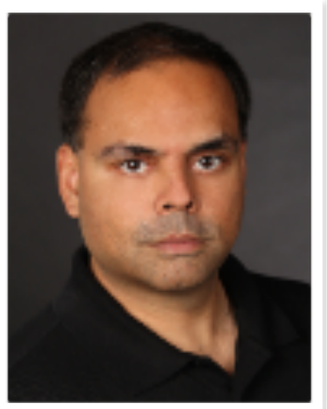
Achieving High Utilization with Software-Driven WAN

Chi-Yao Hong (UIUC)

Srikanth Kandula

Ratul Mahajan

Ming Zhang



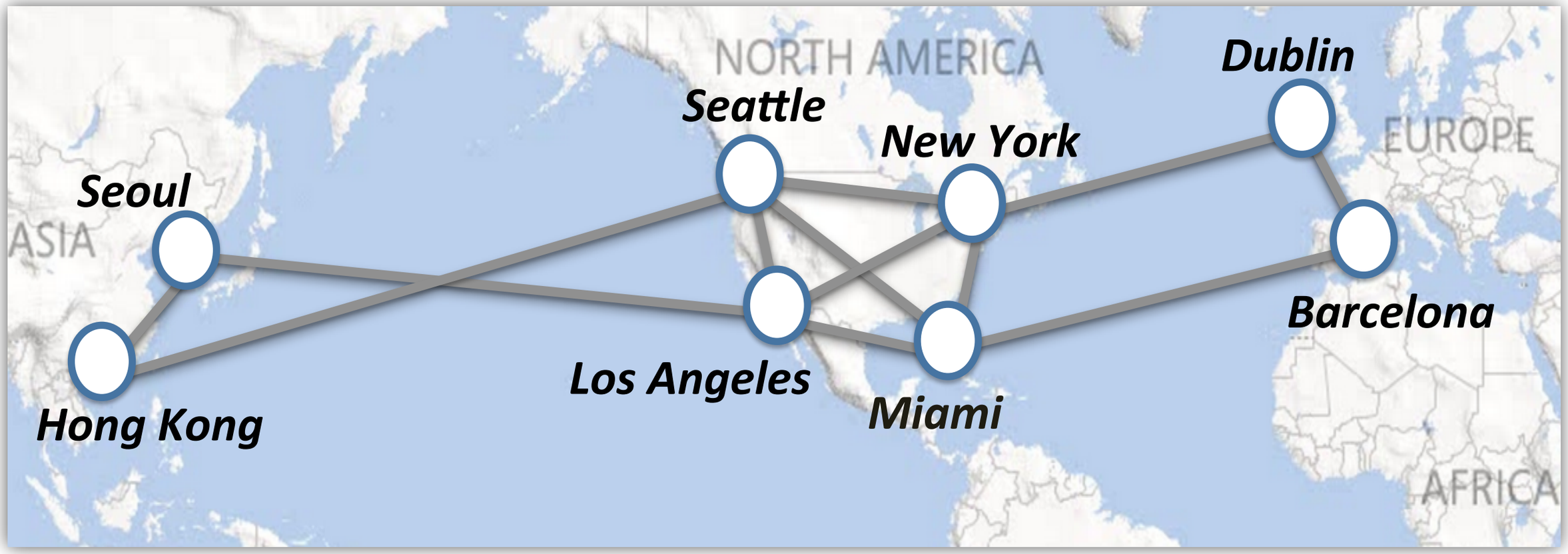
Vijay Gill

Mohan Nanduri

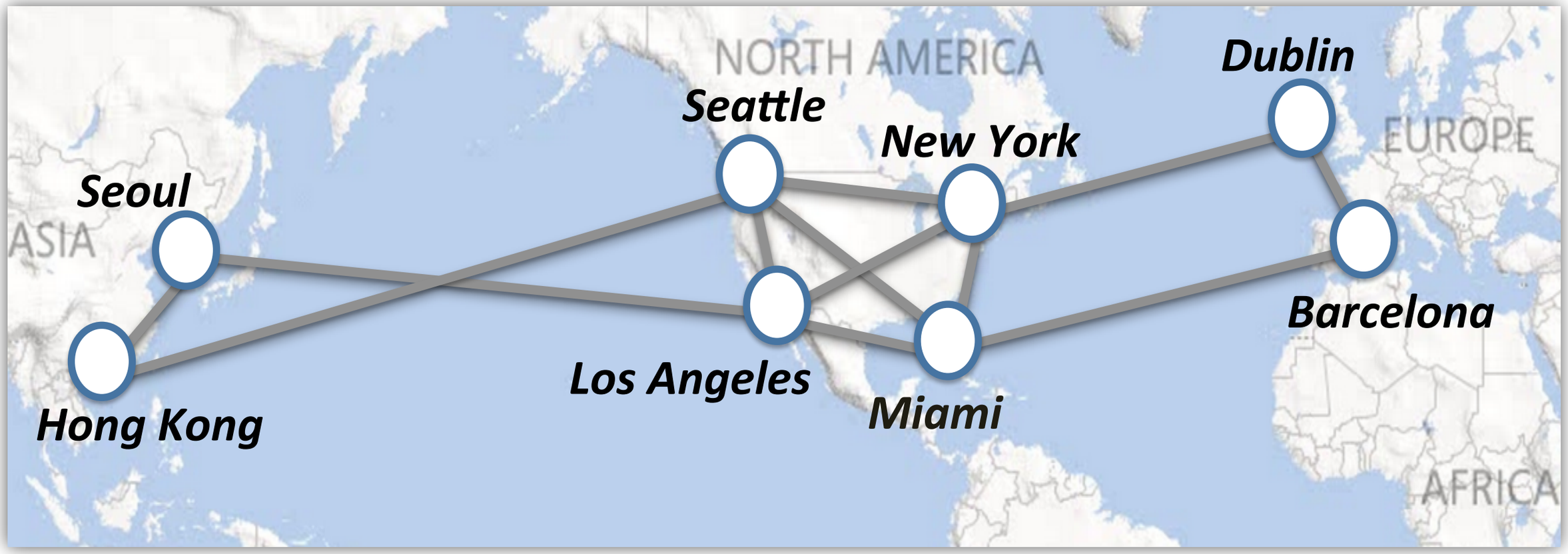
Roger Wattenhofer

Microsoft

Background: Inter-DC WANs

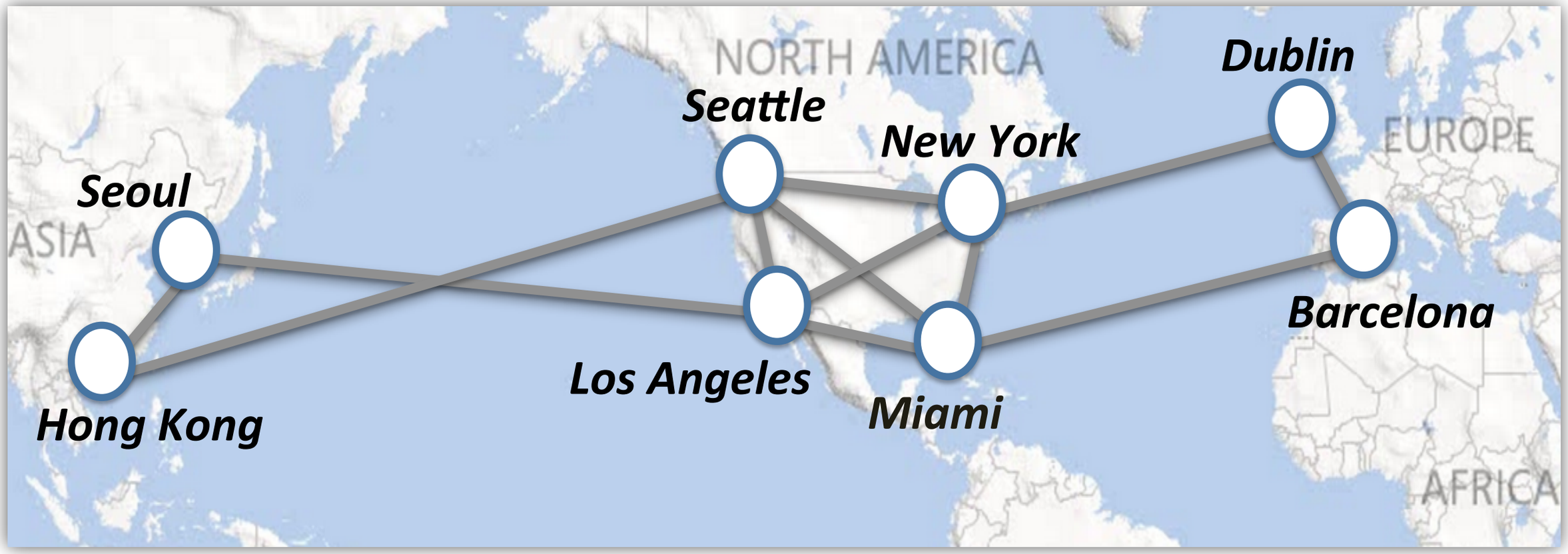


Background: Inter-DC WANs



Inter-DC WANs are critical

Background: Inter-DC WANs



Inter-DC WANs are critical

Inter-DC WANs are highly expensive

Two key problems

Poor efficiency

average utilization over time of busy links is only 30-50%

Poor sharing

little support for flexible resource sharing

Two key problems

Poor efficiency

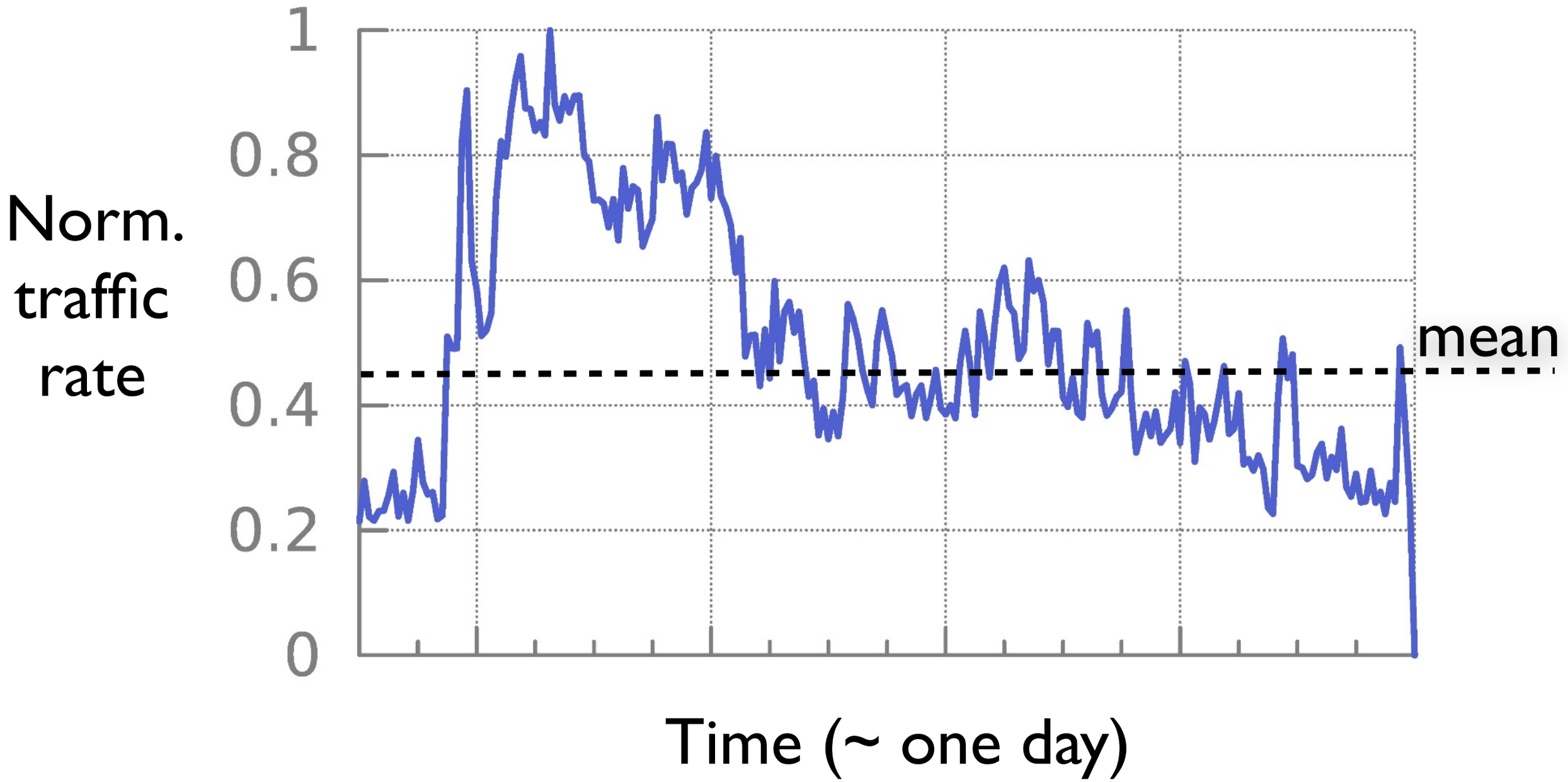
average utilization over time of busy links is only 30-50%

Poor sharing

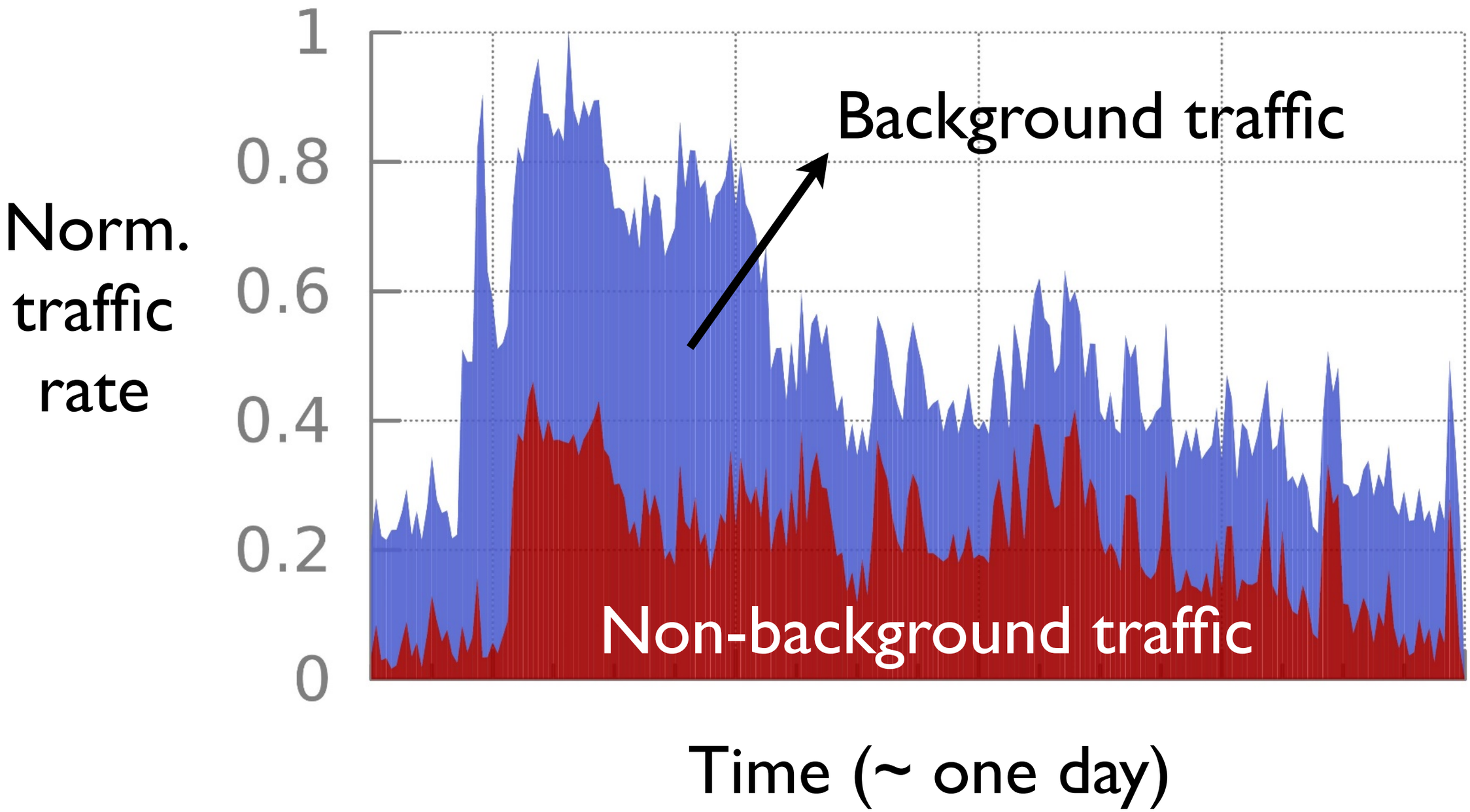
little support for flexible resource sharing

Why?

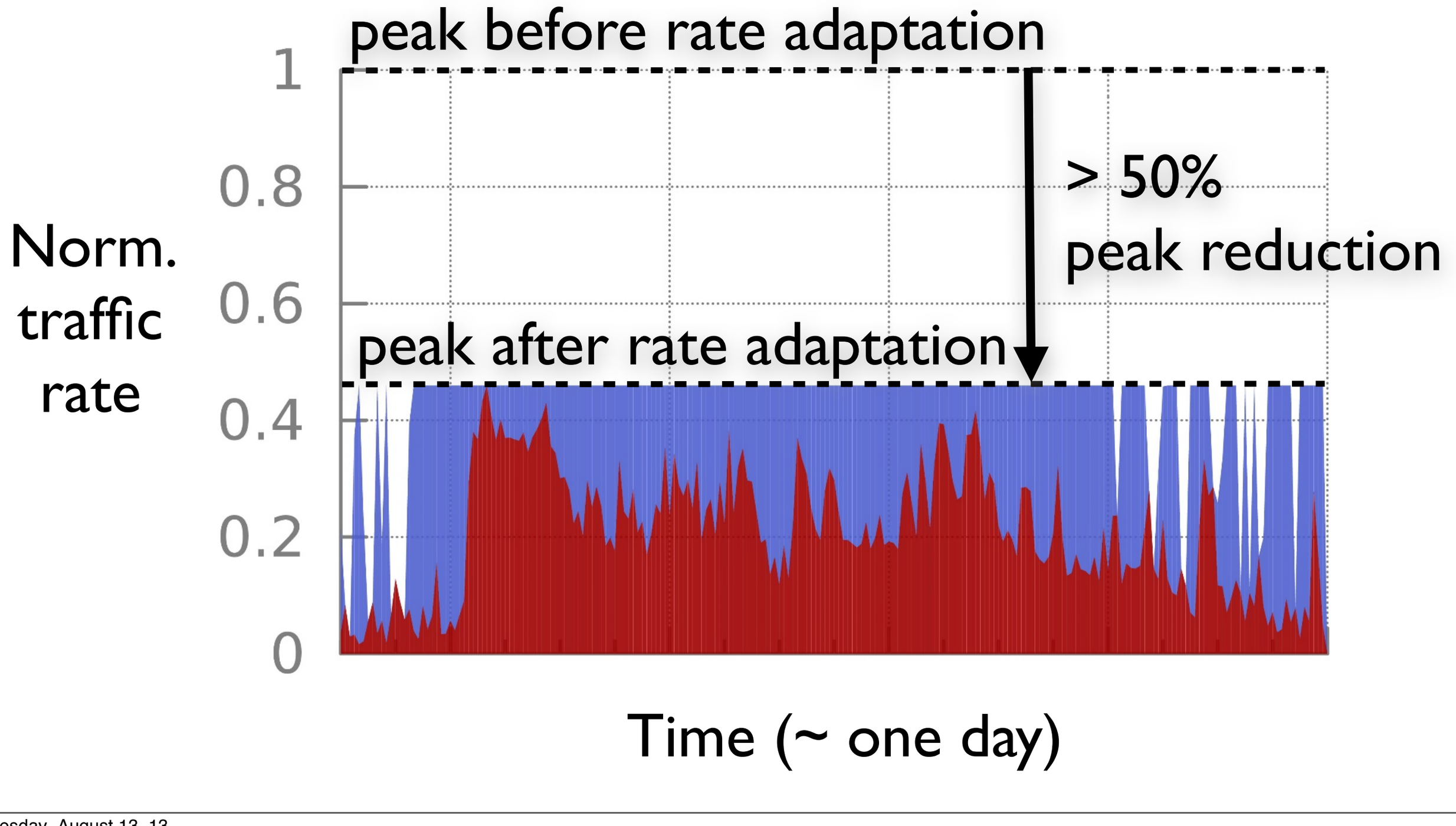
One cause of inefficiency: lack of coordination



