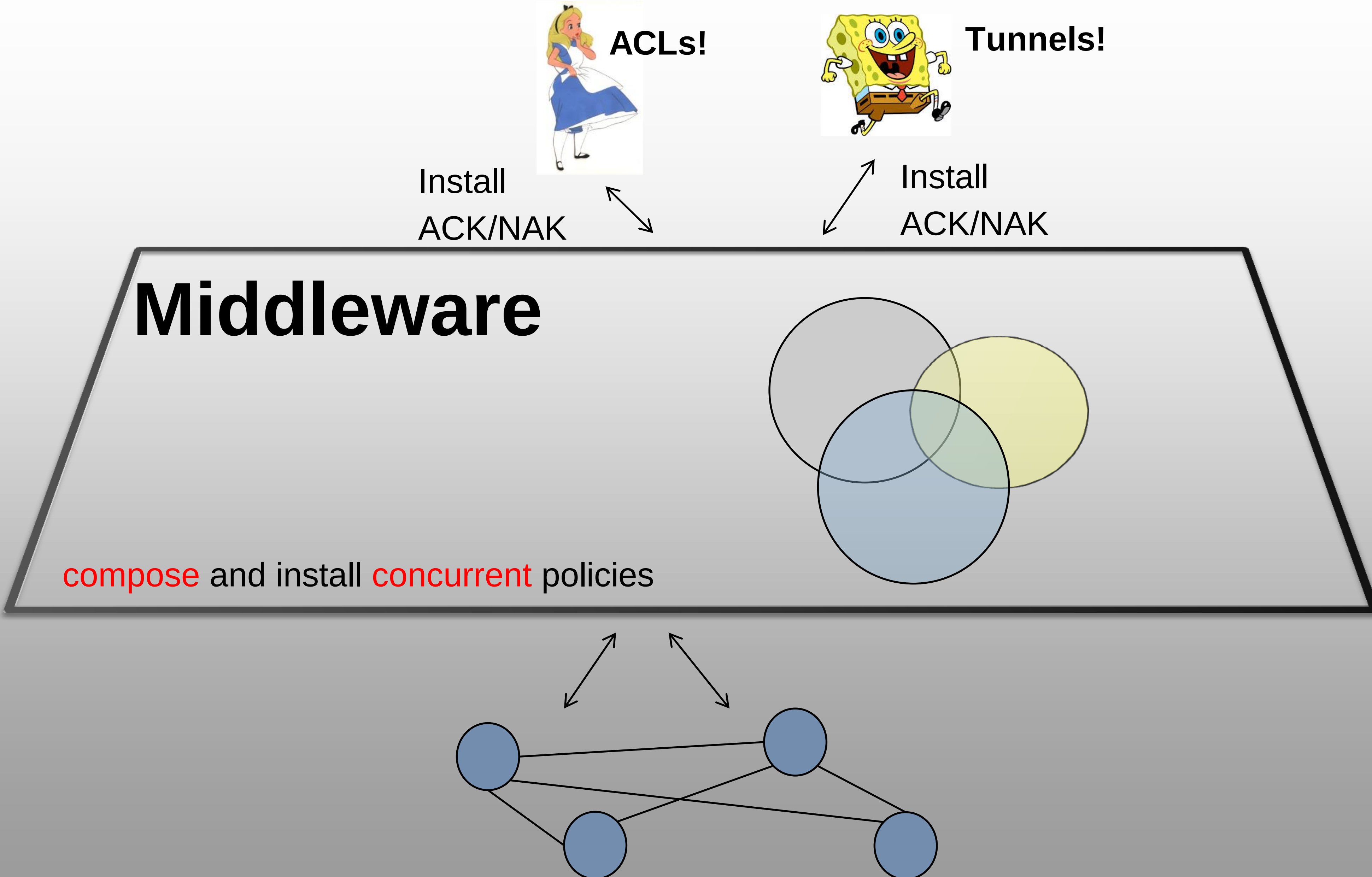

The 1-Slide SDN Lecture



SDN

- Control of (forwarding) rules in network from simple, logically centralized vantage point
- Flow concept: install rules to define flow
- Match-Action concept: apply actions to packets
- Specifies global network policies, e.g., load-balancing, adaptive monitoring / heavy hitter detection, …

Vision: Middleware for Concurrent and Robust Policy Installation



Vision: Middleware for Concurrent and Robust Policy Installation

Robust

ACLs!

Tunnels!