One cause of inefficiency: lack of coordination



One cause of inefficiency: lack of coordination



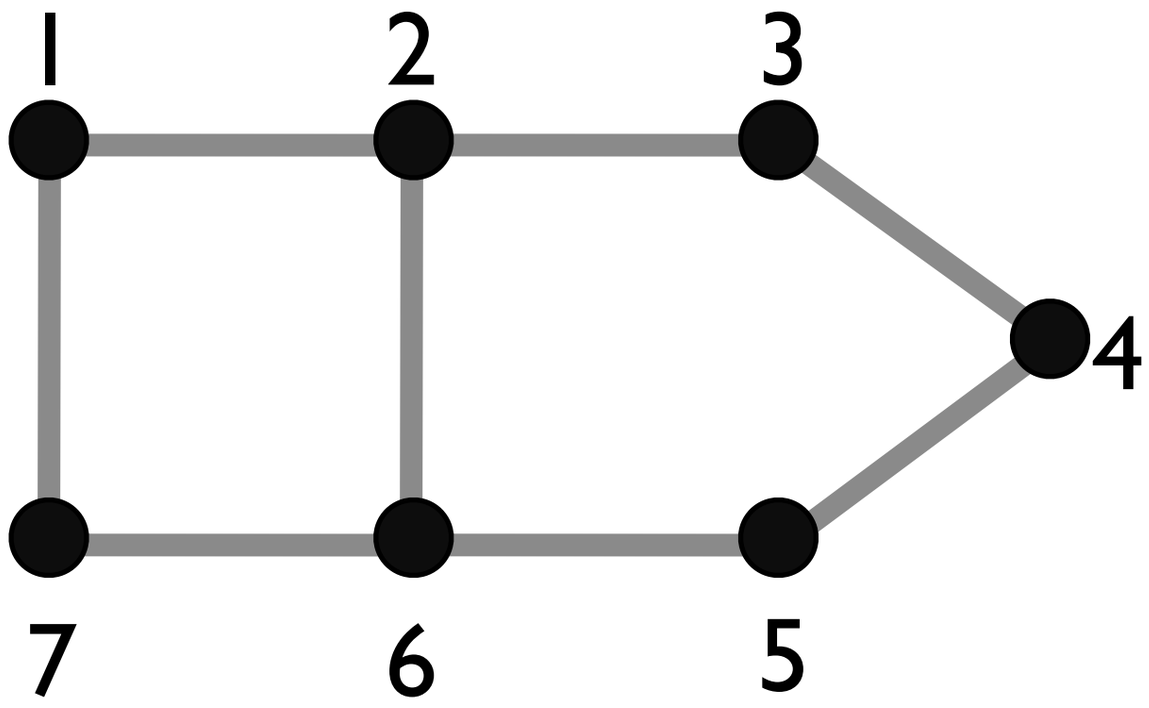
Another cause of inefficiency: local, greedy resource allocation

MPLS TE (Multiprotocol Label Switching Traffic Engineering) greedily selects shortest path fulfilling capacity constraint

Local, greedy resource allocation hurts efficiency

flow arrival order: A, B, C

each link can carry at most one flow



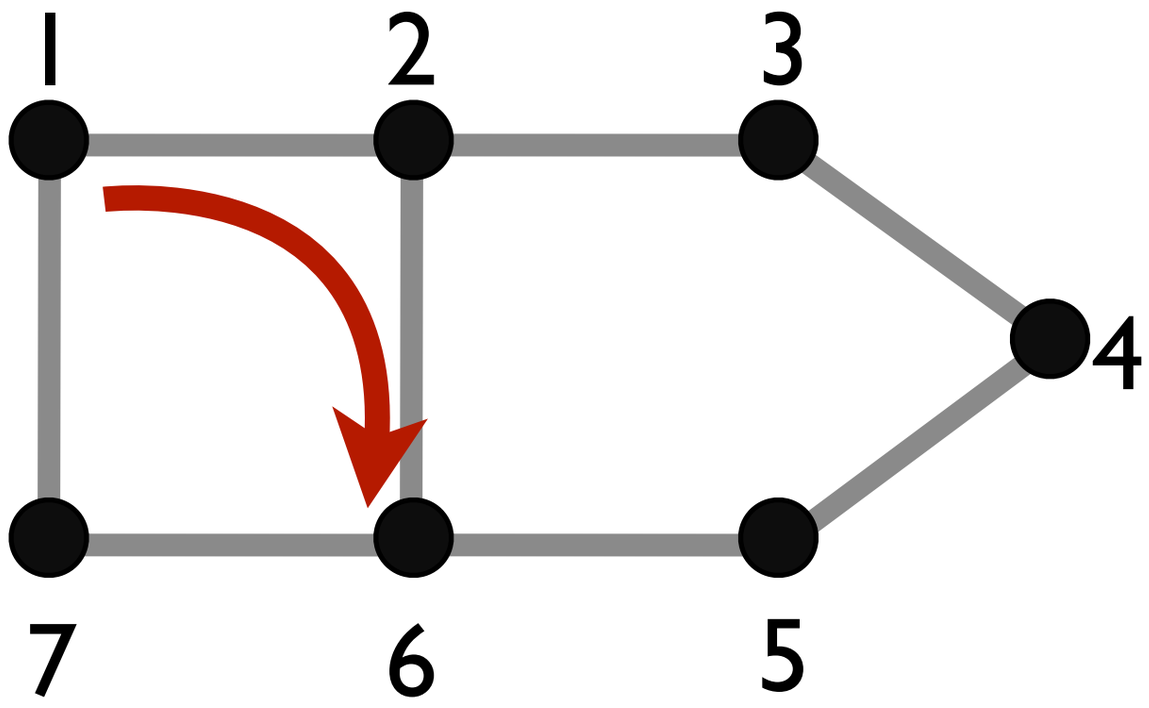
MPLS-TE

Flow	Src → Dst
A	1→6
B	3→6
C	4→6

Local, greedy resource allocation hurts efficiency

flow arrival order: A, B, C

each link can carry at most one flow



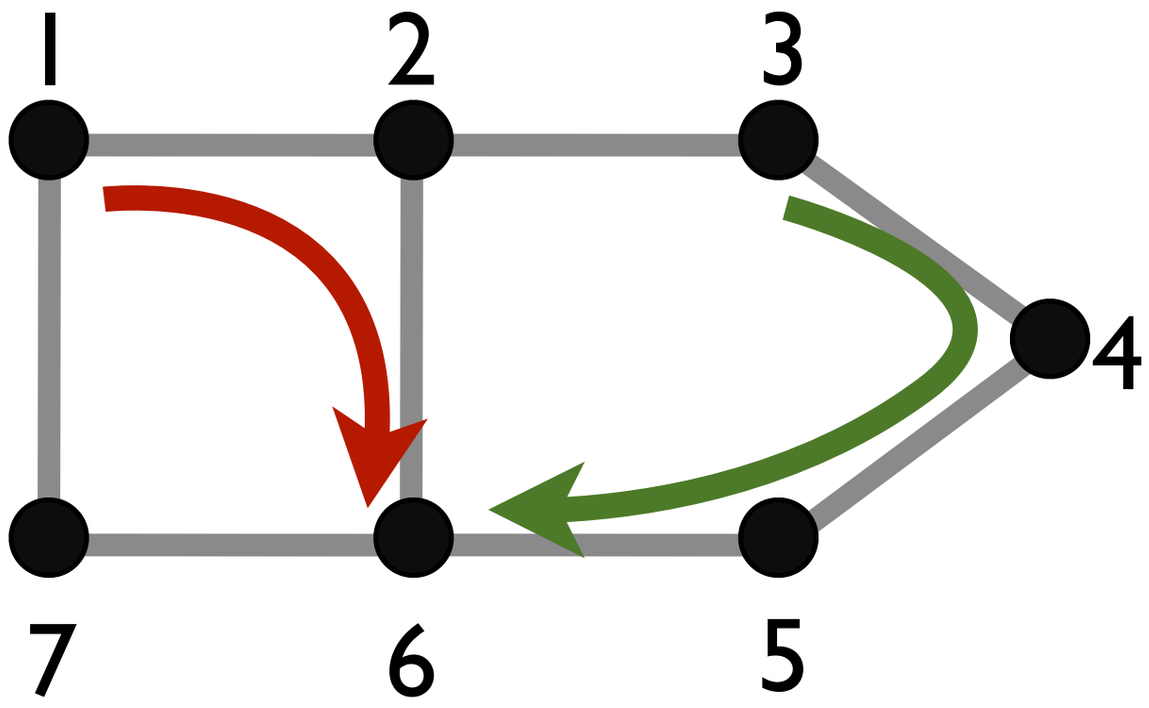
Flow	Src → Dst
A	1→6
B	3→6
C	4→6

MPLS-TE

Local, greedy resource allocation hurts efficiency

flow arrival order: A, B, C

each link can carry at most one flow



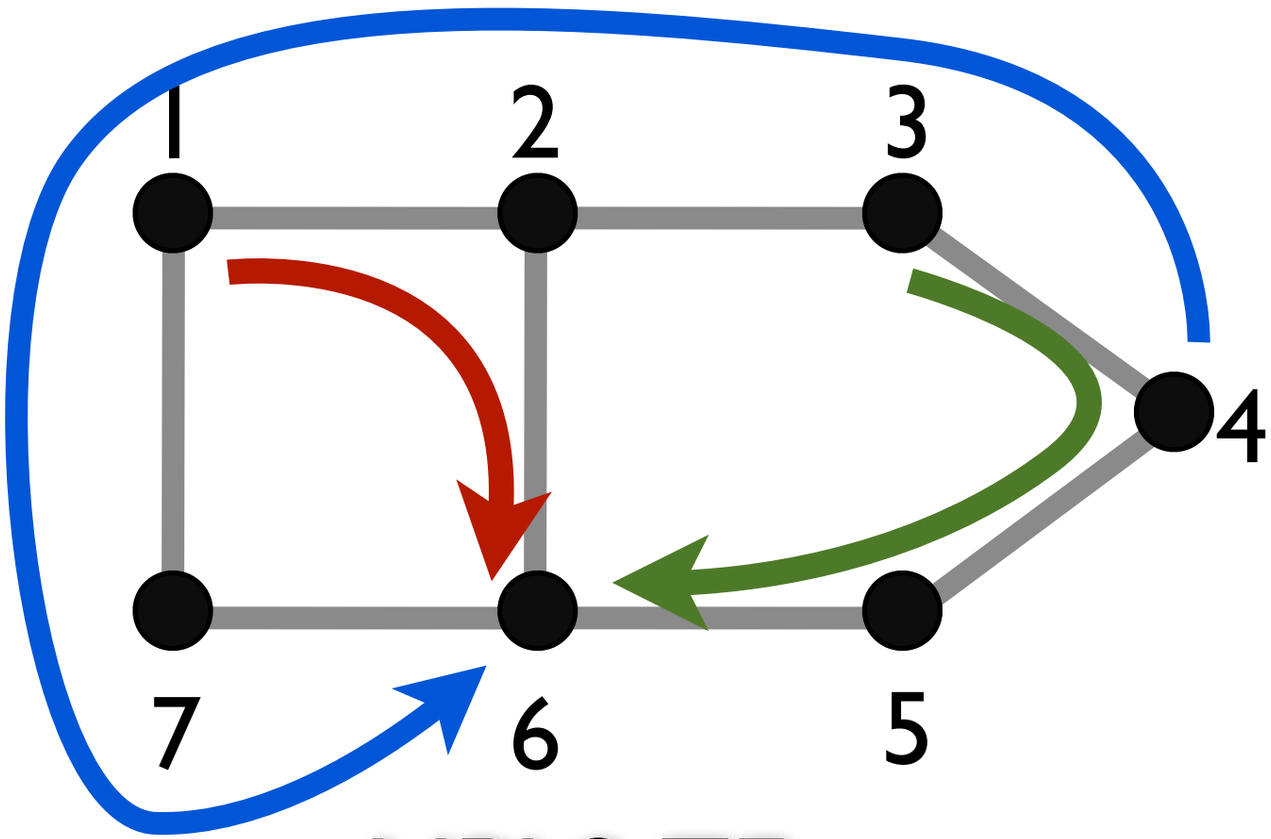
Flow	Src → Dst
A	1→6
B	3→6
C	4→6

MPLS-TE

Local, greedy resource allocation hurts efficiency

flow arrival order: A, B, C

each link can carry at most one flow



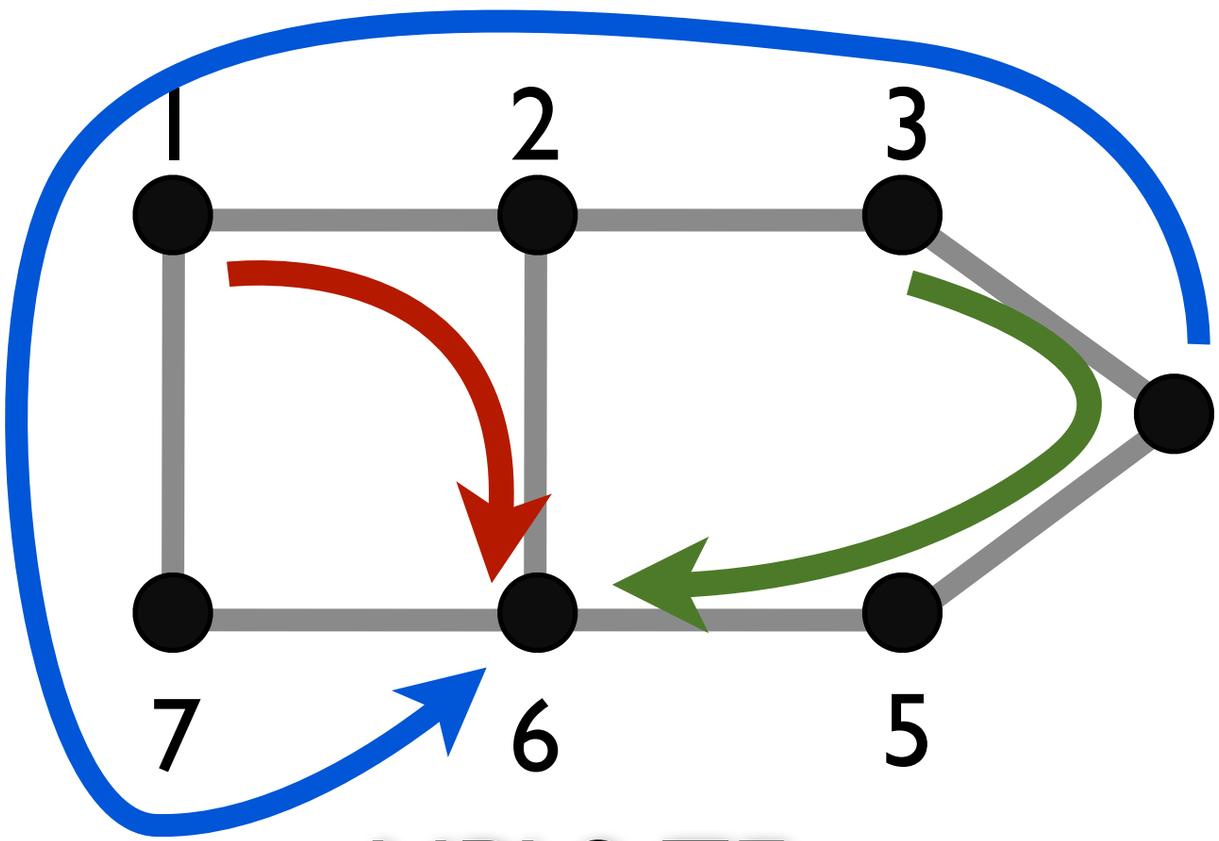
MPLS-TE

Flow	Src → Dst
A	1→6
B	3→6
C	4→6

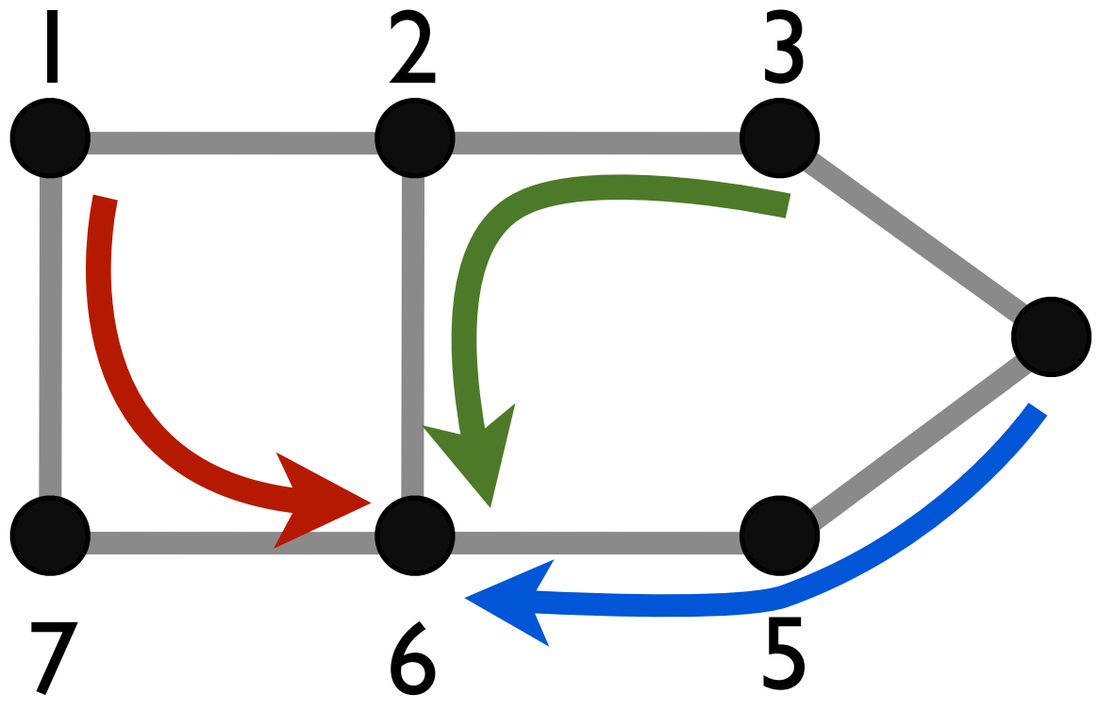
Local, greedy resource allocation hurts efficiency

flow arrival order: A, B, C

each link can carry at most one flow



MPLS-TE



Optimal

Poor sharing

Poor sharing

- When services compete today, they can get higher throughput by sending faster

Poor sharing

- When services compete today, they can get higher throughput by sending faster

- Mapping services onto different queues at switches helps, but # services ≫ # queues

(hundreds) (4 - 8 typically)

Poor sharing

- When services compete today, they can get higher throughput by sending faster

- Mapping services onto different queues at switches helps, but # services ≫ # queues

(hundreds) (4 - 8 typically)

Borrowing the idea of edge rate limiting, we can have better sharing without many queues