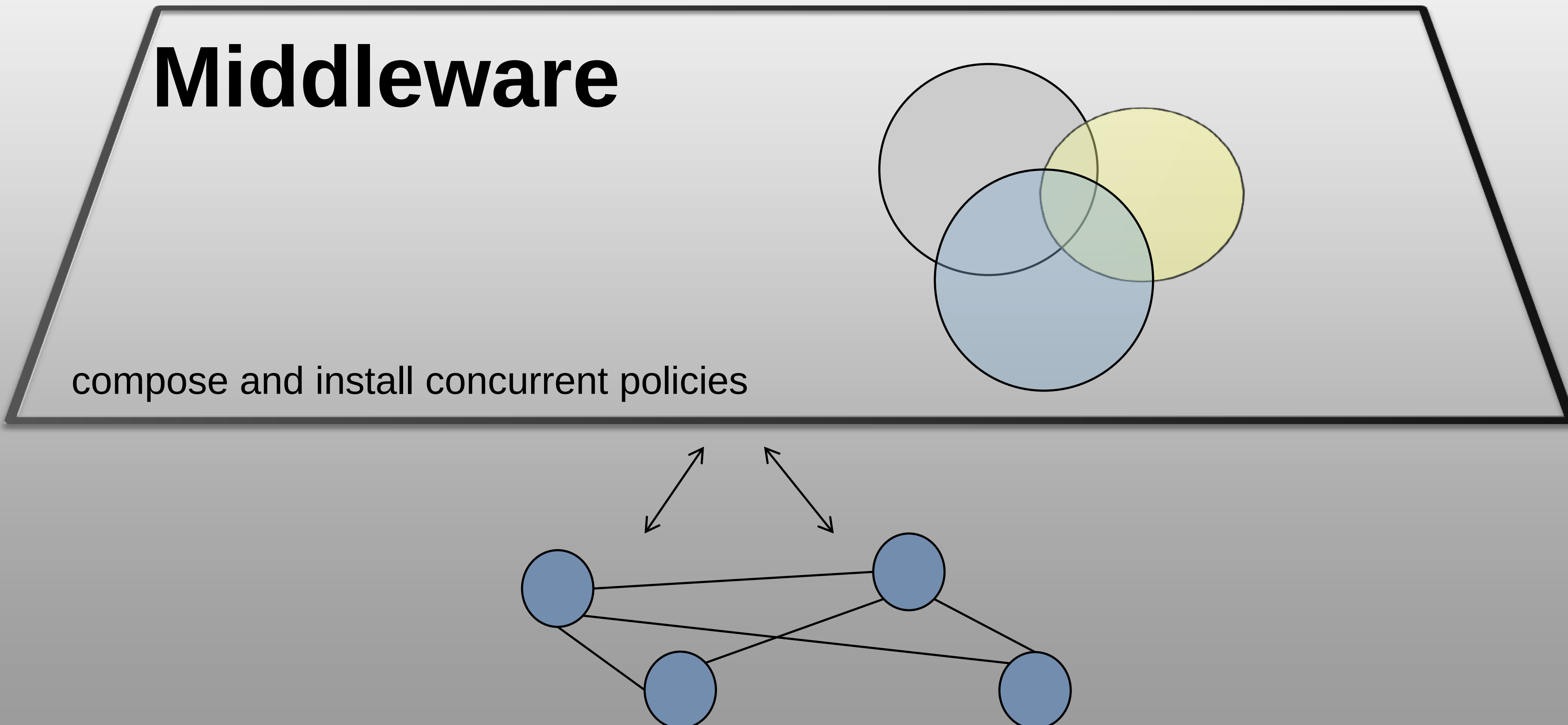
failures (fail-stop)

Install

ACK/NAK

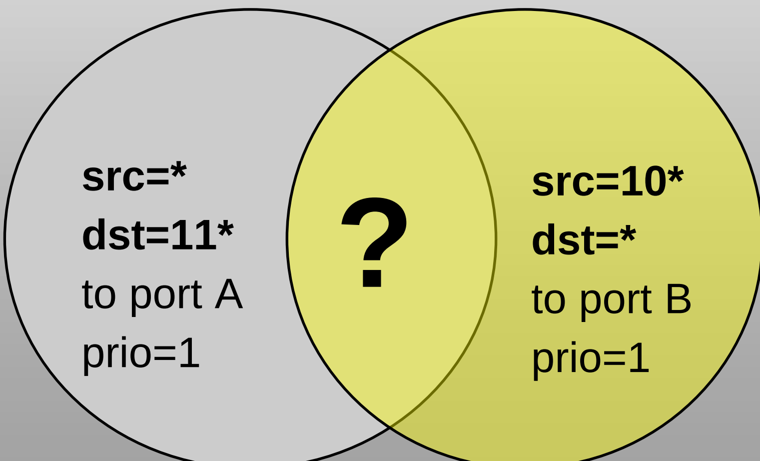
Install

ACK/NAK

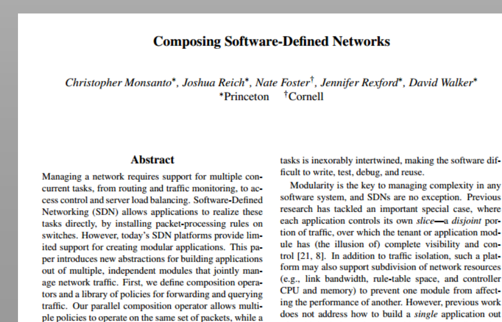


Policies and Composition

- Policy = defined over (header) domain ("flow space")
- Policy priority
- Implies rules on switch ports
- Conflict = overlapping domains, same priority, different treatment



- Policy composition = combined policy, avoids conflicts
- E.g., composition by priorities or most specific, or do both parts
- Implement exactly one policy if two conflict
- Only known central solution: need to compose, e.g., Frenetic/Pyrethic:



Composing Software-Defined Networks

Christopher Monsanto*, Joshua Reich*, Nate Foster†, Jennifer Rexford*, David Walker*
*Princeton †Cornell