Our solution

Our solution

SWAN : Software-driven *WAN*

Our solution

SWAN : Software-driven *WAN*

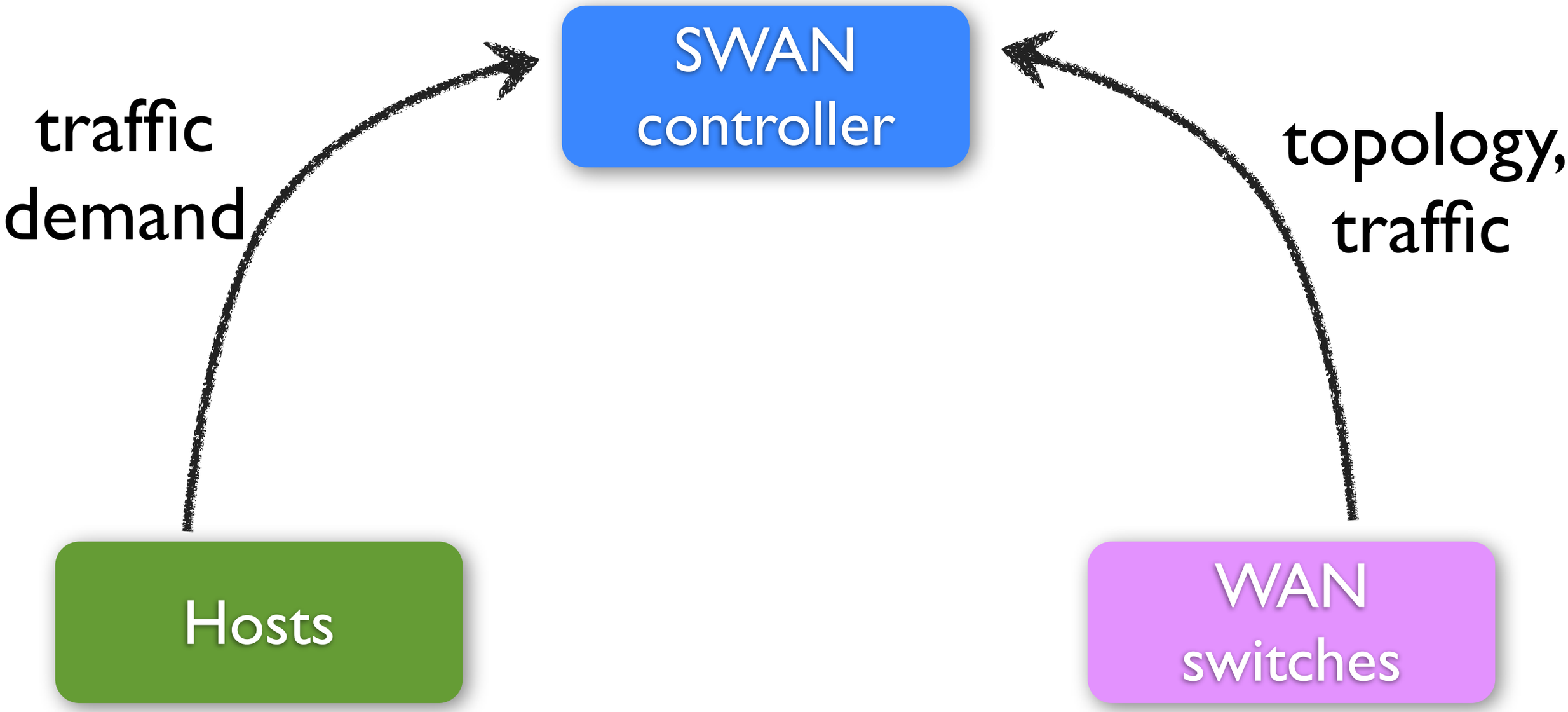
- high utilization
- flexible sharing

System flow

Hosts

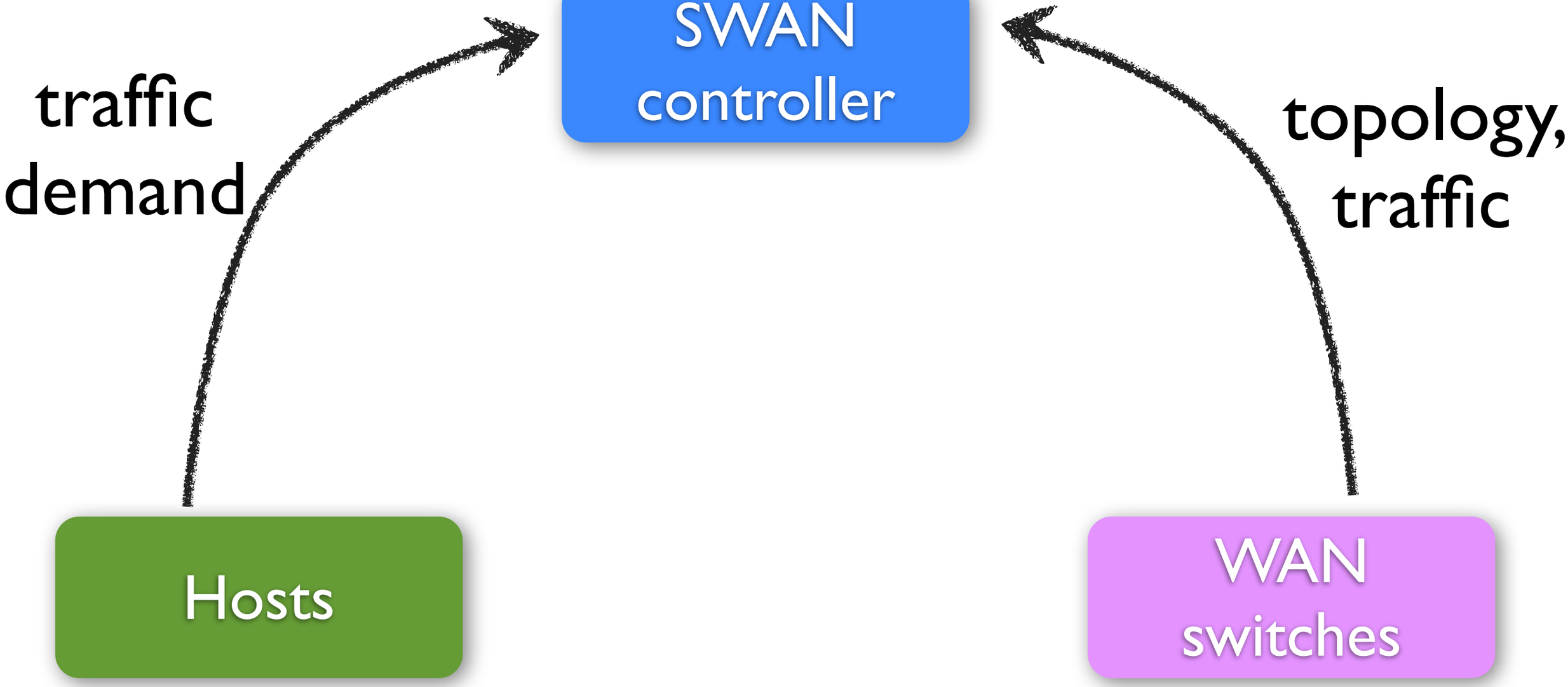
WAN switches

System flow


SWAN
controller
traffic
demand
topology,
traffic
Hosts
WAN
switches

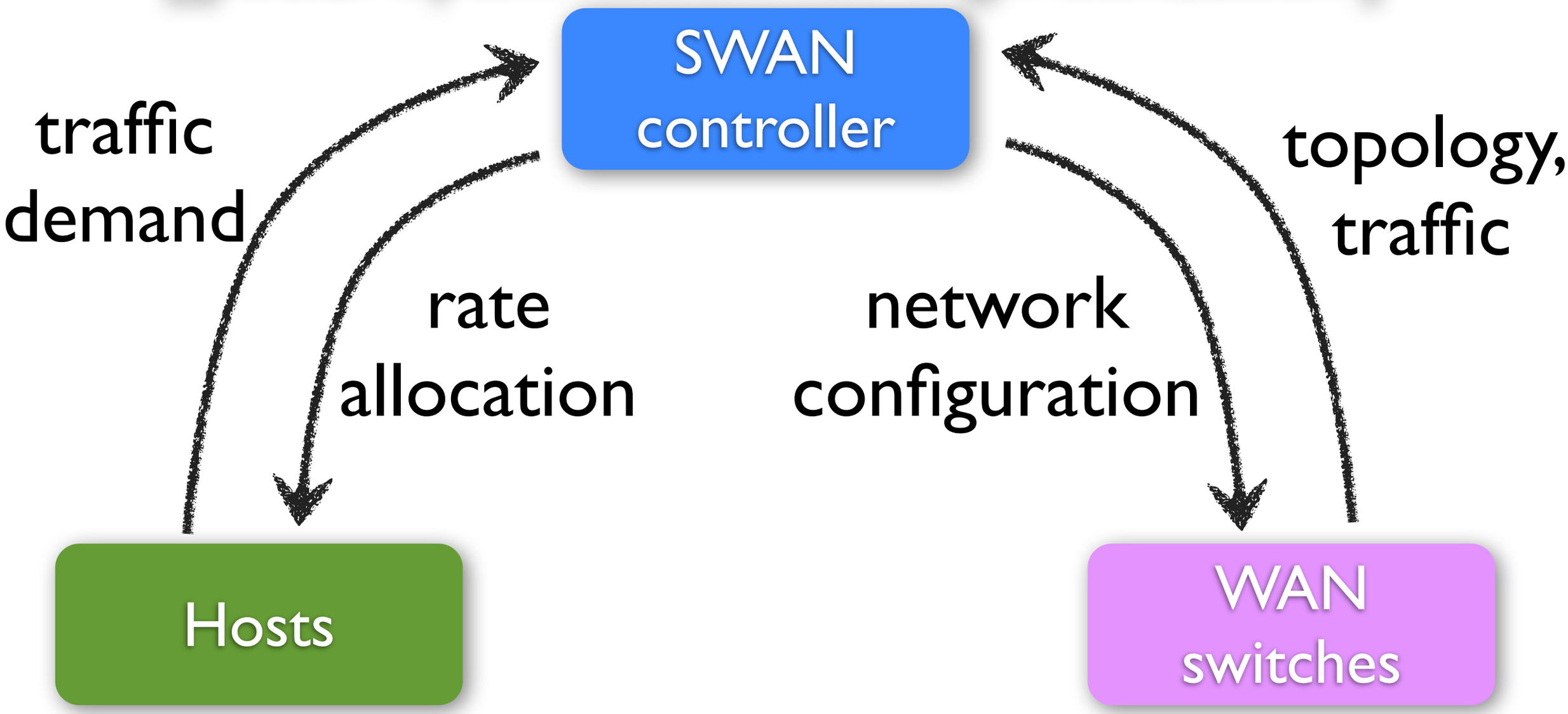
System flow

[global optimization for high utilization]



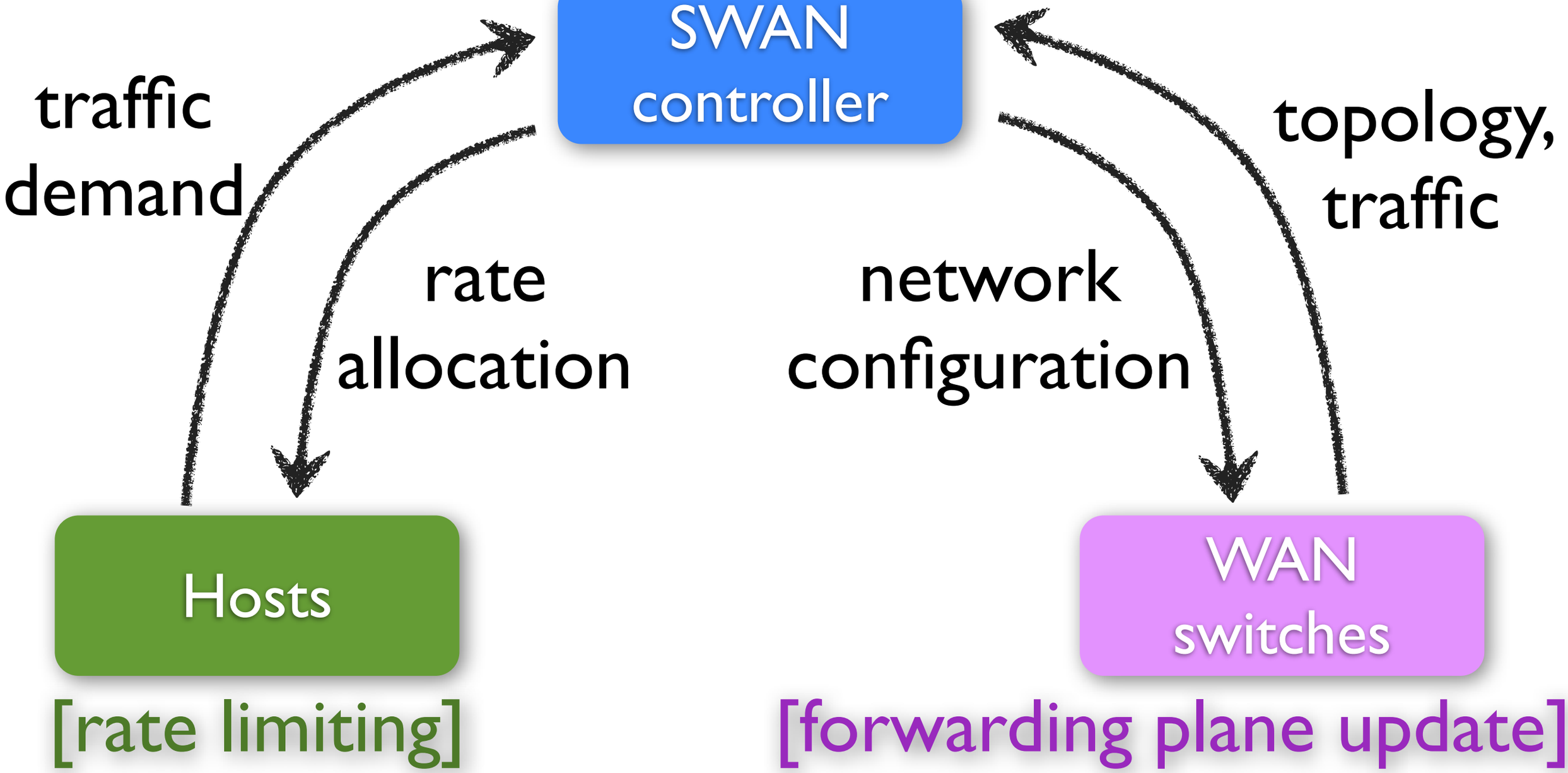
System flow

[global optimization for high utilization]



System flow

[global optimization for high utilization]



Challenges

- scalable allocation computation
- congestion-free data plane update
- working with limited switch memory

Challenge #1: How to compute allocation in a time-efficient manner?

Computing resource allocation

Path-constrained, multi-commodity flow problem

- allocate higher-priority traffic first
- ensure weighted max-min fairness within a class

Solving at the granularity of {DC pairs, priority class}-tuple

- split the allocation fairly among service flows

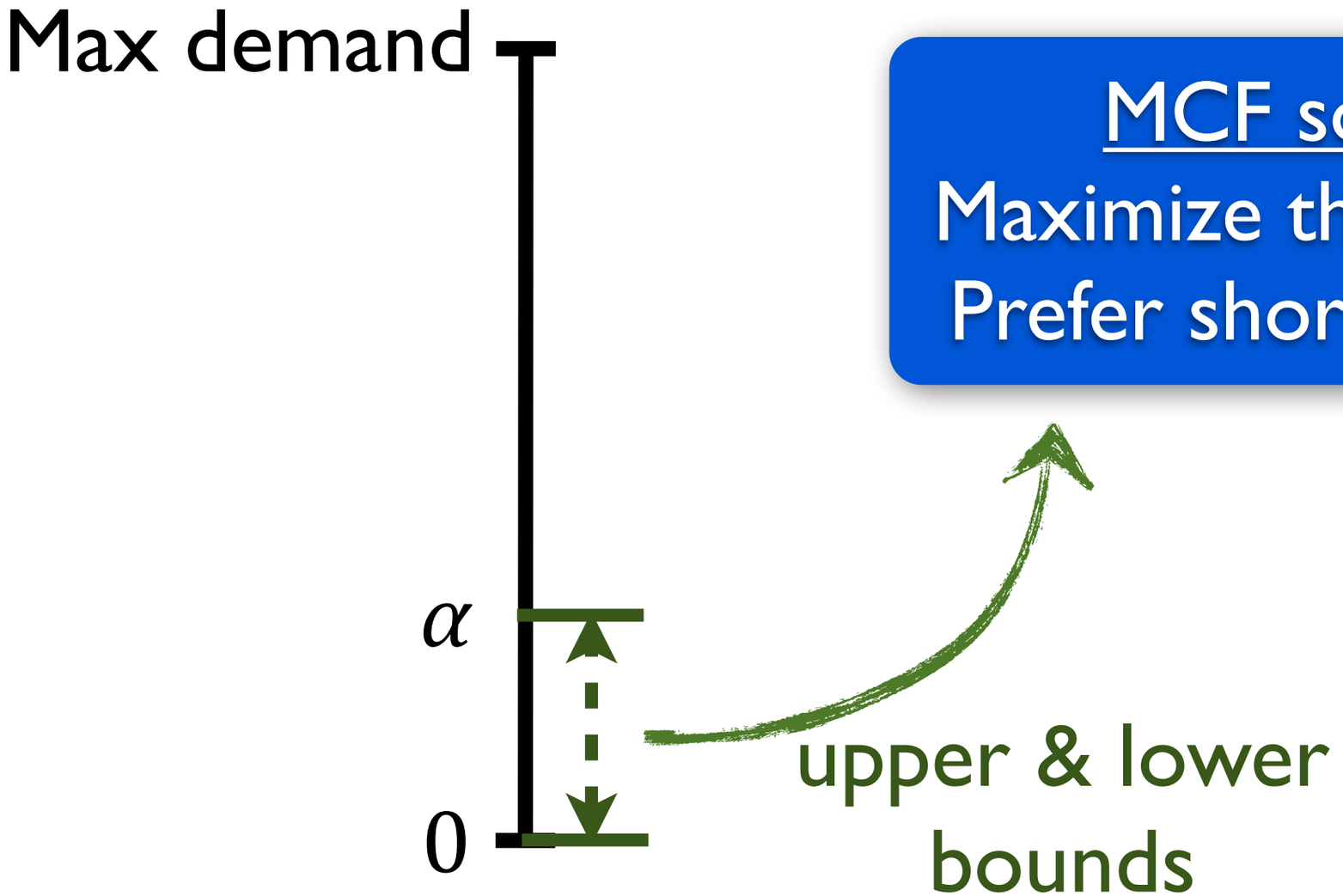
But computing max-min fairness is hard

State-of-the-art takes minutes at our target scale

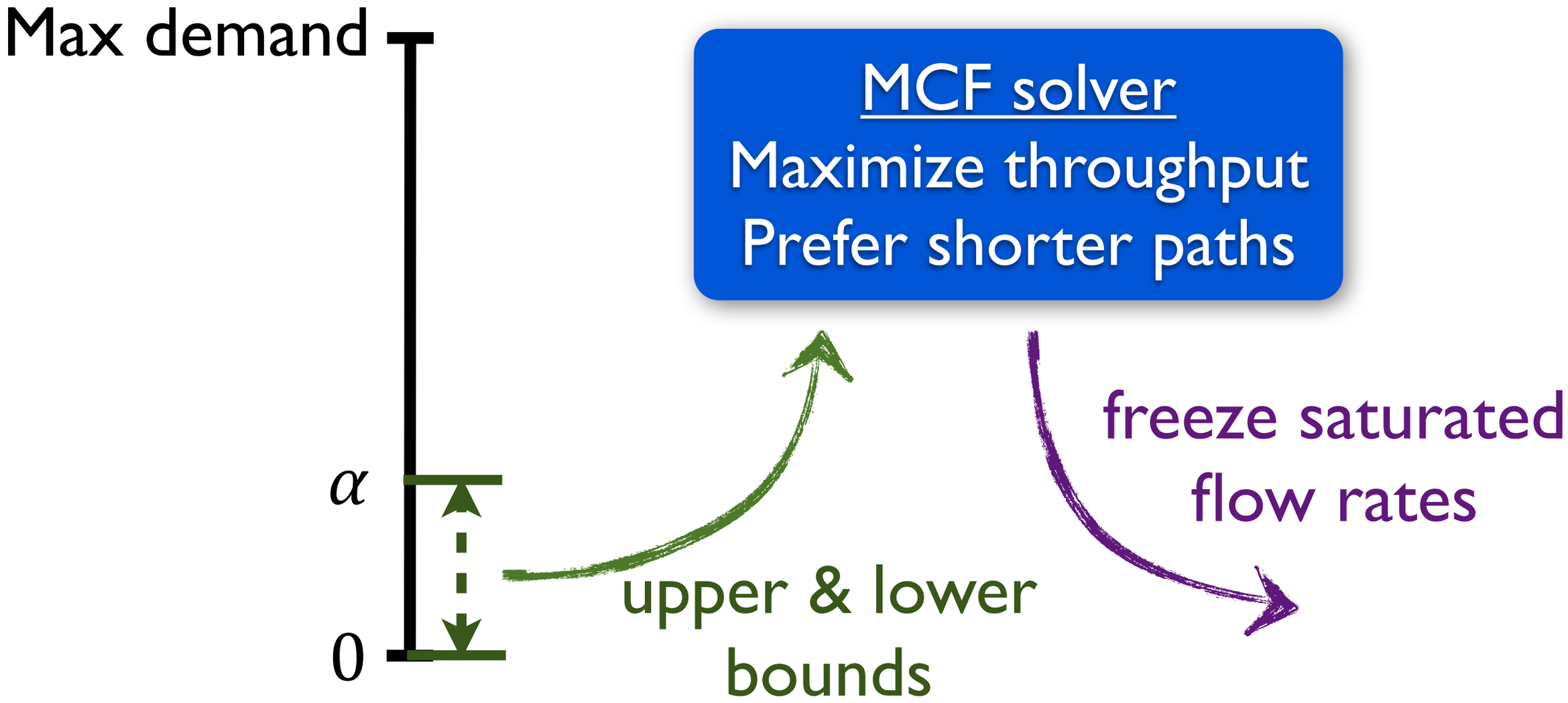
As it needs to solve a long sequence of LPs:
LPs = O(# saturated edges)

[Danna, Mandal, Singh; INFOCOM'12]

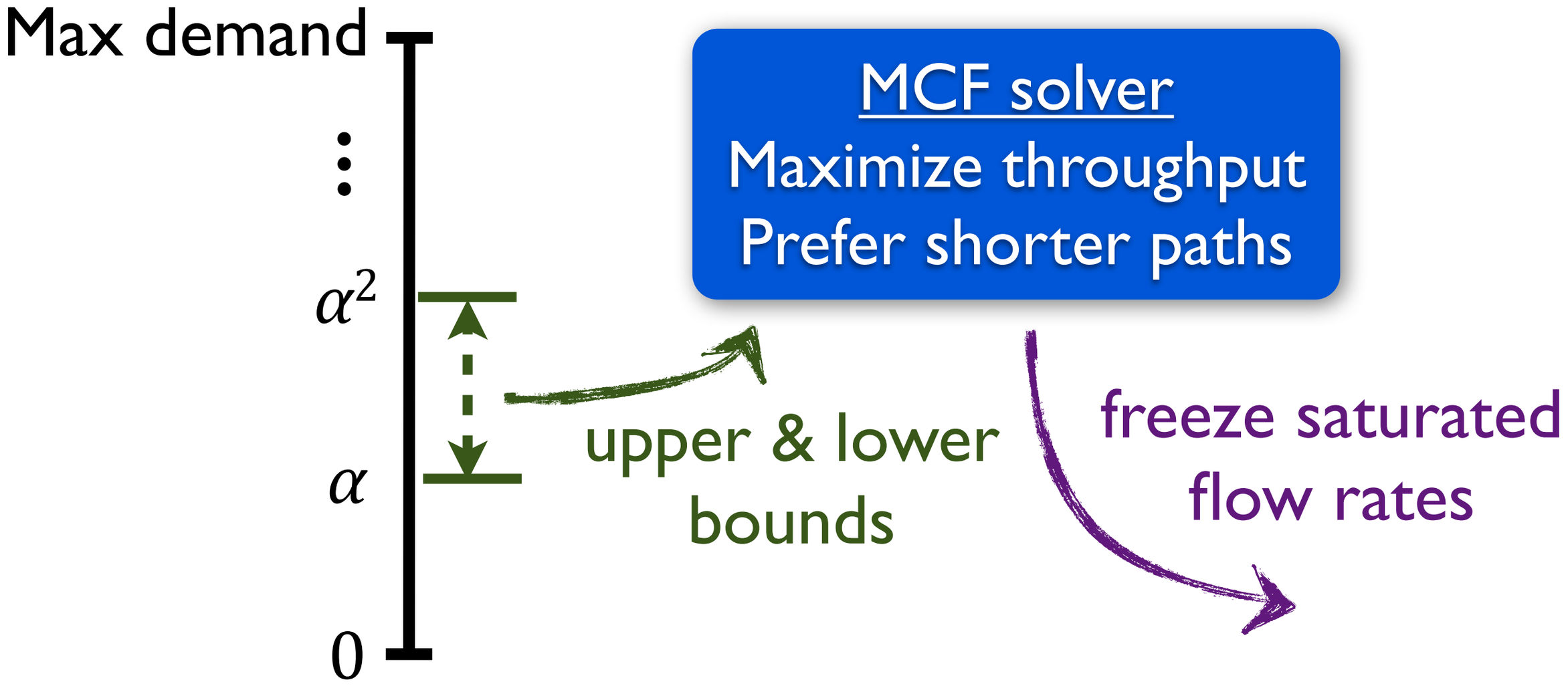
Approximated max-min fairness



Approximated max-min fairness



Approximated max-min fairness



Performance

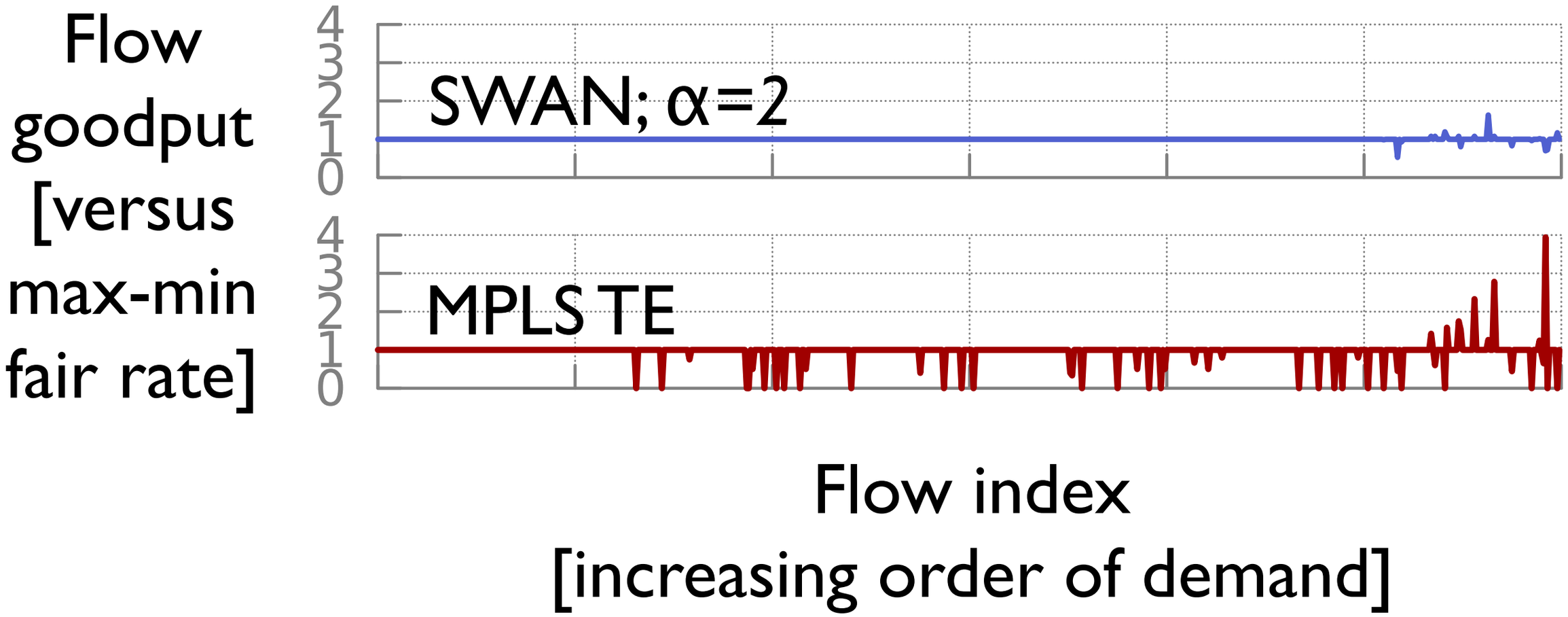
Theoretical bound:

$$MaxMinFair \ / \ \alpha \leq SwanRate \leq MaxMinFair \times \alpha$$

Empirical efficiency (with $\alpha = 2$):

- only 4% of flows deviate over 5% from their fair share rate
- sub-second computational time

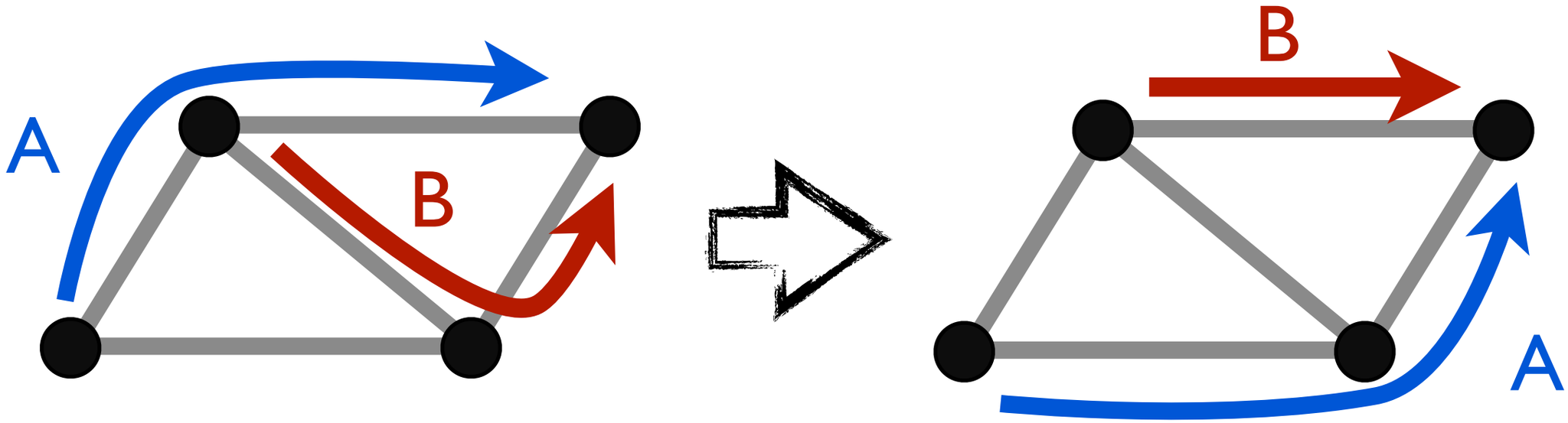
Fairness:
SWAN vs. MPLS TE



Challenge #2: Congestion-free update

How to update forwarding plane without causing transient congestion?

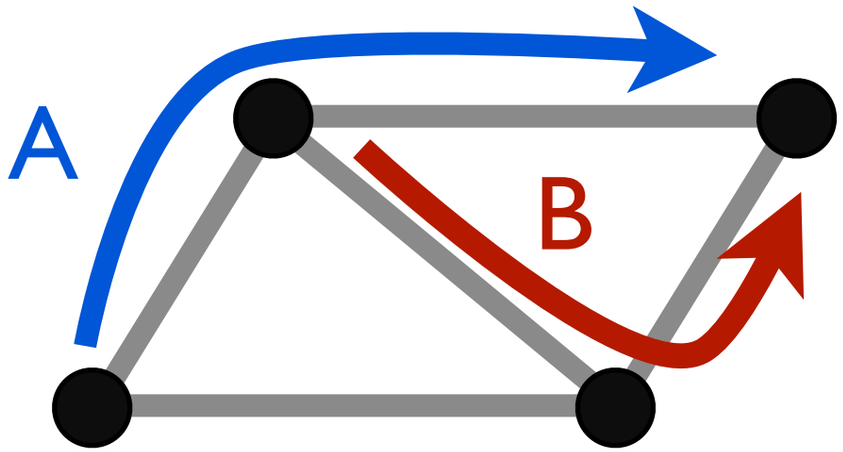
Congestion-free update is hard



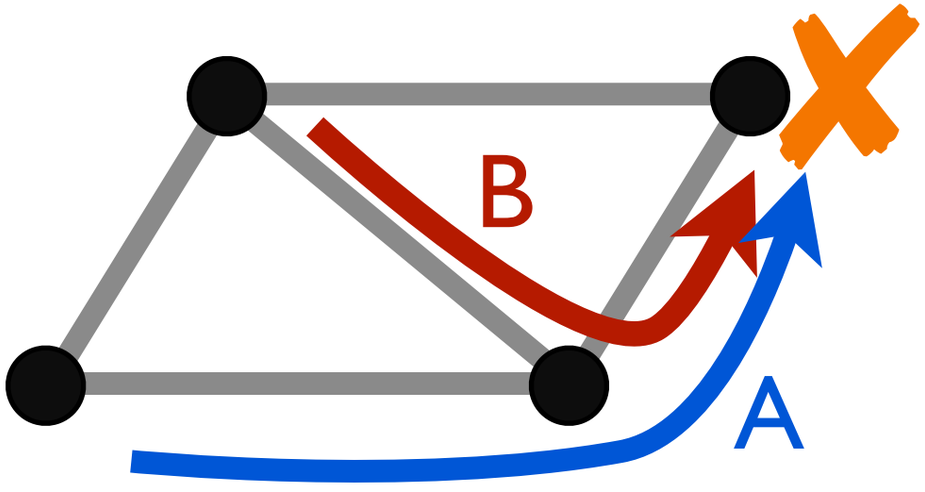
initial state

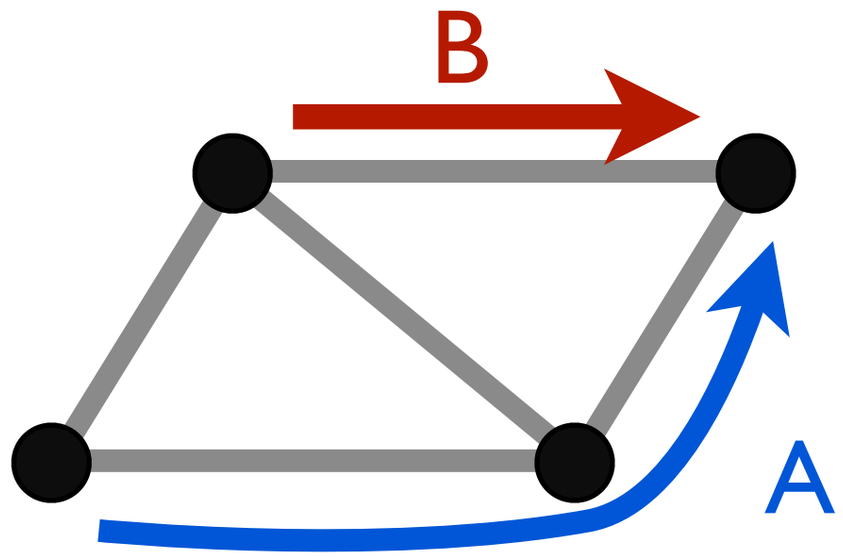
target state

Congestion-free update is hard



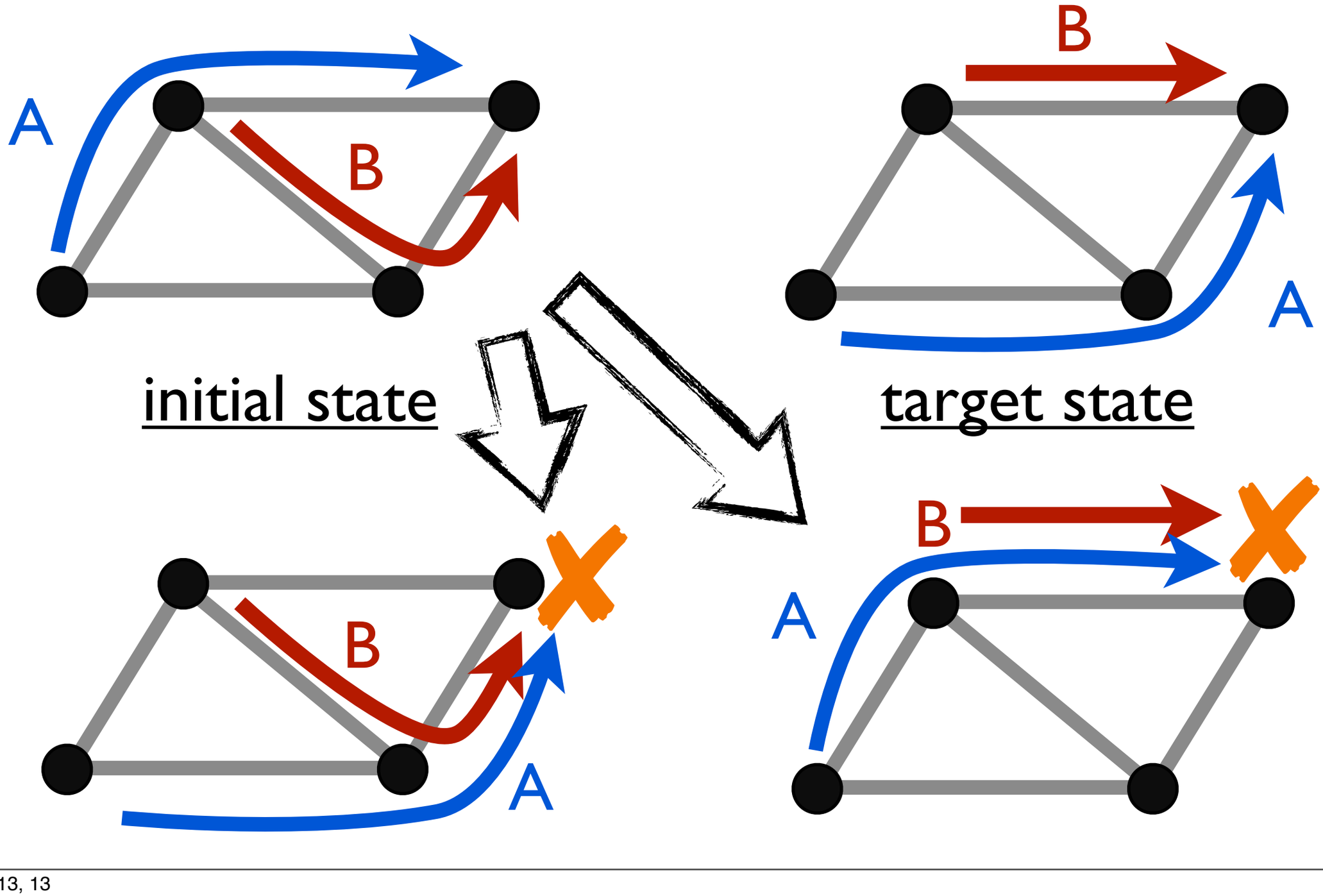
initial state





target state

Congestion-free update is hard



In fact, congestion-free update sequence might not exist!

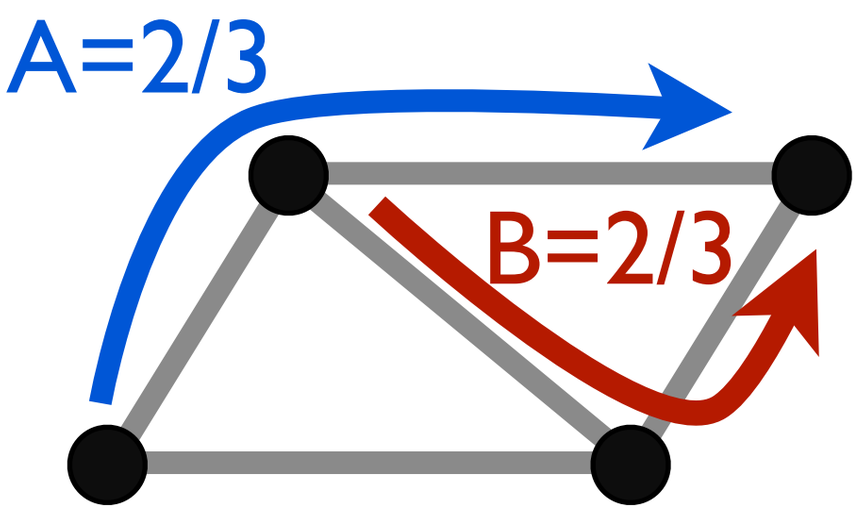
Idea

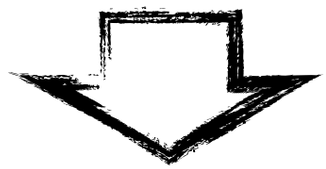
Leave a small amount of
scratch capacity on each link

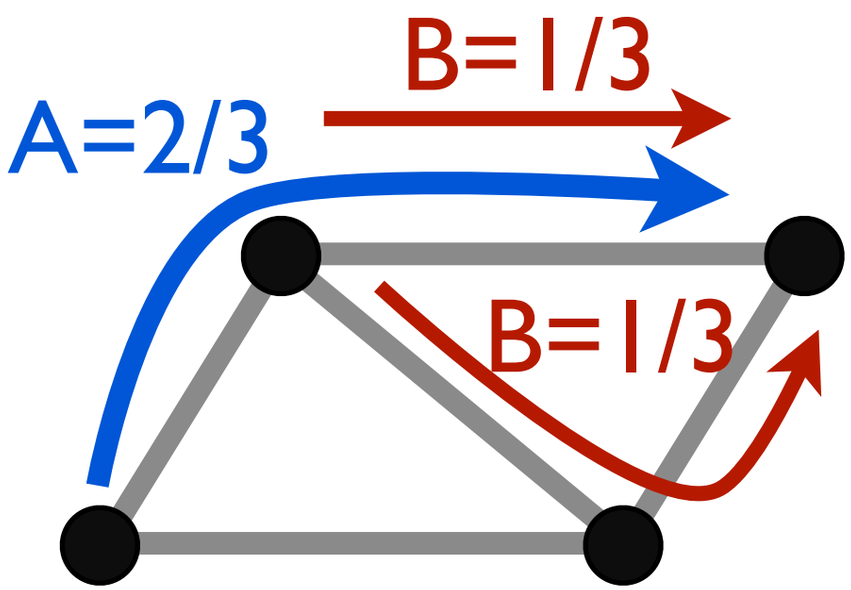
Slack = 1/3 of link capacity ...