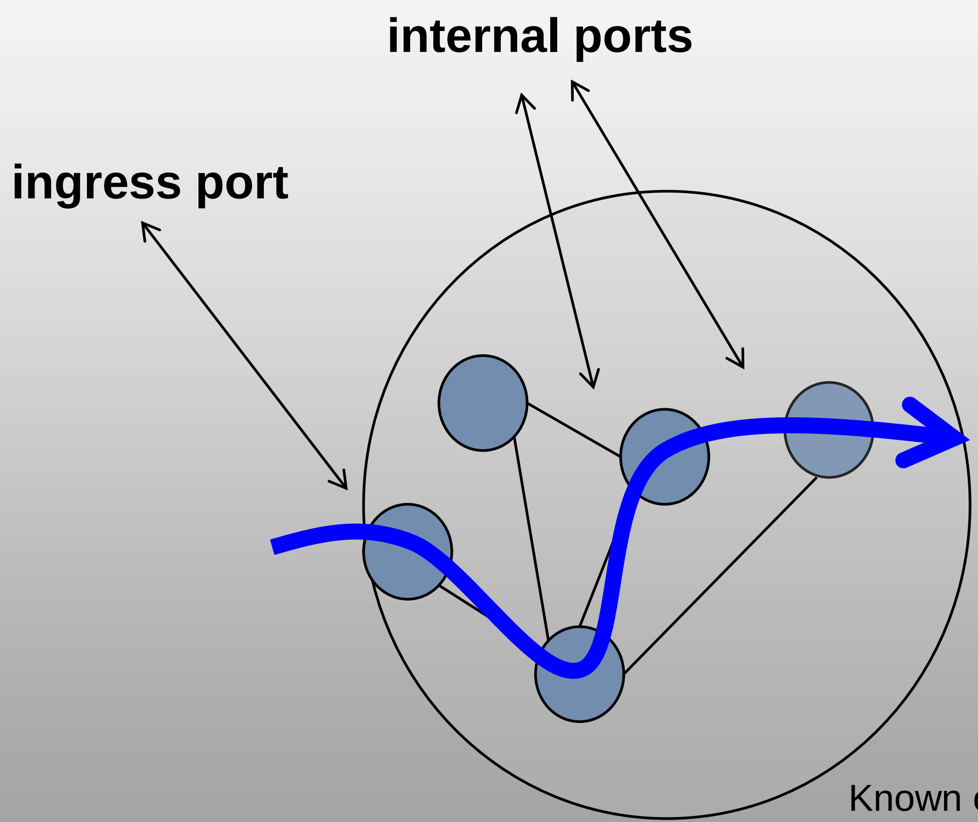
Abstract

Managing a network requires support for multiple concurrent tasks, from routing and traffic monitoring, to access control and server load balancing. Software-Defined Networking (SDN) allows applications to realize these tasks directly, by installing packet-processing rules on switches. However, today's SDN platforms provide limited support for creating modular applications. This paper introduces new abstractions for building applications out of multiple, independent modules that jointly manage network traffic. First, we define composition operators and a library of policies for forwarding and querying traffic. Our parallel composition operator allows multiple policies to operate on the same set of packets, while a novel *sequential composition operator* allows one policy

tasks is inexorably intertwined, making the software difficult to write, test, debug, and reuse.

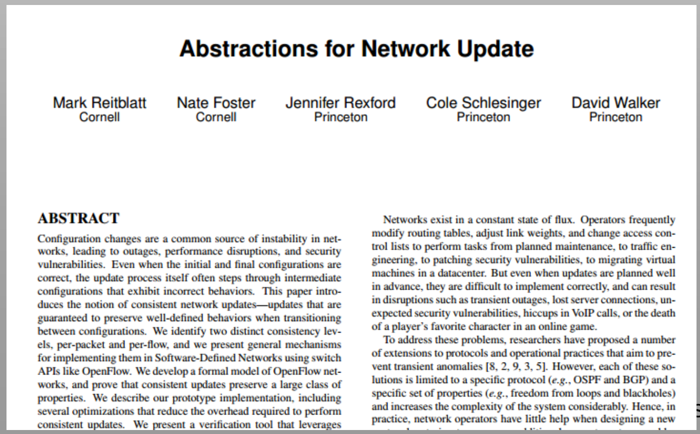
Modularity is the key to managing complexity in any software system, and SDNs are no exception. Previous research has tackled an important special case, where each application controls its own *slice*—a *disjoint* portion of traffic, over which the tenant or application module has (the illusion of) complete visibility and control [21, 8]. In addition to traffic isolation, such a platform may also support subdivision of network resources (e.g., link bandwidth, rule-table space, and controller CPU and memory) to prevent one module from affecting the performance of another. However, previous work does not address how to build a *single* application out of multiple, independent, reusable network policies that

Policy Installation



- **SDN Match-Action**
  - Match header (define flow)
  - Execute action (e.g., add tag or forward to port)
- **Consistent Update: 2-phase**
  - **At internal ports:** add new rules for new policy with new tag
  - **Then at ingress ports:** start tagging packets with new tag

Known central solution (our model):



Abstractions for Network Update

Mark Reitblatt, Cornell
Nate Foster, Cornell
Jennifer Rexford, Princeton
Cole Schlesinger, Princeton
David Walker, Princeton

ABSTRACT

Configuration changes are a common source of instability in networks, leading to outages, performance disruptions, and security vulnerabilities. Even when the initial and final configurations are correct, the update process itself often steps through intermediate configurations that exhibit incorrect behaviors. This paper introduces the notion of consistent network updates—updates that are guaranteed to preserve well-defined behaviors when transitioning between configurations. We identify two distinct consistency levels, per-packet and per-flow, and we present general mechanisms for implementing them in Software-Defined Networks using switch APIs like OpenFlow. We develop a formal model of OpenFlow networks, and prove that consistent updates preserve a large class of properties. We describe our prototype implementation, including several optimizations that reduce the overhead required to perform consistent updates. We present a verification tool that leverages

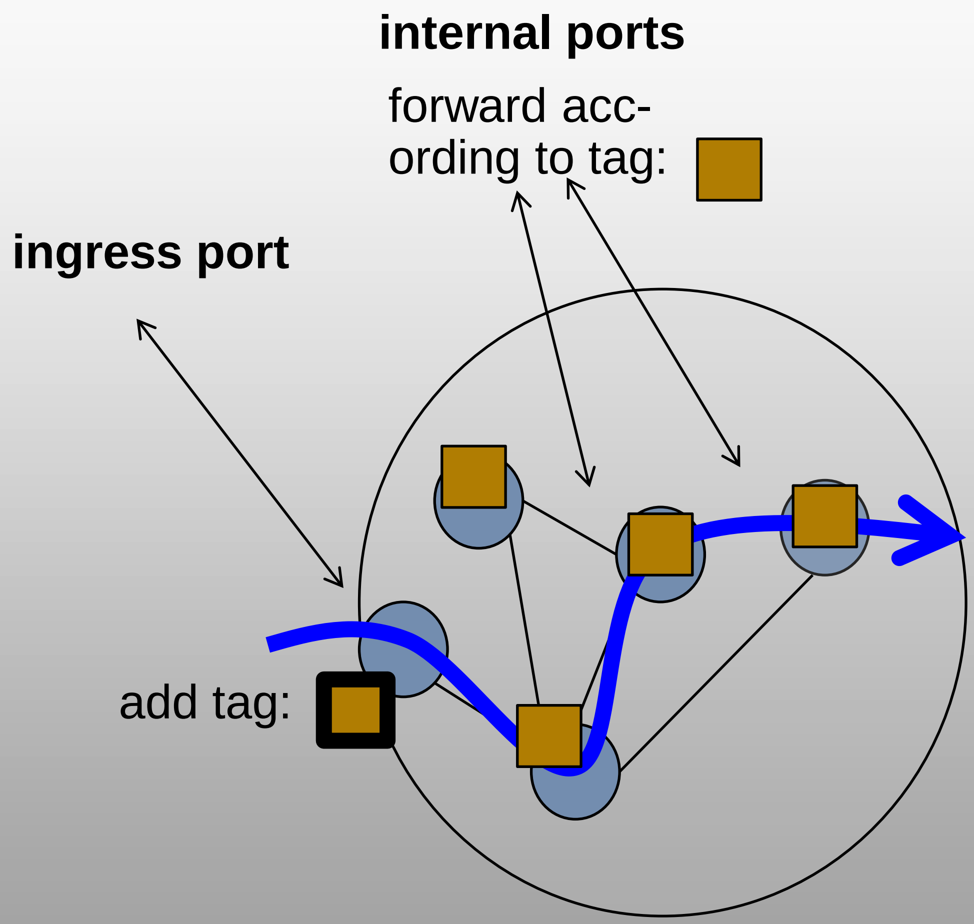
Networks exist in a constant state of flux. Operators frequently modify routing tables, adjust link weights, and change access control lists to perform tasks from planned maintenance, to traffic engineering, to patching security vulnerabilities, to migrating virtual machines in a datacenter. But even when updates are planned well in advance, they are difficult to implement correctly, and can result in disruptions such as transient outages, lost server connections, unexpected security vulnerabilities, hiccups in VoIP calls, or the death of a player's favorite character in an online game.

To address these problems, researchers have proposed a number of extensions to protocols and operational practices that aim to prevent transient anomalies [8, 2, 9, 3, 5]. However, each of these solutions is limited to a specific protocol (*e.g.*, OSPF and BGP) and a specific set of properties (*e.g.*, freedom from loops and blackholes) and increases the complexity of the system considerably. Hence, in practice, network operators have little help when designing a new

s)

Policy Installation

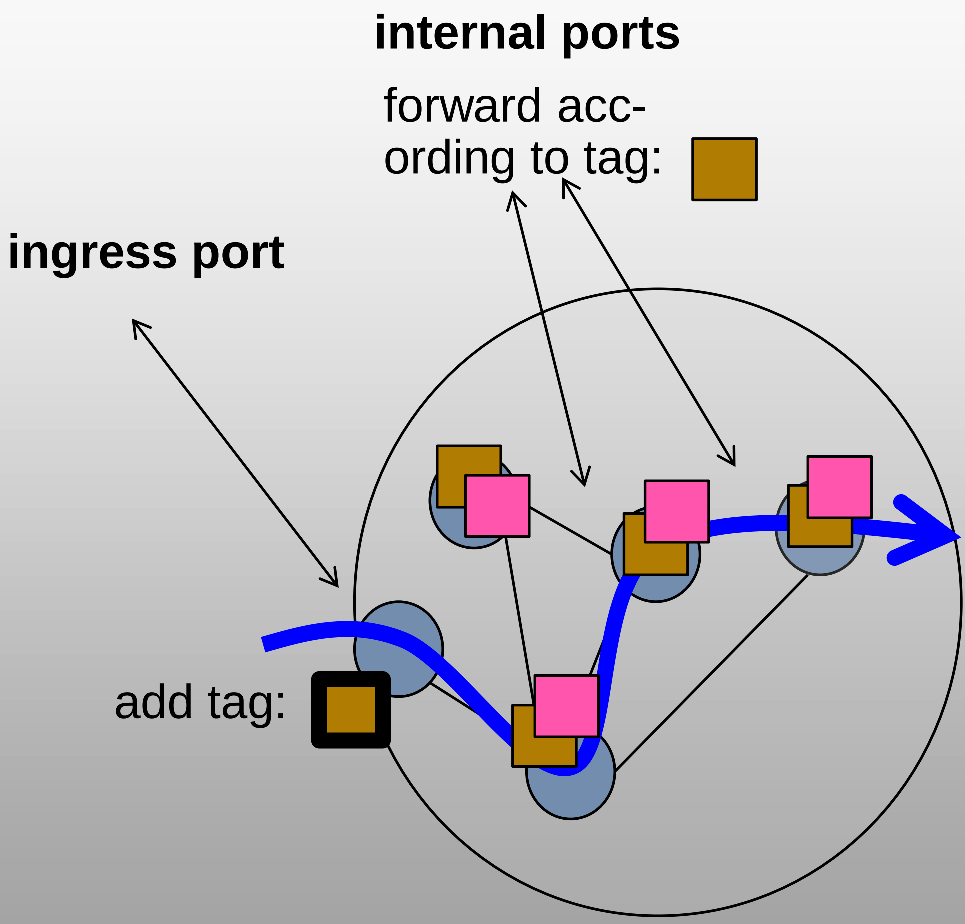
Initially



- **SDN Match-Action**
  - Match header (define flow)
  - Execute action (e.g., add tag or forward to port)
- **Consistent Update:** 2-phase
  - **At internal ports:** add new rules for new policy with new tag
  - **Then at ingress ports:** start tagging packets with new tag