Slack = 1/3 of link capacity ...

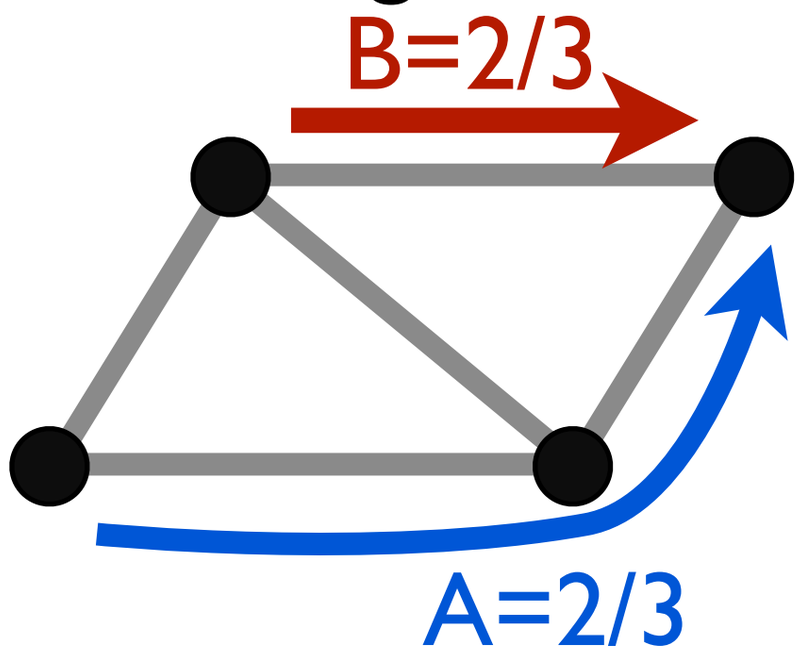
Init. state



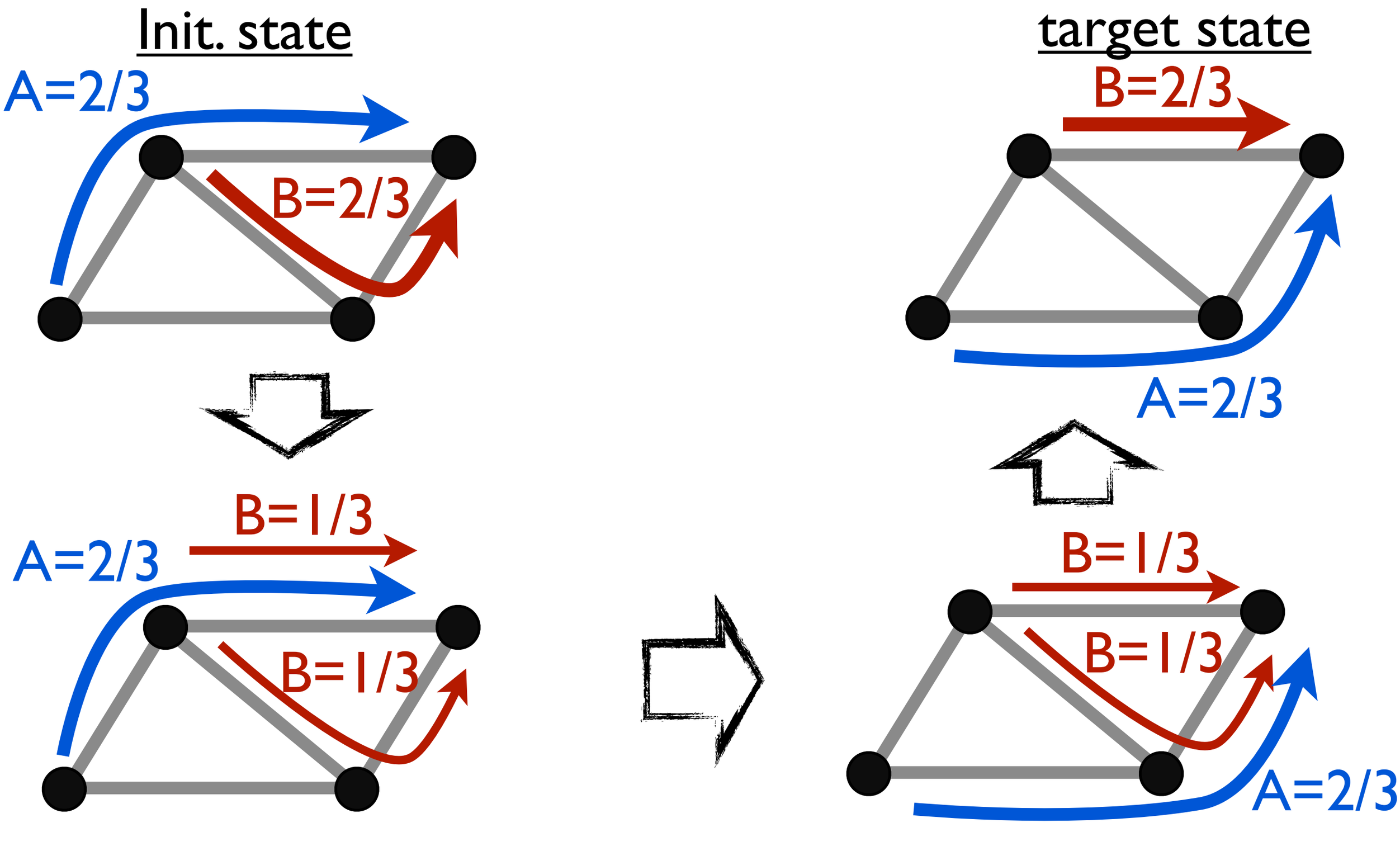




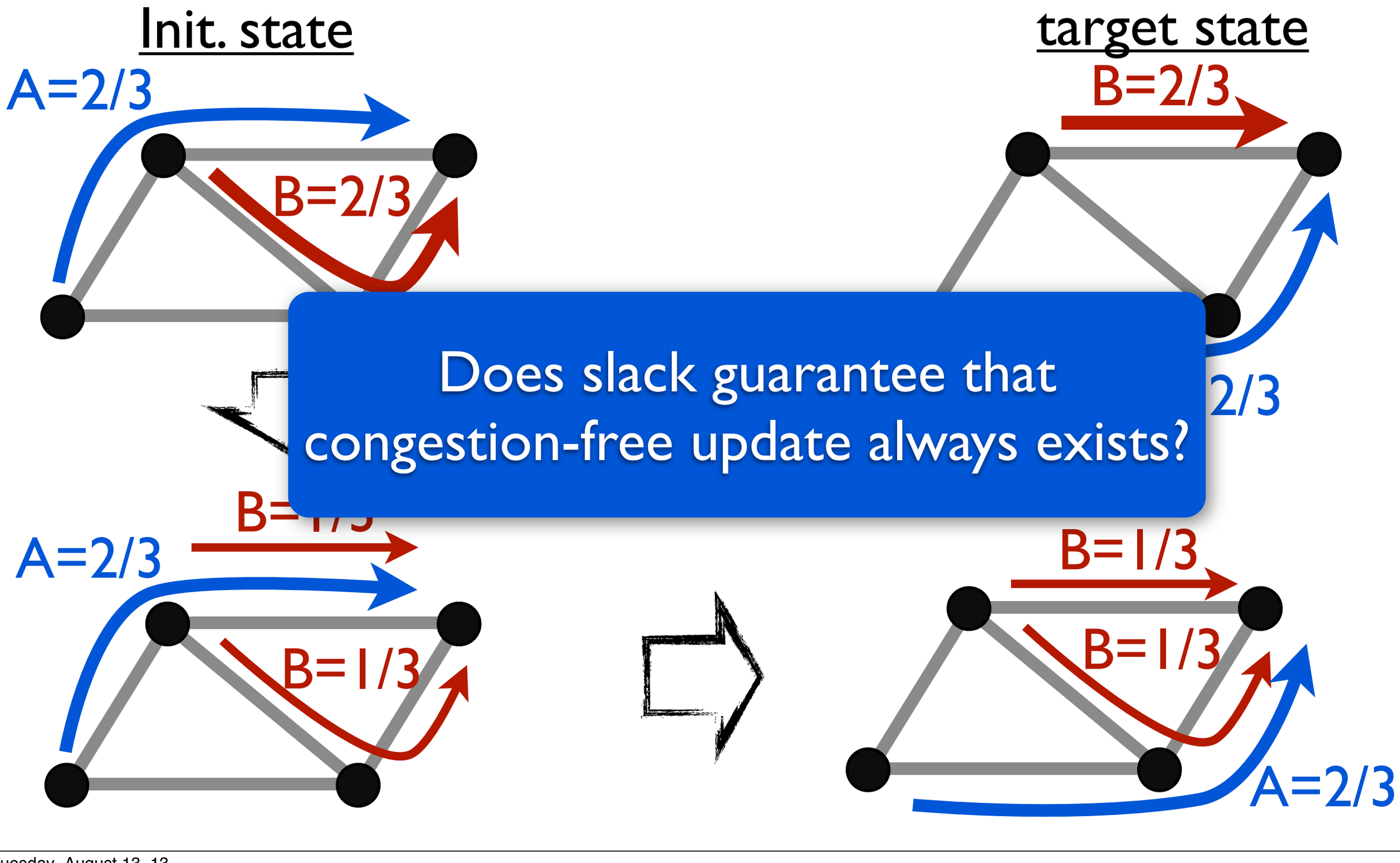
target state



Slack = 1/3 of link capacity ...



Slack = 1/3 of link capacity ...



Yes!

With slack $s \in \{0, 50\%\}$:

- we prove there exists a congestion-free update in $\leq \left\lceil \frac{1}{s} \right\rceil - 1$ steps

one step = multiple updates
whose order can be arbitrary

Yes!

With slack $s \in \{0, 50\%\}$:

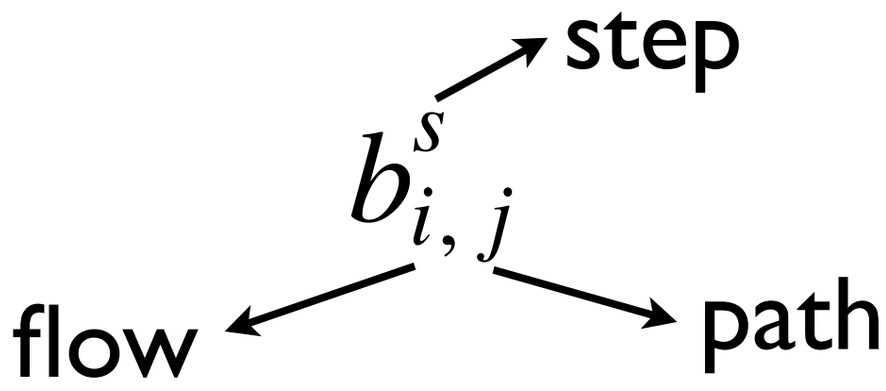
- we prove there exists a congestion-free update in $\leq \left\lceil \frac{1}{s} \right\rceil$-1 steps

one step = multiple updates
whose order can be arbitrary

It exsits, but how to find it?

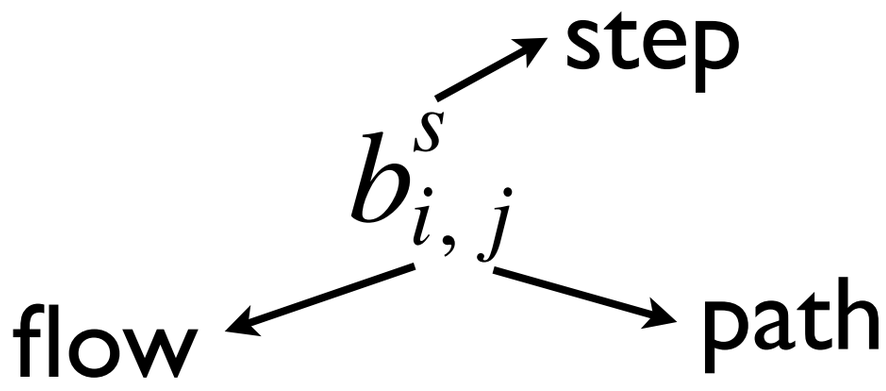
Congestion-free update: LP-based solution

- rate variable:



Congestion-free update: LP-based solution

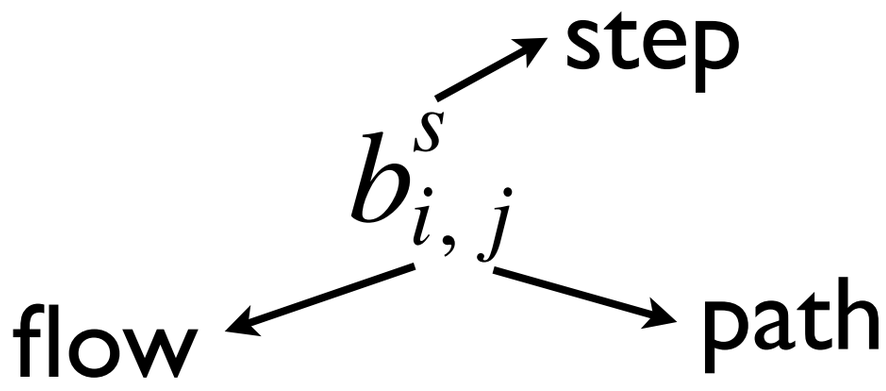
- rate variable: $b^{s}_{i,j}$



- input: $b^{0}_{i,j}$ and $b^{k}_{i,j}$
- output: $b^{1}_{i,j}$... $b^{k\text{-}1}_{i,j}$

Congestion-free update: LP-based solution

- rate variable: $b^{s}_{i,j}$



- input: $b^{0}_{i,j}$ and $b^{k}_{i,j}$
- output: $b^{1}_{i,j}$... $b^{k\text{-}1}_{i,j}$
- congestion-free constraint:

$$\sum max\left(b^{s}_{i,j}, b^{s+1}_{i,j}\right) \leq \text{link capacity}$$

∀i,j on a link

Utilizing all the capacity

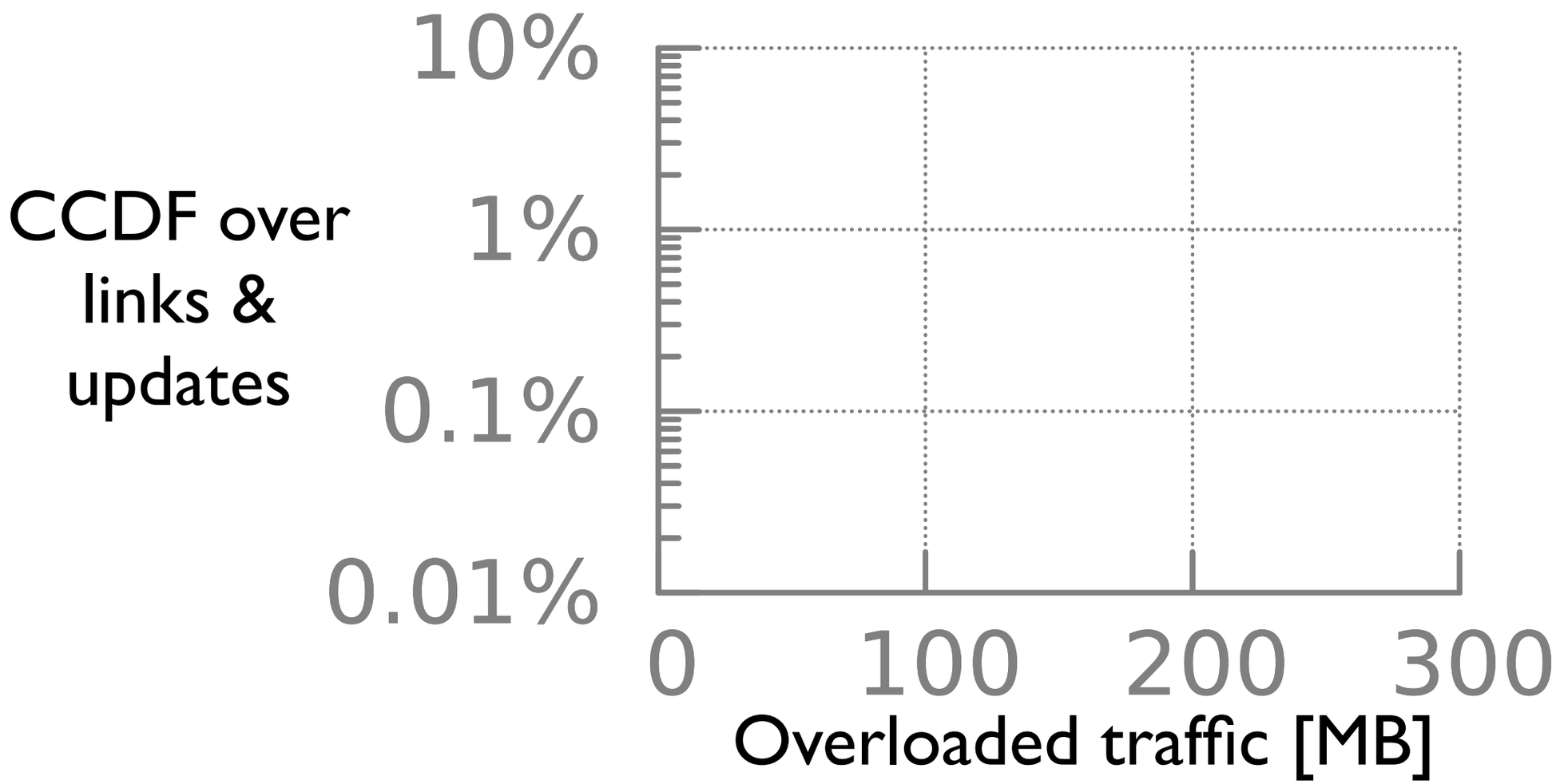
non-background is congestion-free

using 90% capacity ($s = 10\%$)

background has bounded congestion

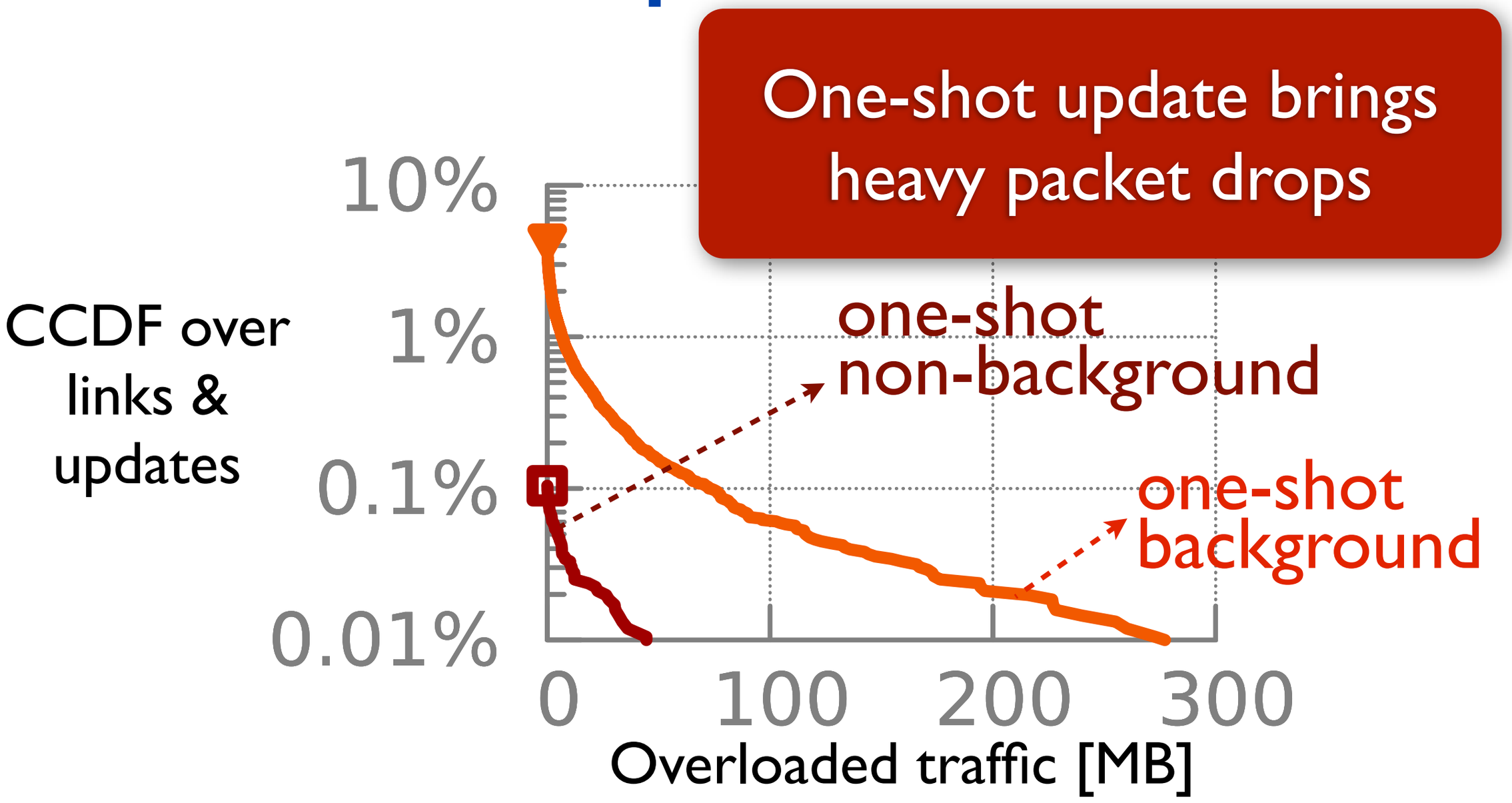
using 100% capacity ($s = 0\%$)