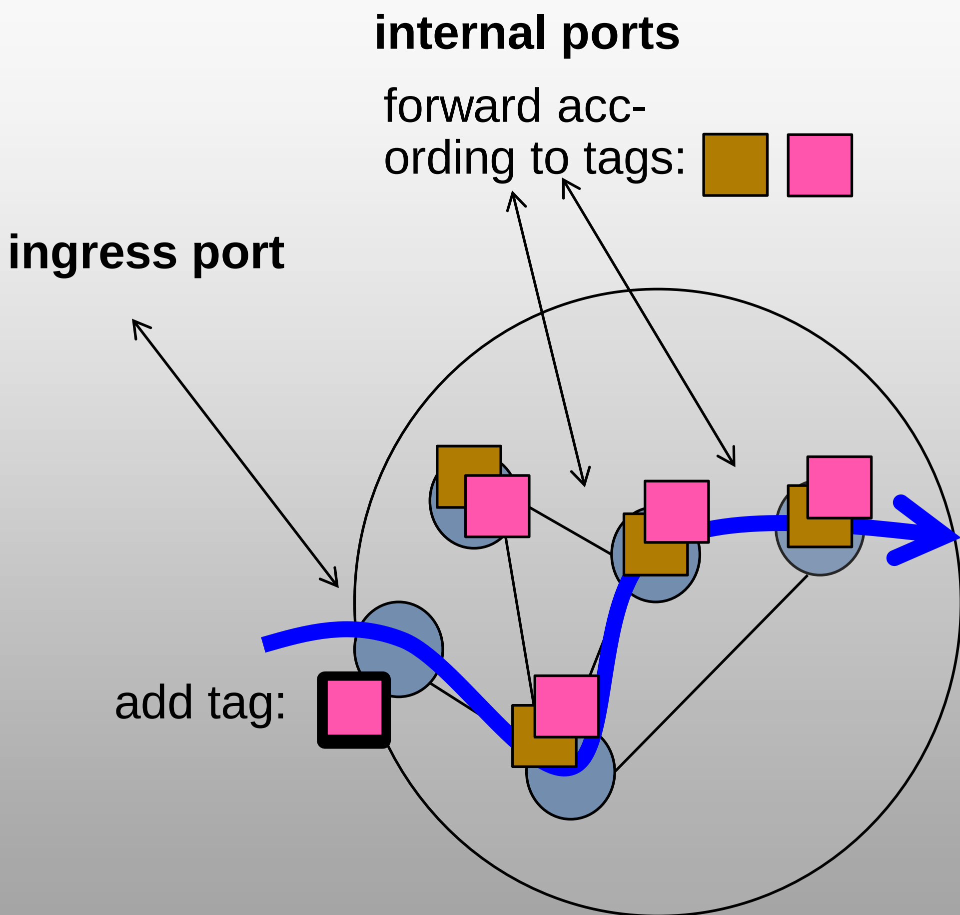
Policy Installation

Phase 1



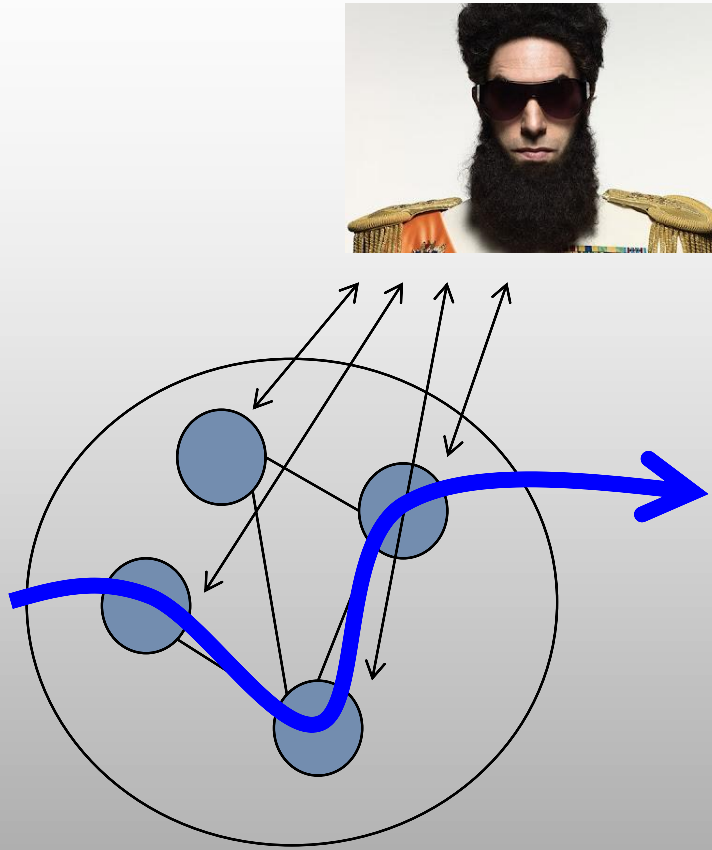
- **SDN Match-Action**
  - Match header (define flow)
  - Execute action (e.g., add tag or forward to port)
- **Consistent Update:** 2-phase
  - **At internal ports:** add new rules for new policy with new tag
  - **Then at ingress ports:** start tagging packets with new tag

Phase 2

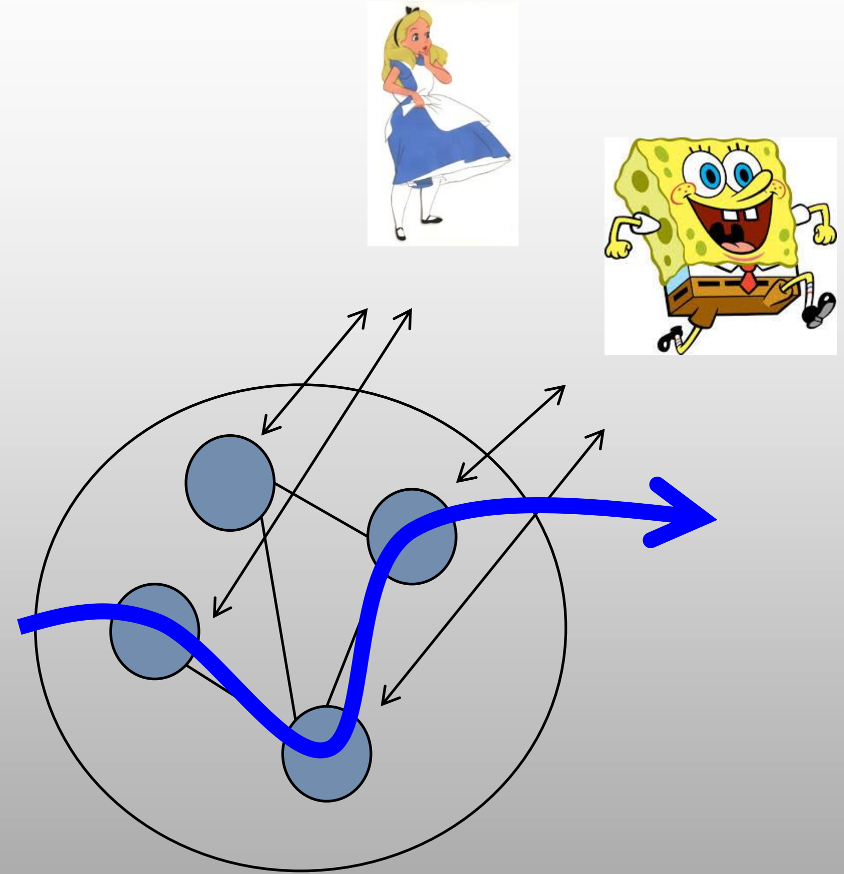


- **SDN Match-Action**
  - Match header (define flow)
  - Execute action (e.g., add tag or forward to port)
- **Consistent Update:** 2-phase
  - **At internal ports:** add new rules for new policy with new tag
  - **Then at ingress ports:** start tagging packets with new tag

But what about distributed and multi-author policies?

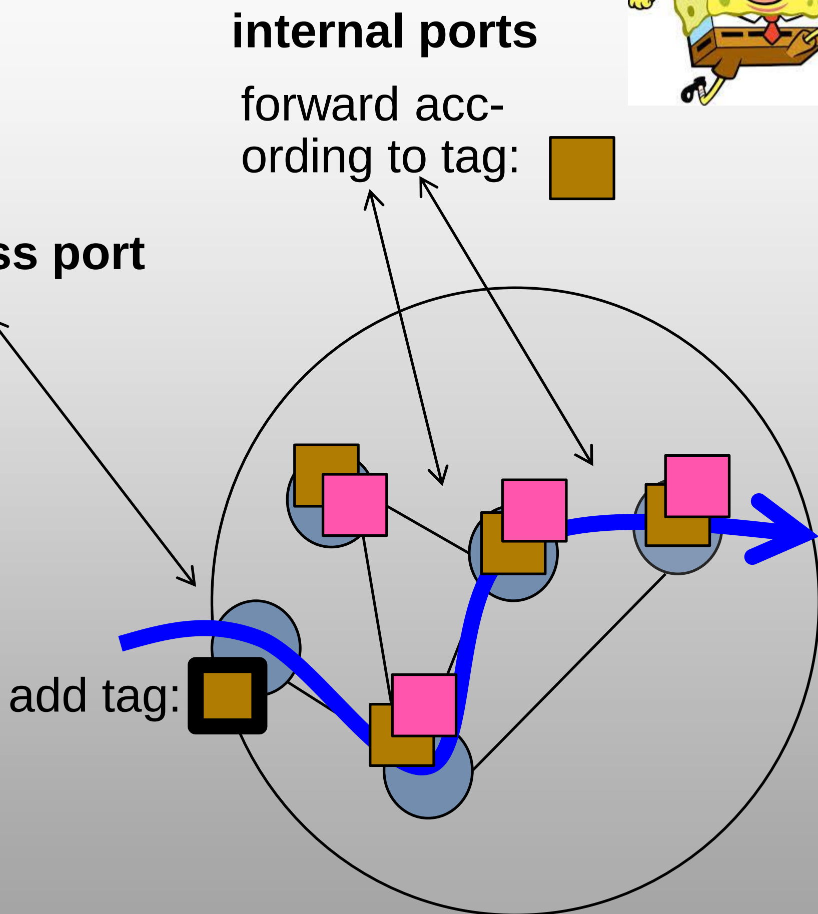


VS



One guy in charge of setting up tunnels,
one guy in charge of ACLs, …

Idea: Distributed Version



Synchronize:

- Do not override conflicting policies
- Especially ingress port(s)

Share Tags:

- Agree on tags

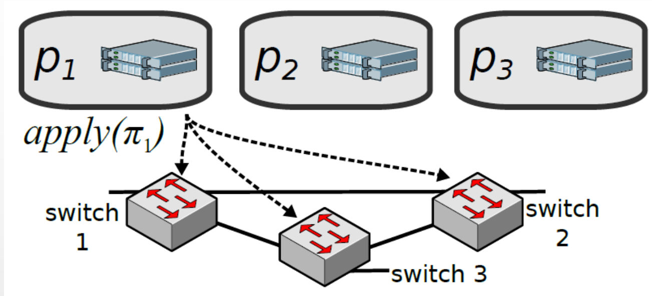
Problem Statement

Goals

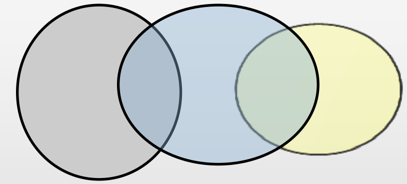
- **All-or-nothing:** policy fully installed or not at all
- **Conflict-free:** never two conflicting policies
- **Progress:** non-conflicting policy eventually installed; and: at least one conflicting policy
- **Per-packet consistency:** per packet only one policy applied (during journey through network)
- Always rules ready when packets arrive (not under control!)

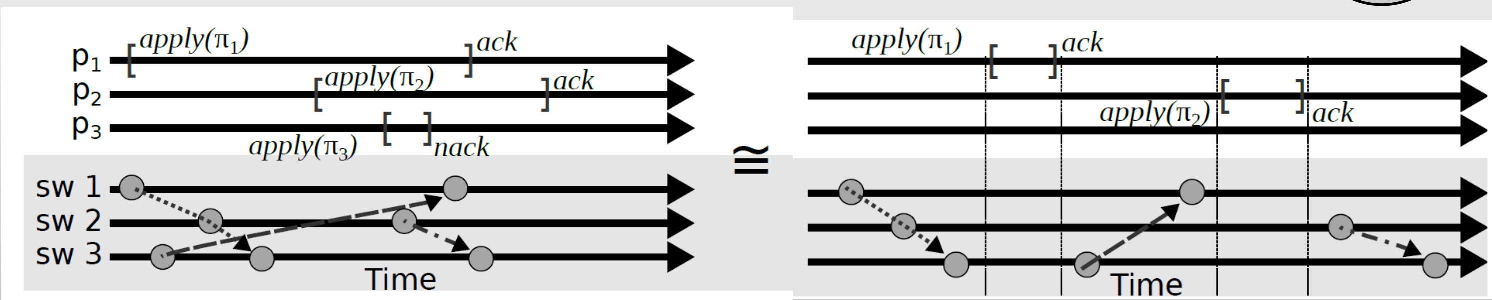
Goal: Serializable!

Example



Three switches, three policies, policy 1 and 2 with independent flow space, policy 3 conflicting:





Control Plane

Packet Traces

Left: Concurrent history: 3rd policy aborted due to conflict.

Right: In the sequential history, no two requests applied concurrently. No packet is in flight while an update is being installed.

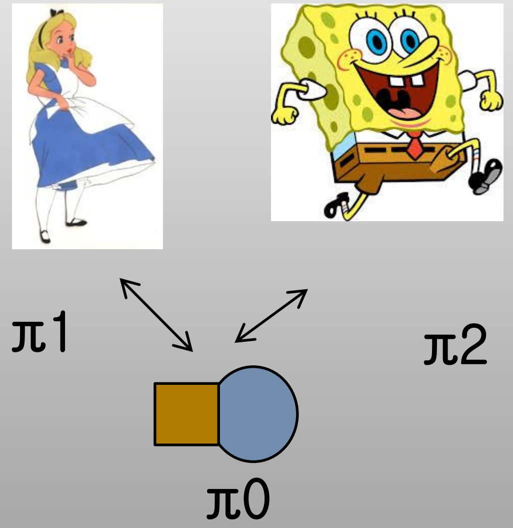
No packet can distinguish the two histories! So as though the application of policy updates is atomic and packets cross the network instantaneously.

Bad News: Impossible Without Atomic Read-Modify-Write Ports

Thm: Without atomic rmw-ports, per-packet consistent network update is impossible if a controller may crash-fail.