SWAN versus one-shot update



[data-driven evaluation; s = 10% for non-background]

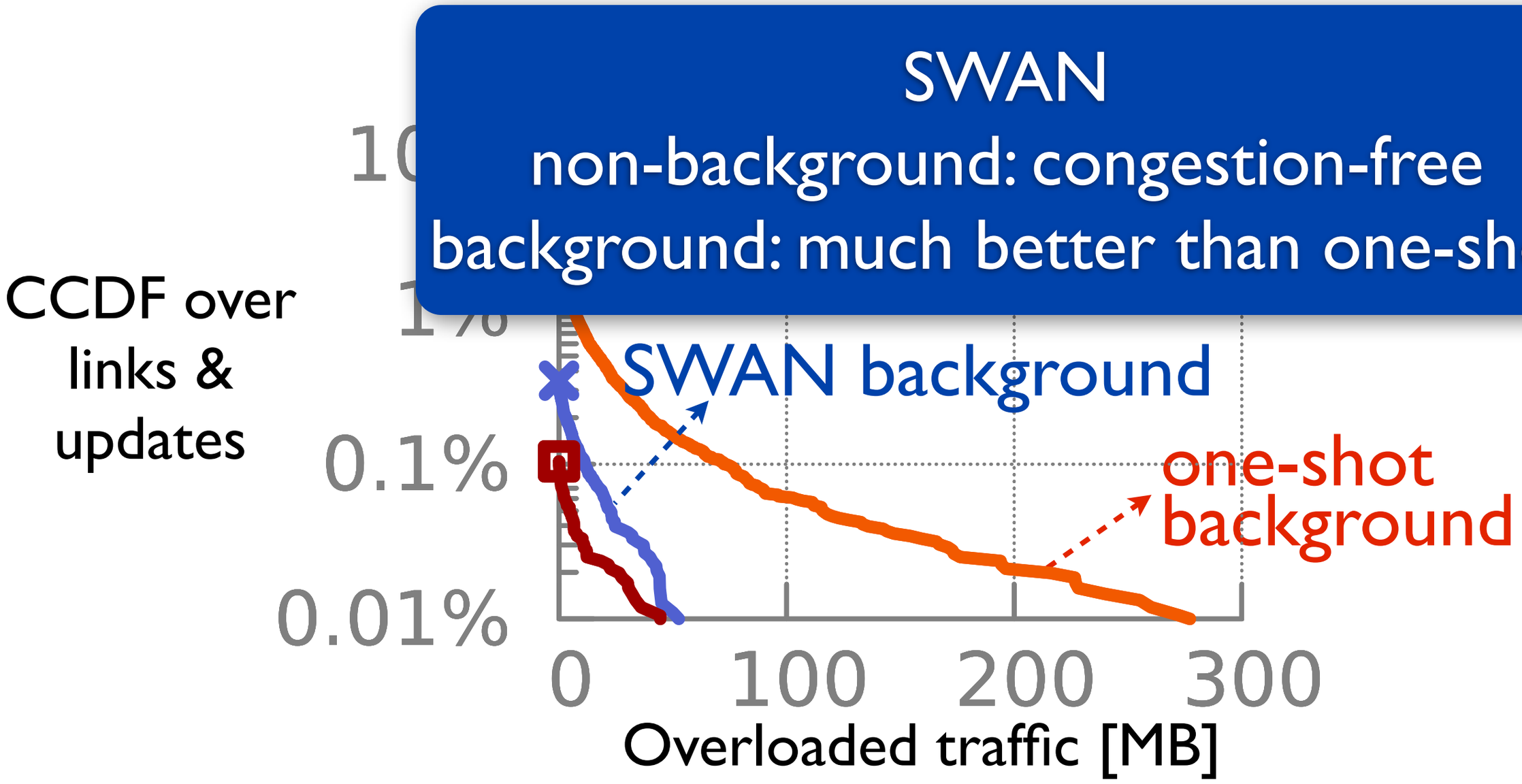
SWAN versus one-shot update



[data-driven evaluation; s = 10% for non-background]

SWAN versus one-shot update

SWAN
non-background: congestion-free
background: much better than one-shot



[data-driven evaluation; s = 10% for non-background]

Challenge #3

Working with limited switch memory

Why does switch memory matter?

Why does switch memory matter?

How many we need?

- 50 sites = 2,500 pairs
- 3 priority classes
- static k-shortest path routing

[by data-driven analysis]

Why does switch memory matter?

How many we need?

- 50 sites = 2,500 pairs
- 3 priority classes
- static k-shortest path routing

[by data-driven analysis]

it requires 20K rules to fully use network capacity

Why does switch memory matter?

How many we need?

- 50 sites = 2,500 pairs
- 3 priority classes
- static k-shortest path routing

[by data-driven analysis]

it requires 20K rules to fully use network capacity

Commodity switches has limited memory:

- today's OpenFlow switch: 1-4K rules
- next generation: 16K rules [Broadcom Trident II]

Hardness

Finding the set of paths with a given size that carries the most traffic is NP-complete

[Hartman et al., INFOCOM'12]

Heuristic: Dynamic path set adaptation

Heuristic: Dynamic path set adaptation

Observation:

- working path set ≪ total needed paths

Heuristic: Dynamic path set adaptation

Observation:

- working path set ≪ total needed paths

Path selection:

- important ones that carry more traffic and provide basic connectivity
- 10x fewer rules than static k-shortest path routing

Heuristic: Dynamic path set adaptation

Observation:

- working path set ≪ total needed paths

Path selection:

- important ones that carry more traffic and provide basic connectivity
- 10x fewer rules than static k-shortest path routing

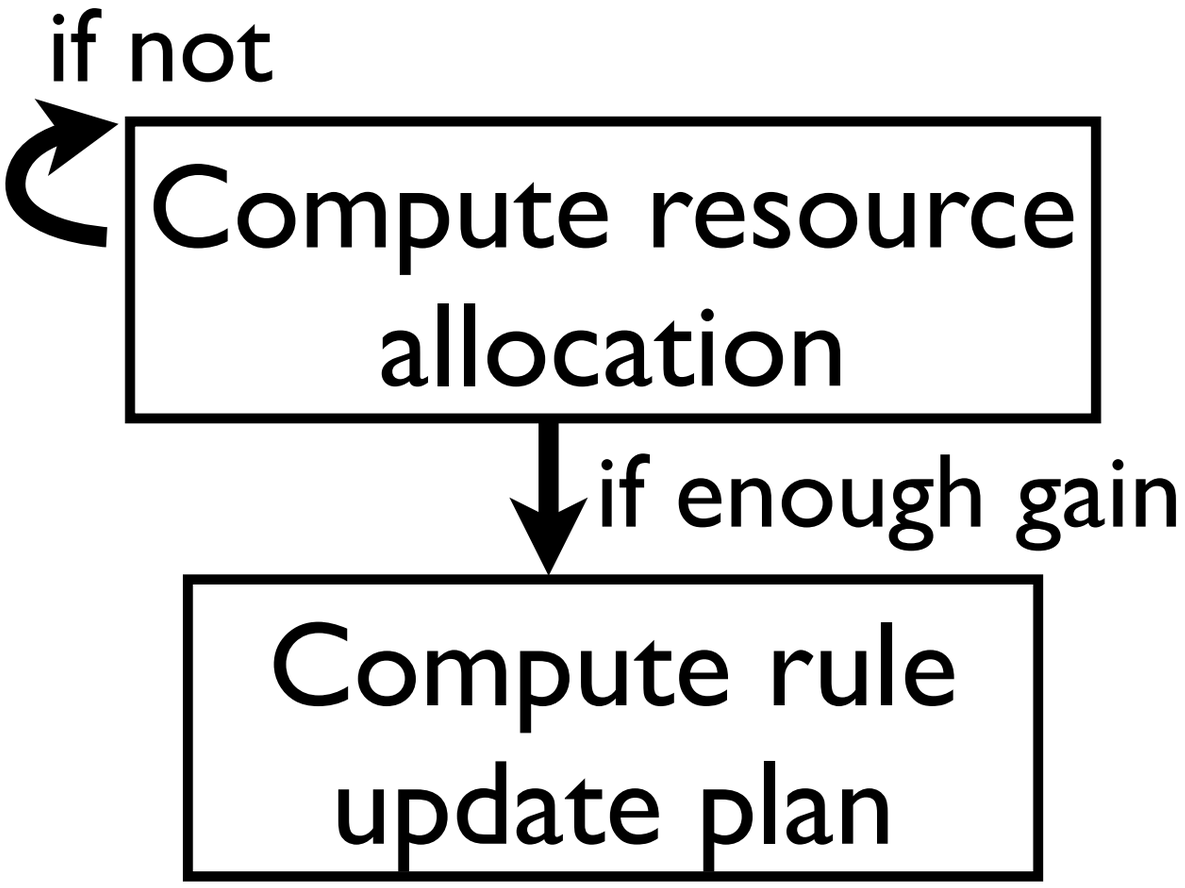
Rule update:

- multi-stage rule update
- with 10% memory slack, typically 2 stages needed

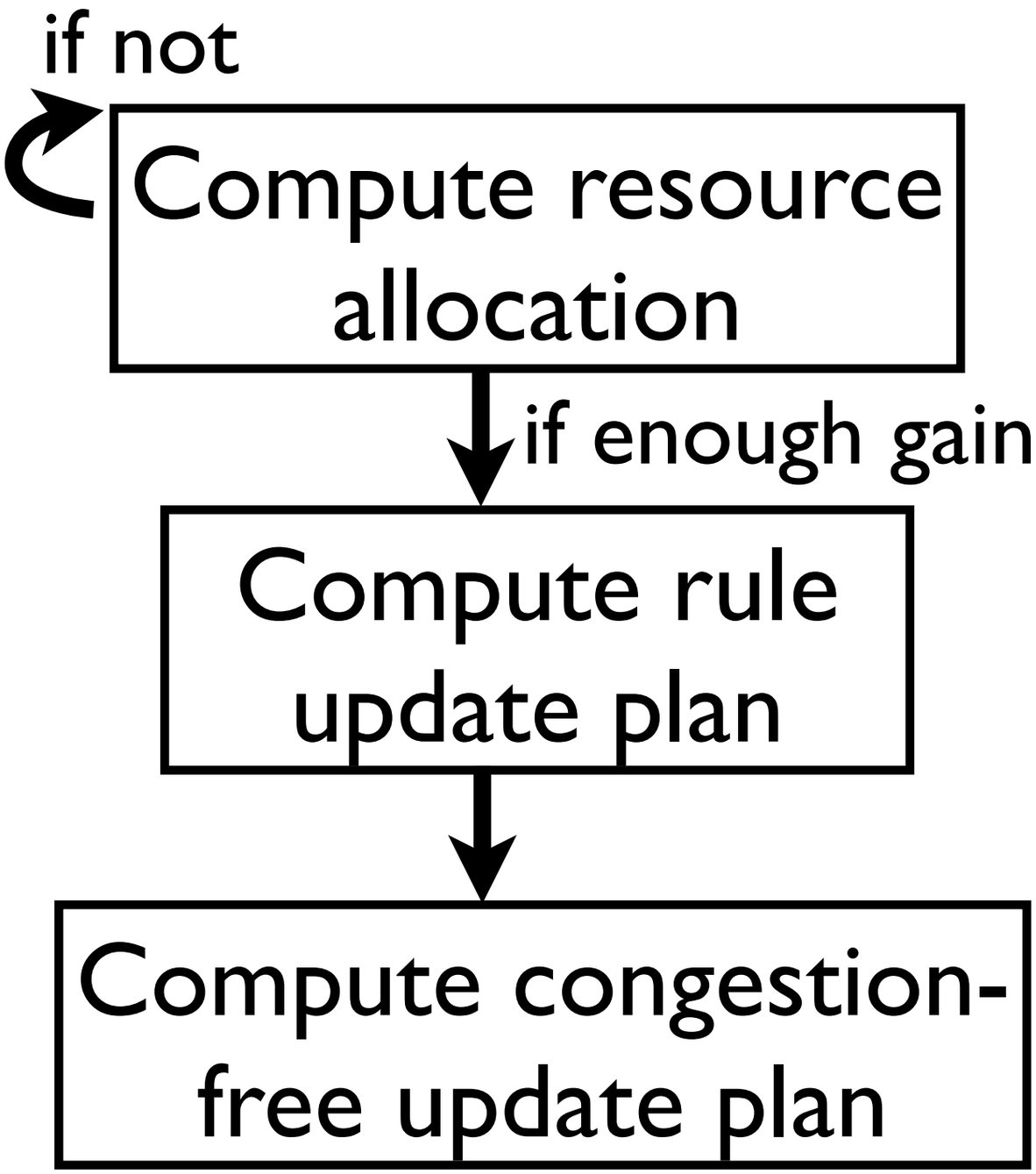
Overall workflow

Compute resource allocation

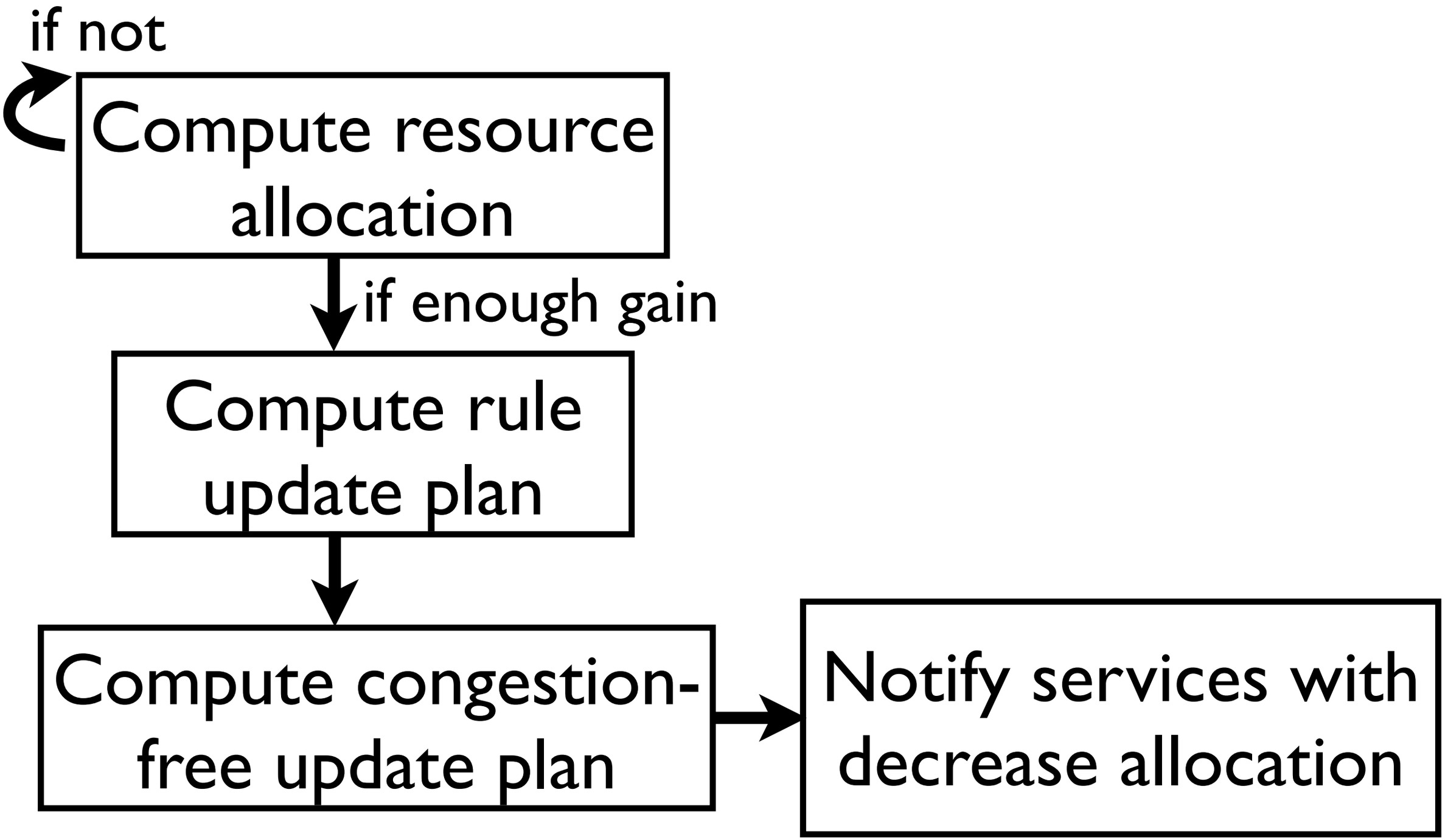
Overall workflow



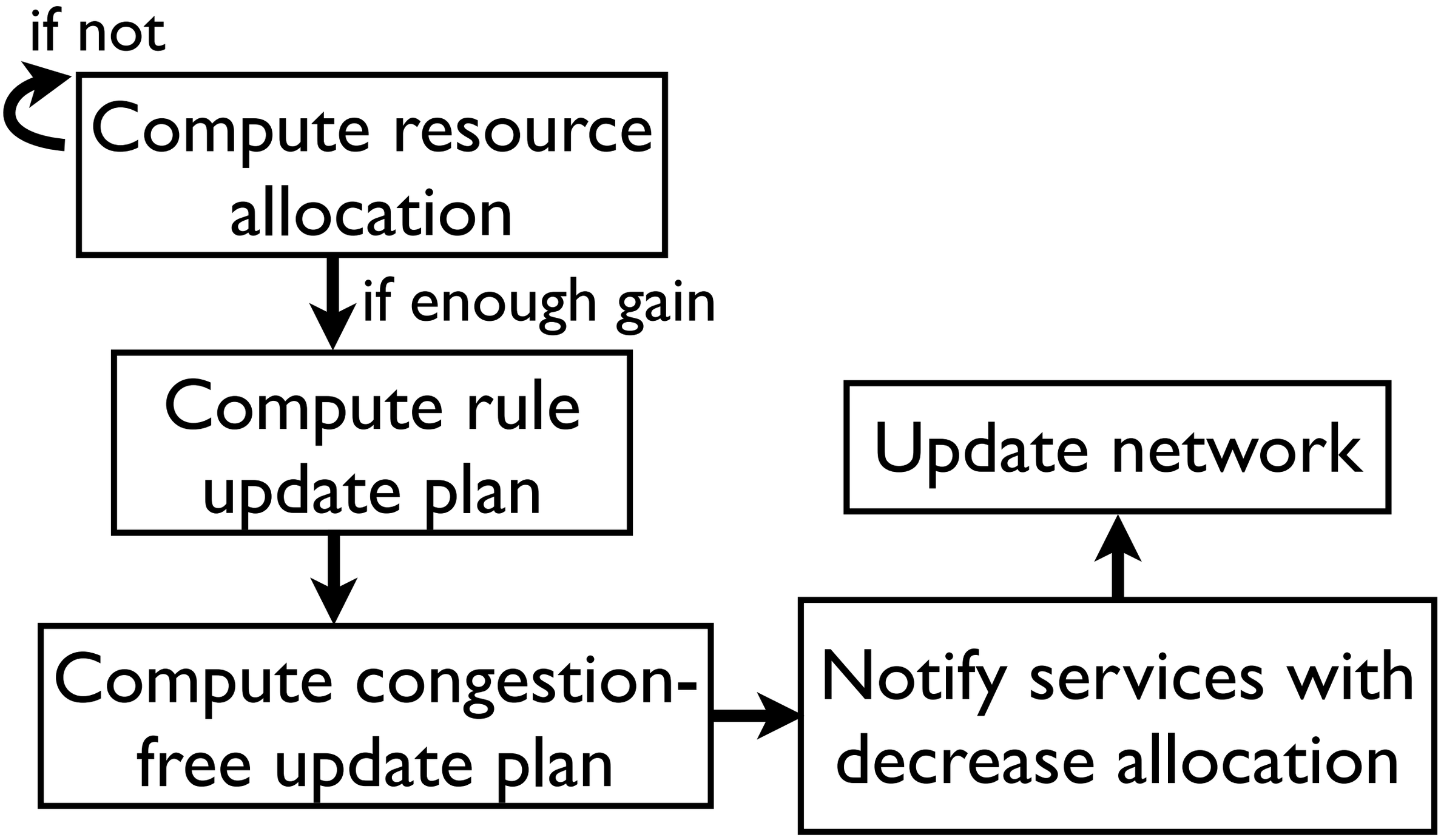
Overall workflow



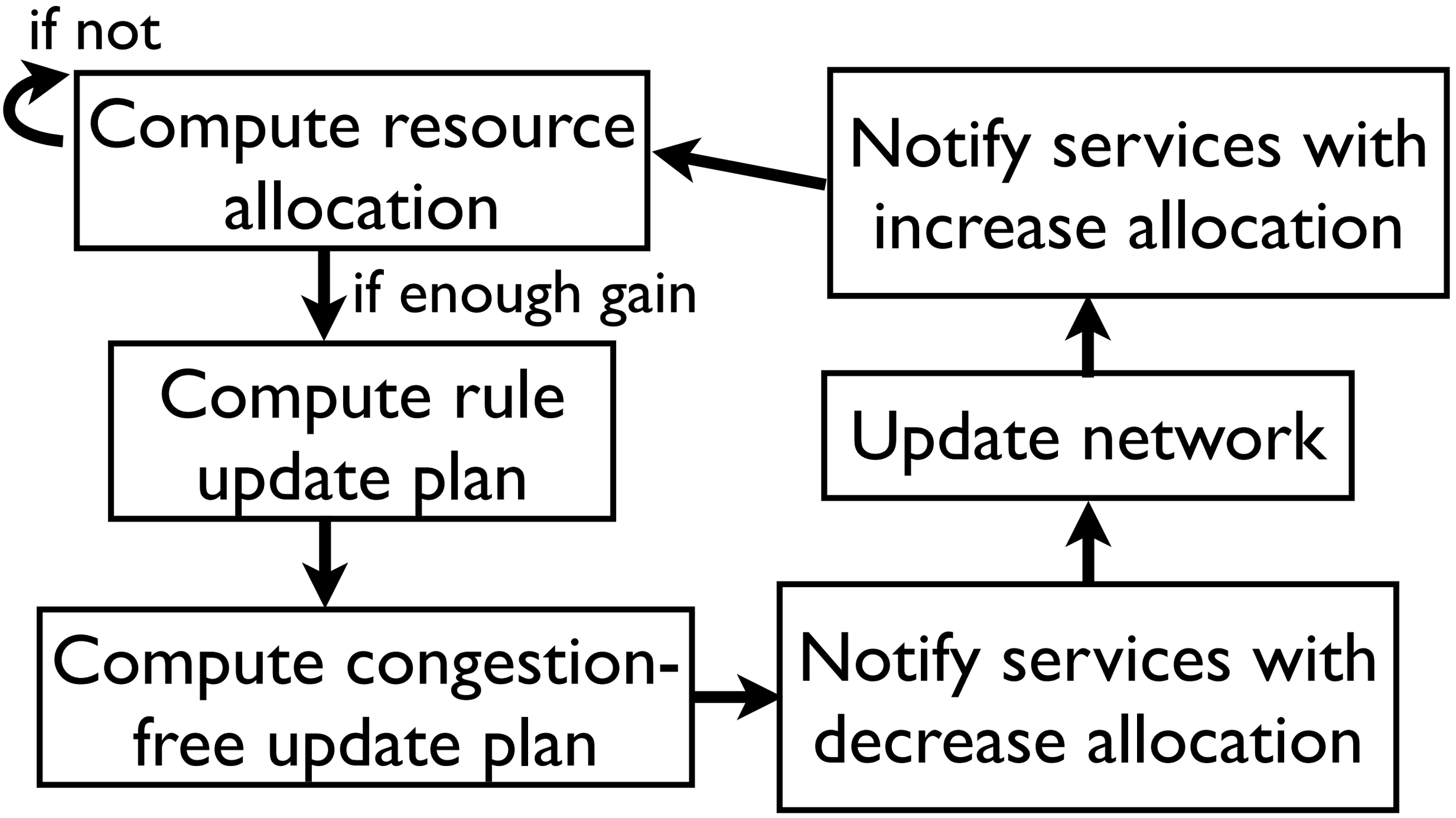
Overall workflow



Overall workflow



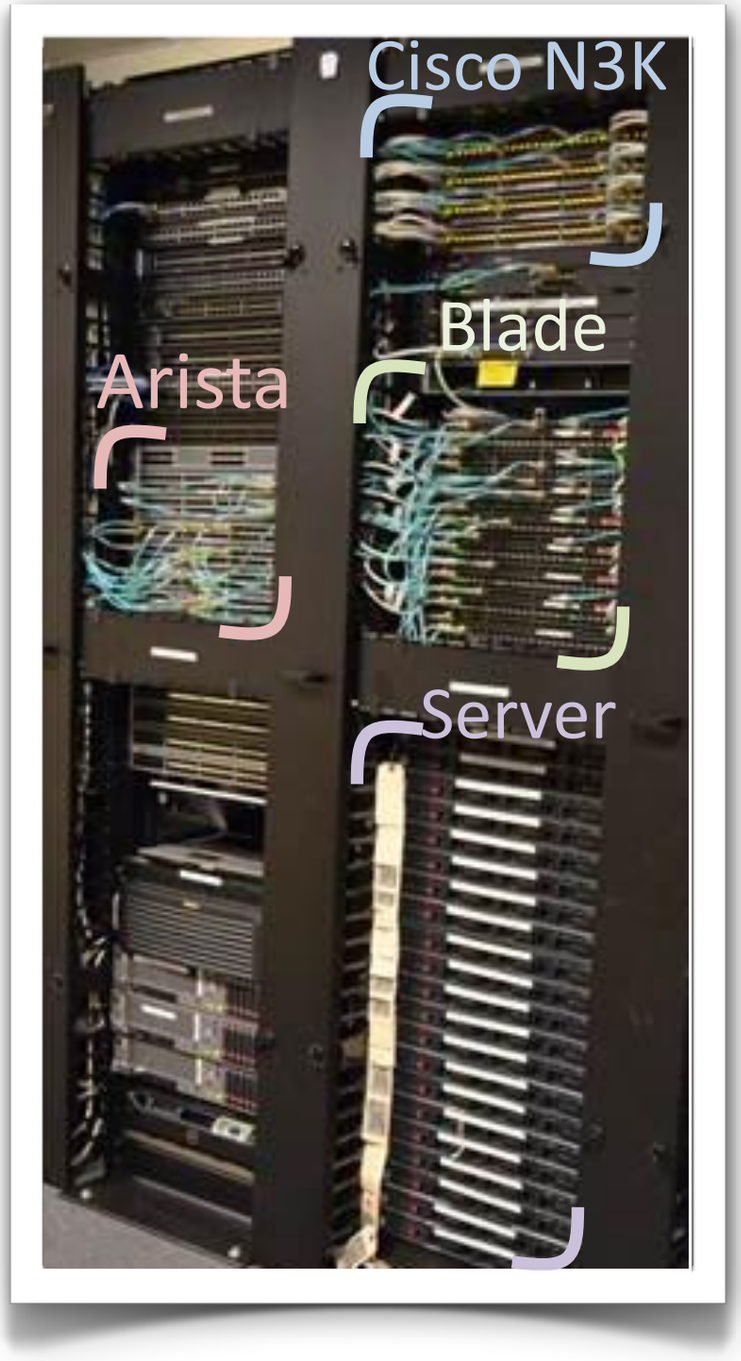
Overall workflow



Evaluation platforms

Prototype →

- 5 DCs across 3 continents;
  10 switches



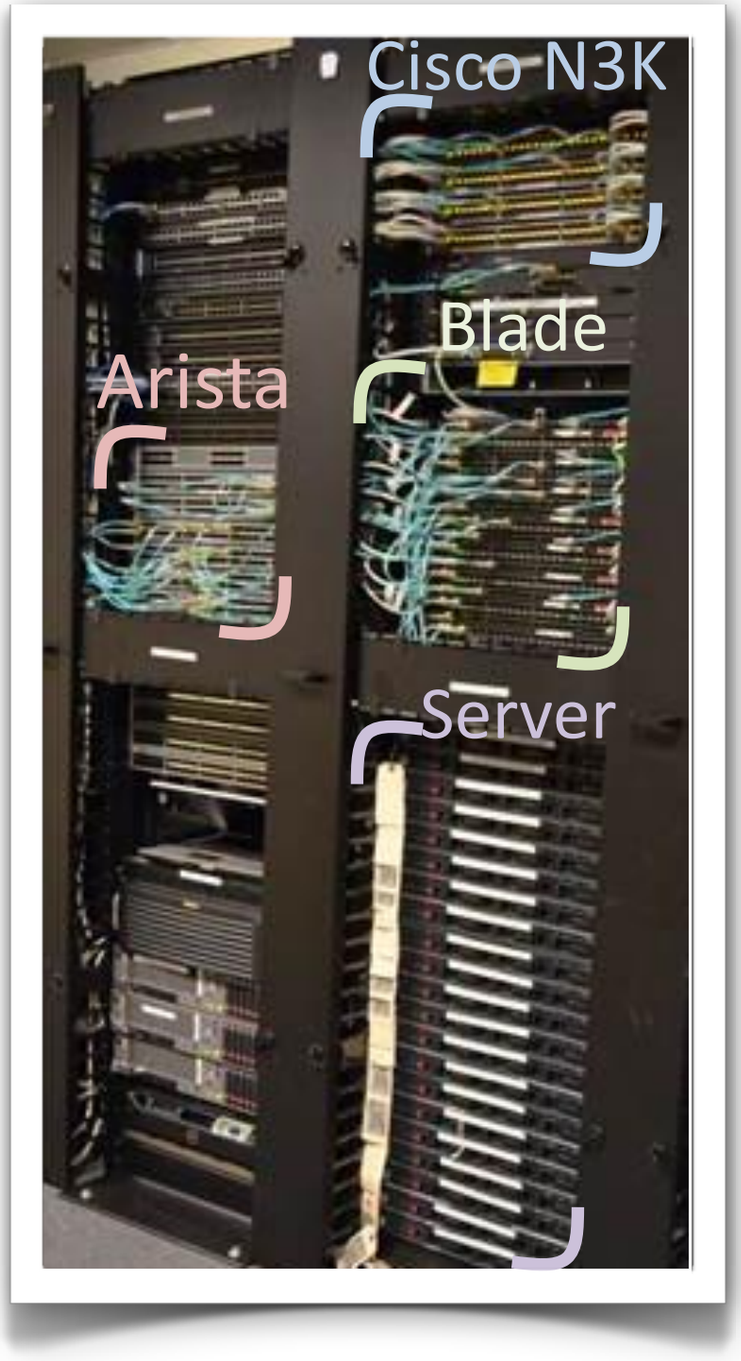
Evaluation platforms

Prototype

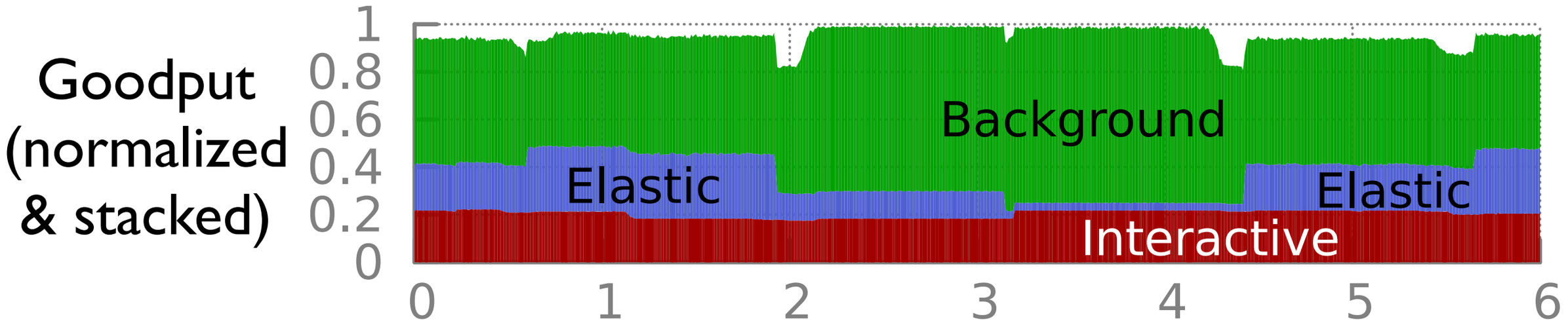
- 5 DCs across 3 continents; 10 switches