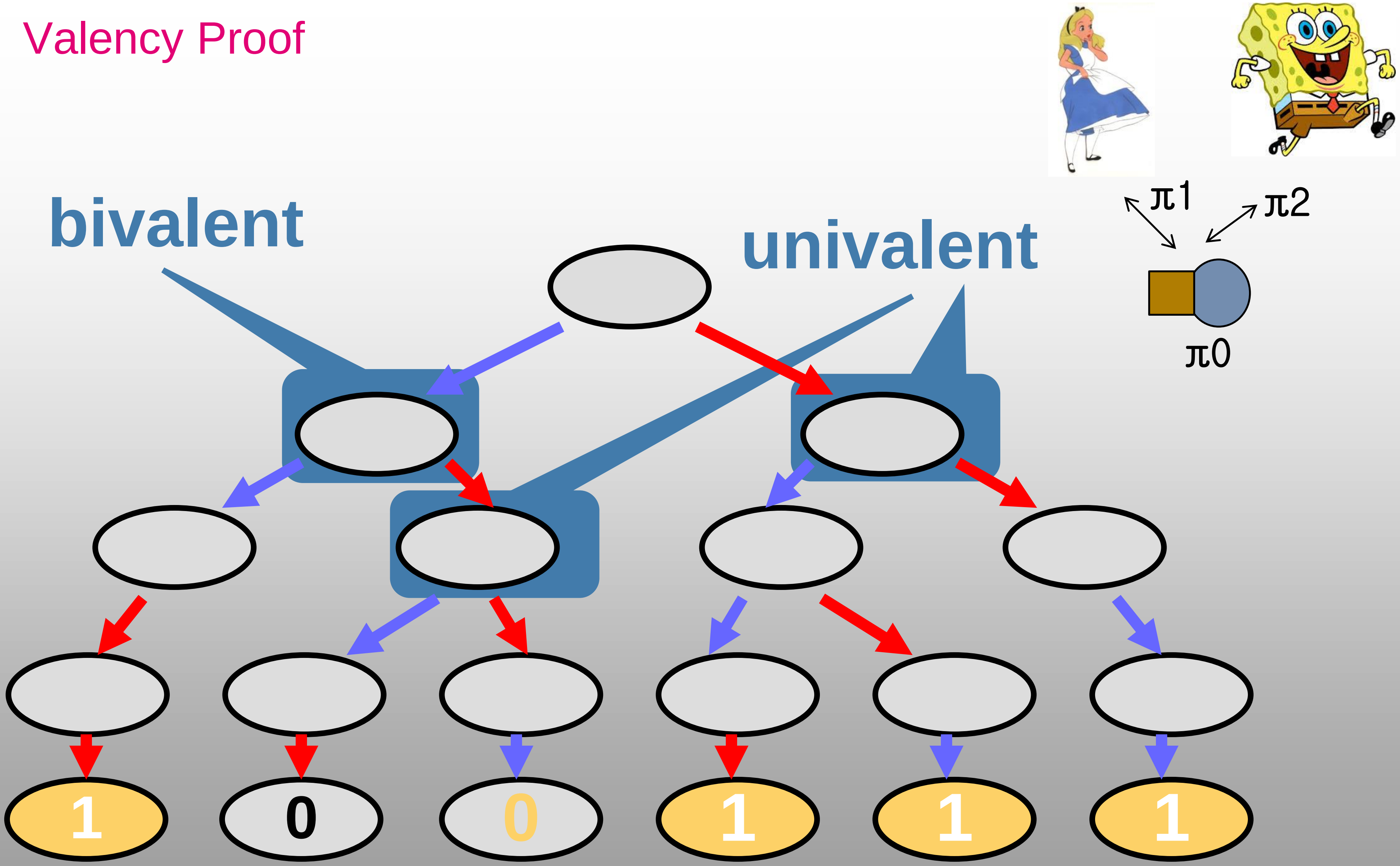
Proof:

- Single port already!
- π1 and π2 are conflicting
- Descendant of state σ is extension of execution of σ.
- State σ is i-valent if all descendants of σ are processed according to πi. Otherwise it is undecided.
- Initial state is undecided, and in undecided state nobody can commit its request and at least one process cannot abort its request.
- There must exist a critical undecided state after which it’s univalent if a process not longer proceeds.
- Difference cannot be observed: overriding violates consistency (sequential composition).



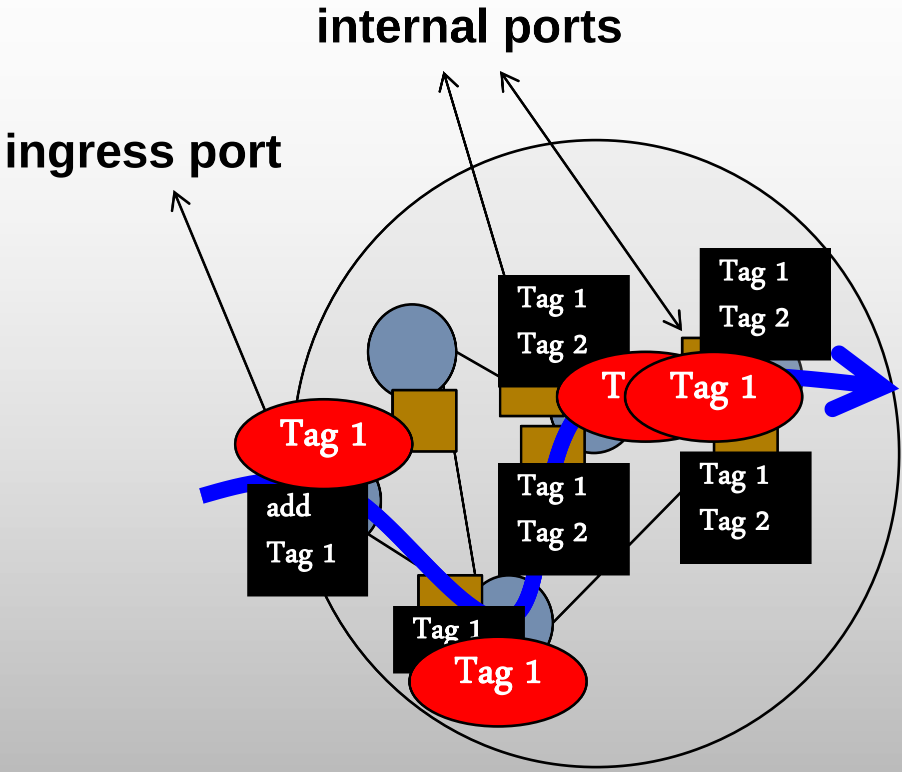
QED

Valency Proof



Good News: Middleware for Concurrent Policy Updates

Thm: With atomic RMW, the TAG algorithm is correct and wait-free (up to n-1 failures).



- Principles:
  - (1) Unique tag per policy
  - (2) Install at internal ports first (compose if necessary*)
  - (3) Once installed at internal ports…
  - (4) … add tag to all packets at ingress port(s)!

- Observations:
  - Rule always ready internally (2)
  - Per-packet consistency solved (4): packet never changes tag!
  - Wait-free policy installation!

* requires atomic read-modify-write

Conclusion

- Concurrent SDN policy updates: A case for "Software Transactional Networking"?
- Concurrent control not possible under atomic r/w, but possible under atomic r+w
- Future work: reduce tag size