Data-driven evaluation

- 40+ DCs across 3 continents, 80+ switches
- G-scale: 12 DCs, 24 switches

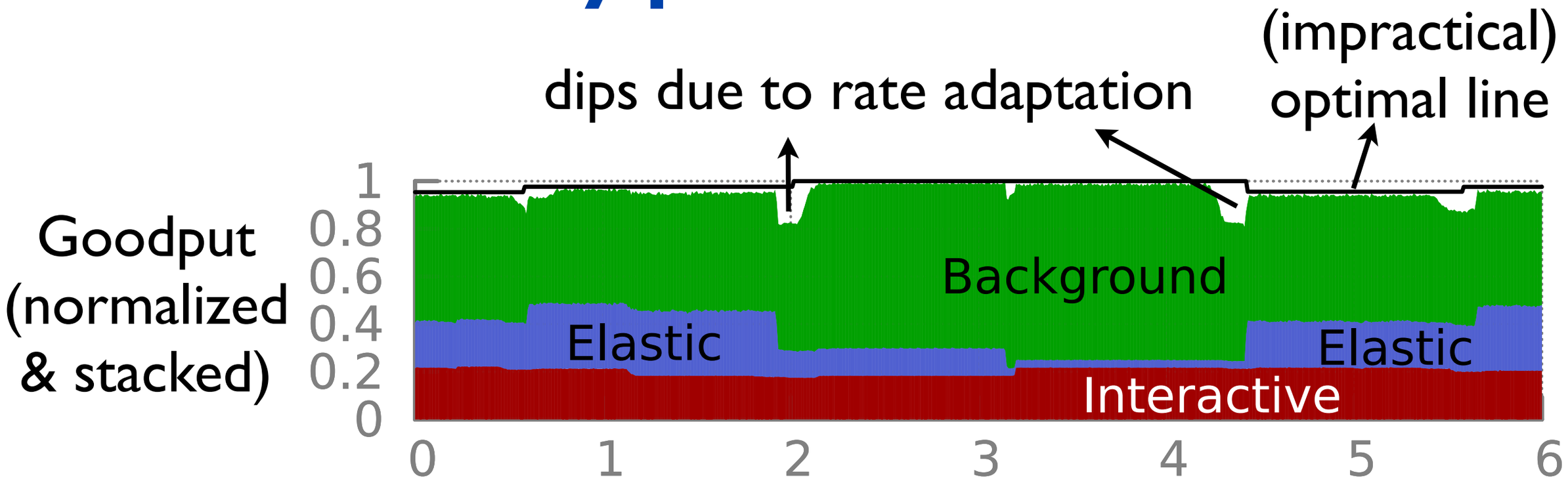


Prototype evaluation



Traffic: (∀DC-pair) 125 TCP flows per class

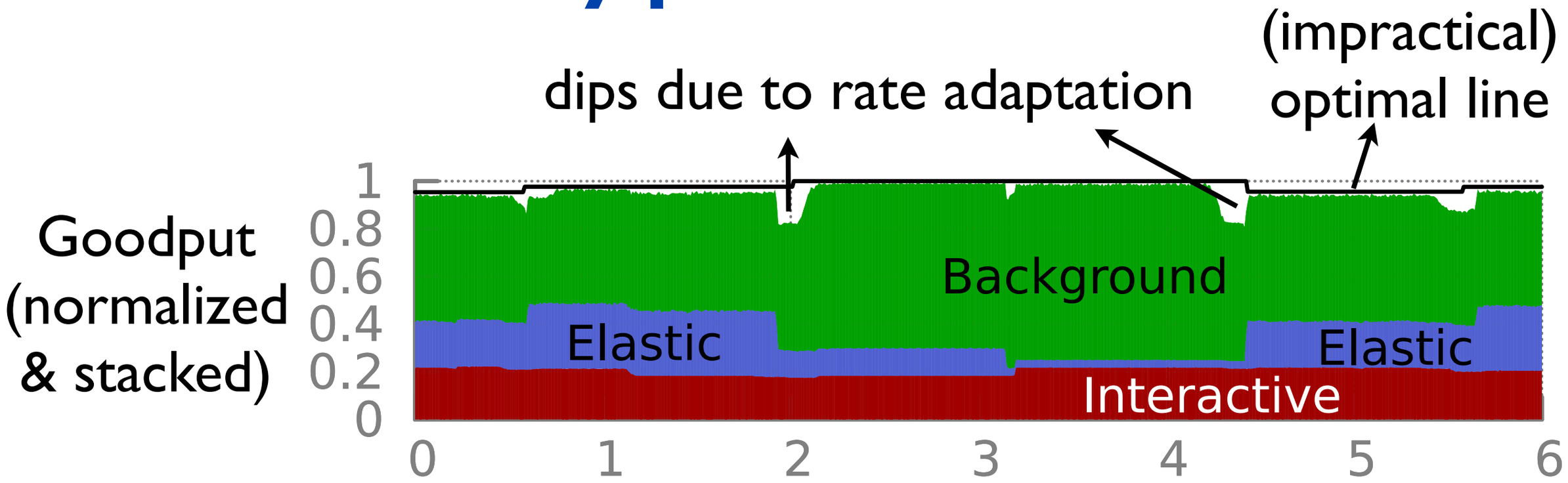
Prototype evaluation



Traffic: (∀DC-pair) 125 TCP flows per class

High utilization
SWAN's goodput:
98% of an optimal method

Prototype evaluation

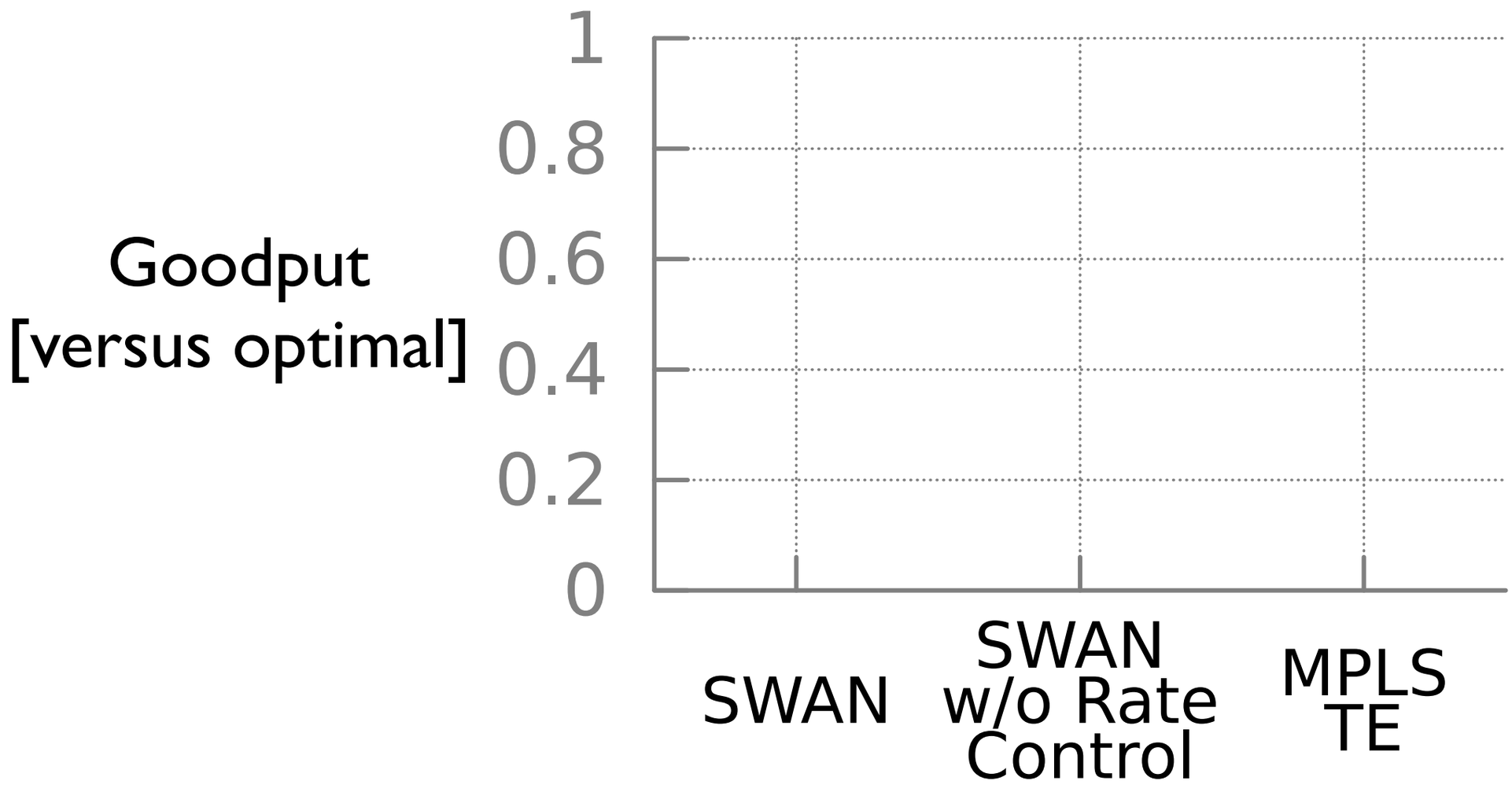


Traffic: (∀DC-pair) 125 TCP flows per class

High utilization
SWAN's goodput:
98% of an optimal method

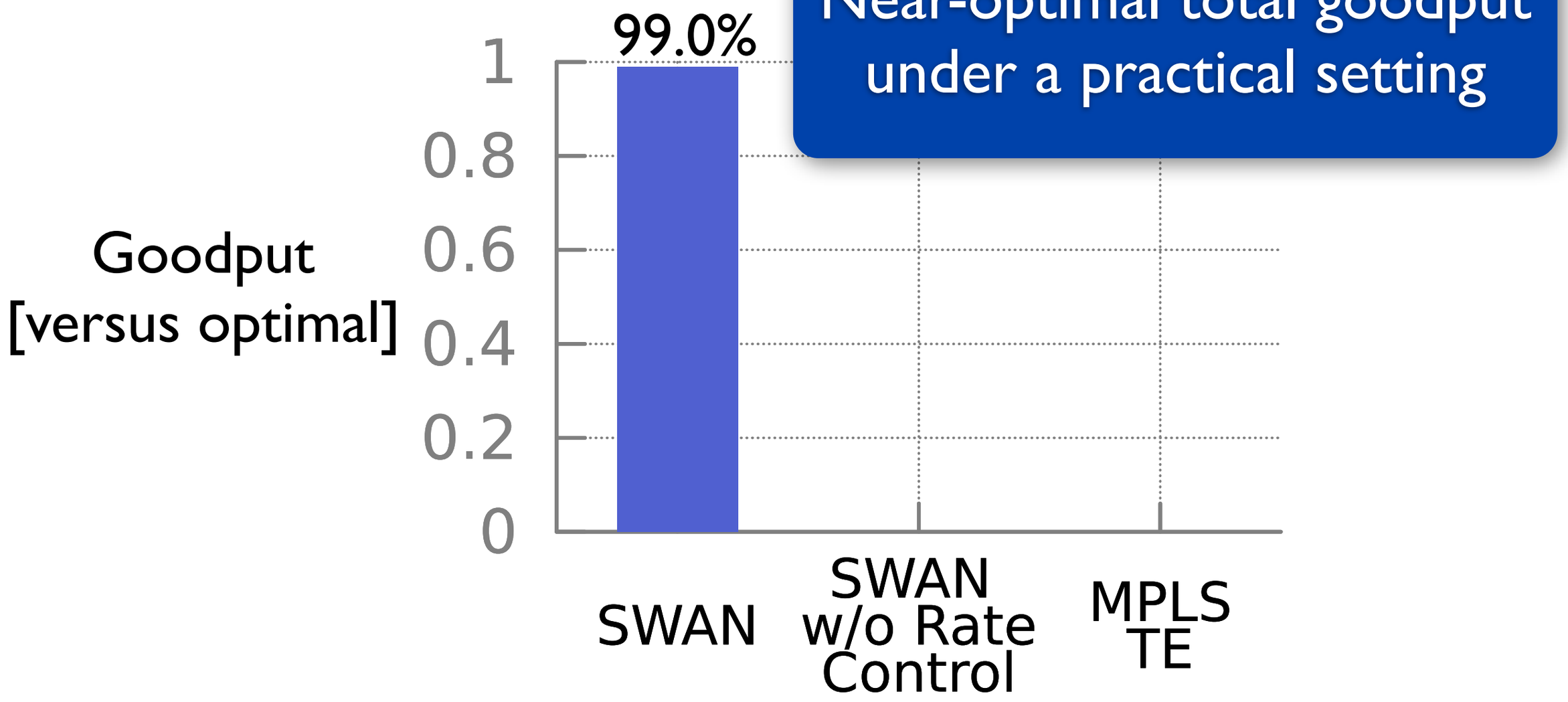
Flexible sharing
Interactive protected;
background rate-adapted

Data-driven evaluation of 40+ DCs



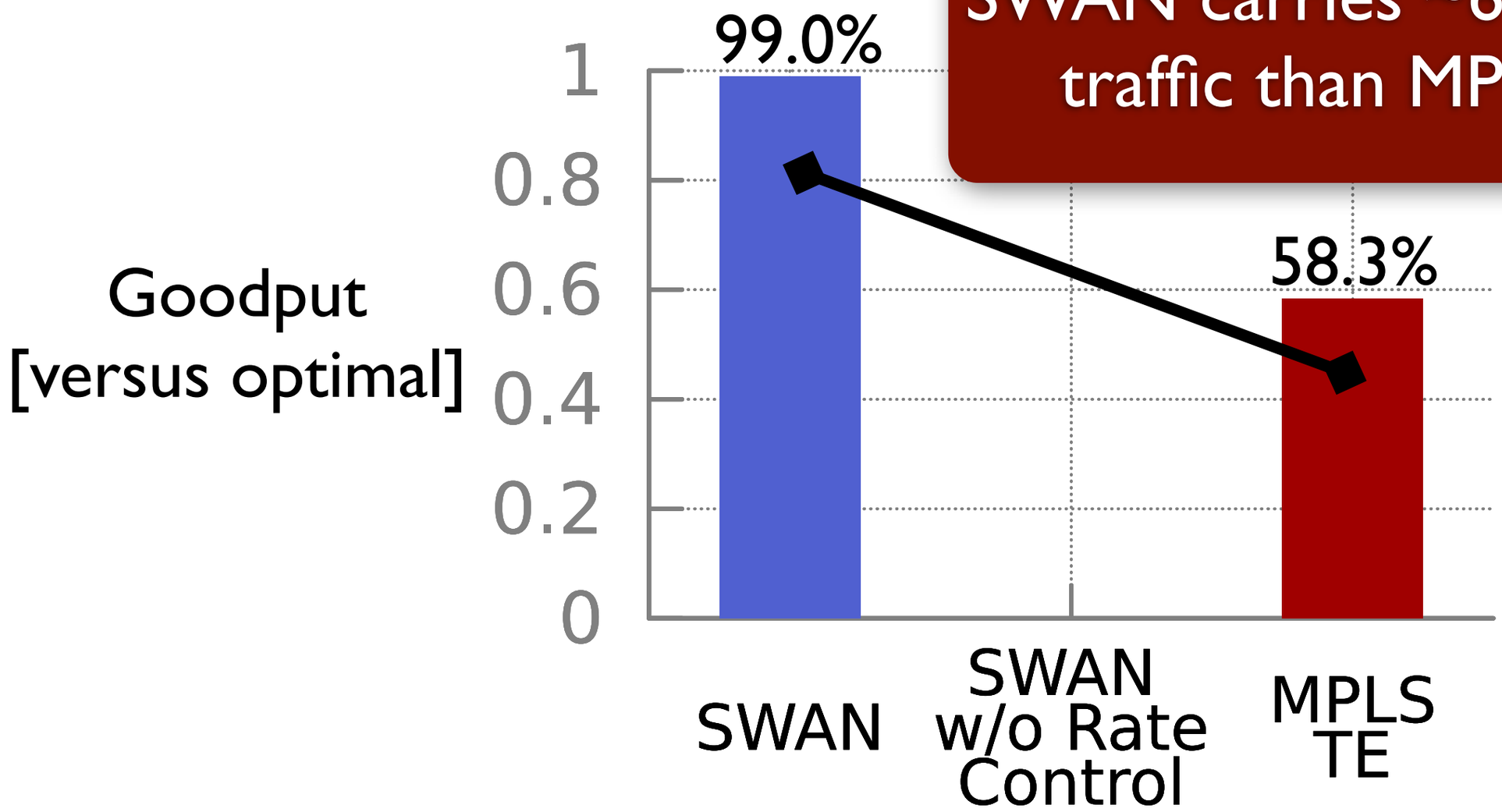
Data-driven evaluation of 40+ DCs

Near-optimal total goodput under a practical setting



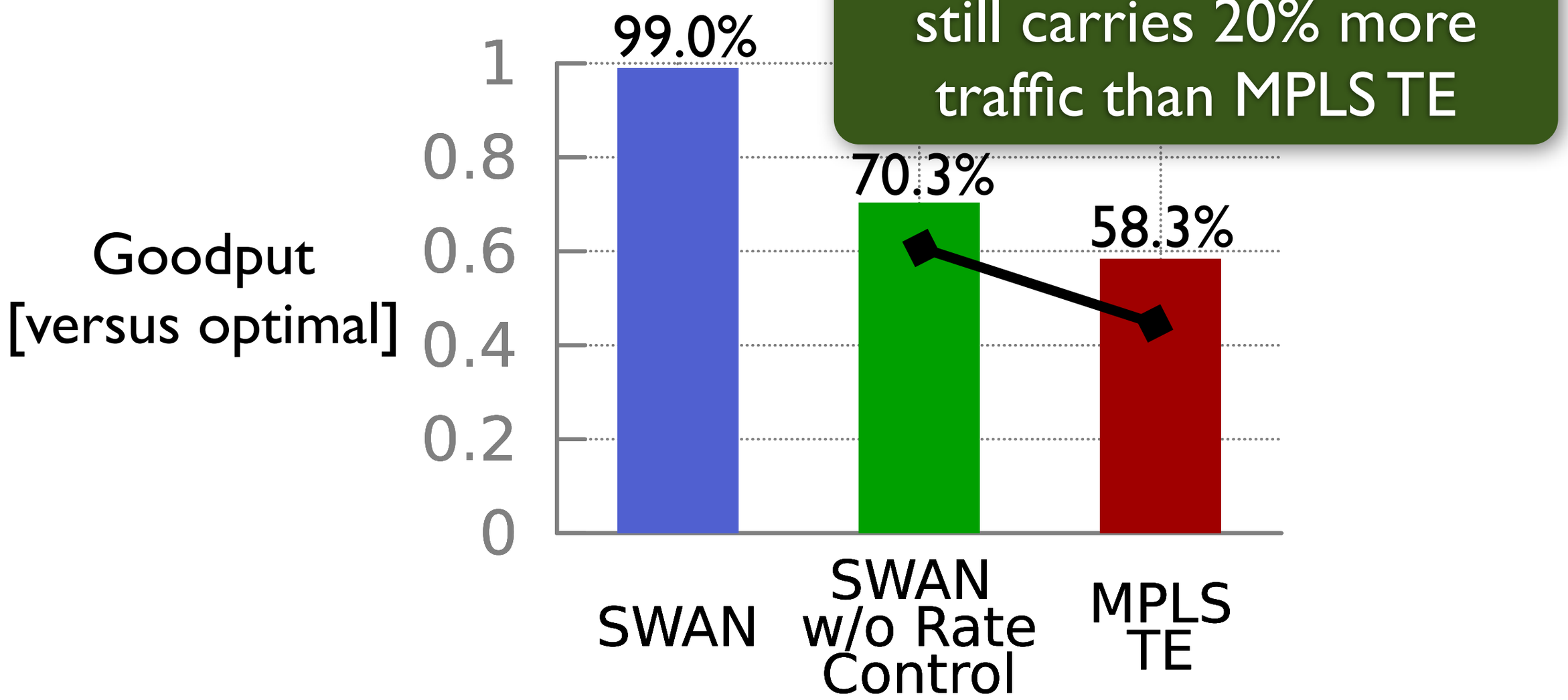
Data-driven evaluation of 40+ DCs

SWAN carries ~60% more traffic than MPLS-TE



Data-driven evaluation of 40+ DCs

SWAN w/o rate control still carries 20% more traffic than MPLS TE



SWAN : Software-driven *WAN*

SWAN: Software-driven *WAN*

✔ High utilization and flexible sharing via global rate and route coordination

SWAN: Software-driven *WAN*

- ✔ High utilization and flexible sharing via global rate and route coordination
- ✔ Scalable allocation via approximated max-min fairness

SWAN: Software-driven *WAN*

- ✔ High utilization and flexible sharing via global rate and route coordination
- ✔ Scalable allocation via approximated max-min fairness
- ✔ Congestion-free update in bounded stages

SWAN: Software-driven *WAN*

✔ High utilization and flexible sharing via global rate and route coordination

✔ Scalable allocation via approximated max-min fairness

✔ Congestion-free update in bounded stages

✔ Using commodity switches with limited memory

Conclusion

Achieving high utilization itself is easy, but coupling it with flexible sharing and change management is hard

Conclusion

Achieving high utilization itself is easy, but coupling it with flexible sharing and change management is hard

Approximating max-min fairness with low computational time

Conclusion

Achieving high utilization itself is easy, but coupling it with flexible sharing and change management is hard

Approximating max-min fairness with low computational time

Keeping scratch capacity of links and switch memory to enable quick transitions

Thanks!

on the job market!

Chi-Yao Hong (UIUC)

Srikanth Kandula

Ratul Mahajan

Ming Zhang

Vijay Gill

Mohan Nanduri

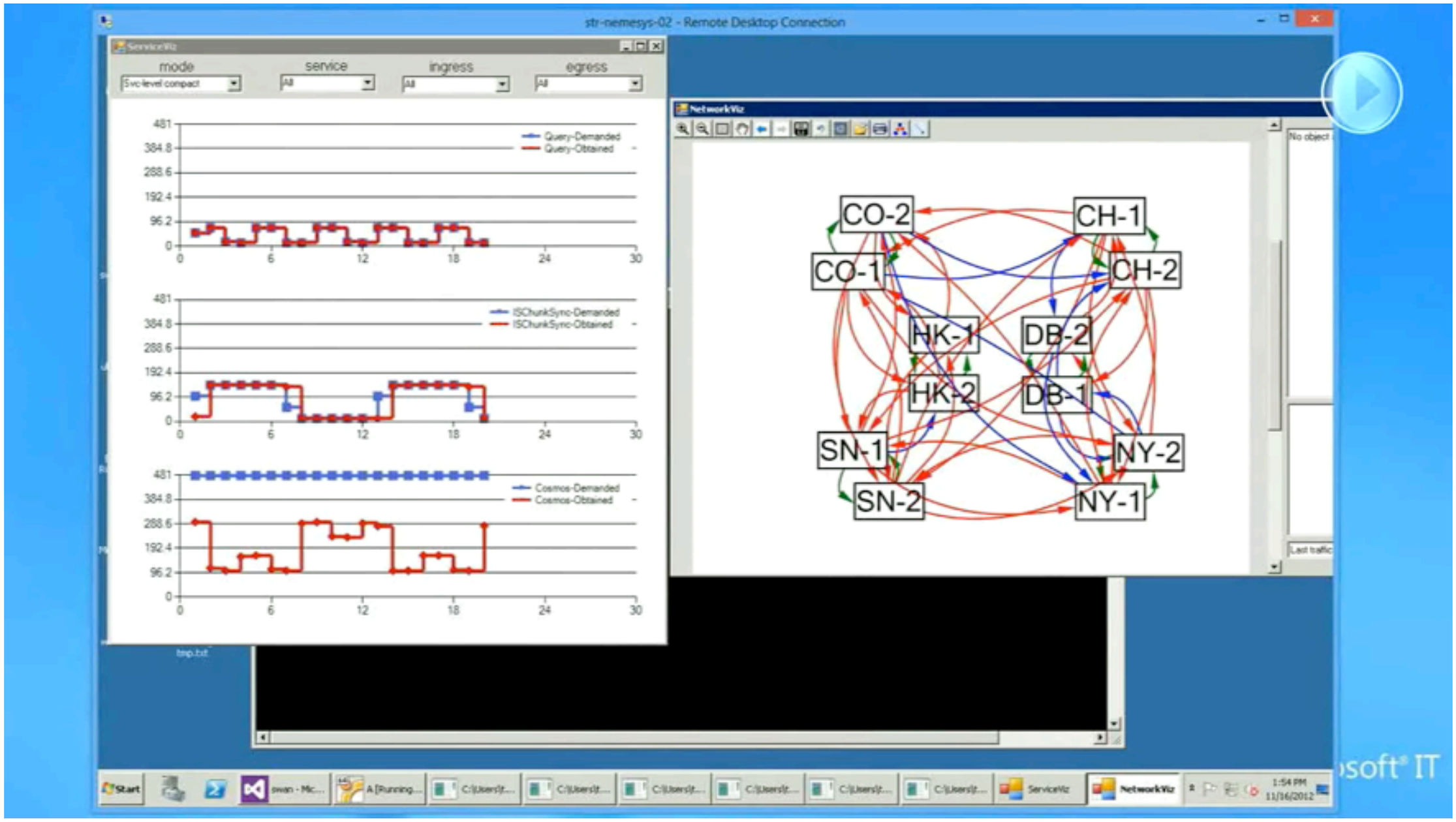
Roger Wattenhofer

Microsoft

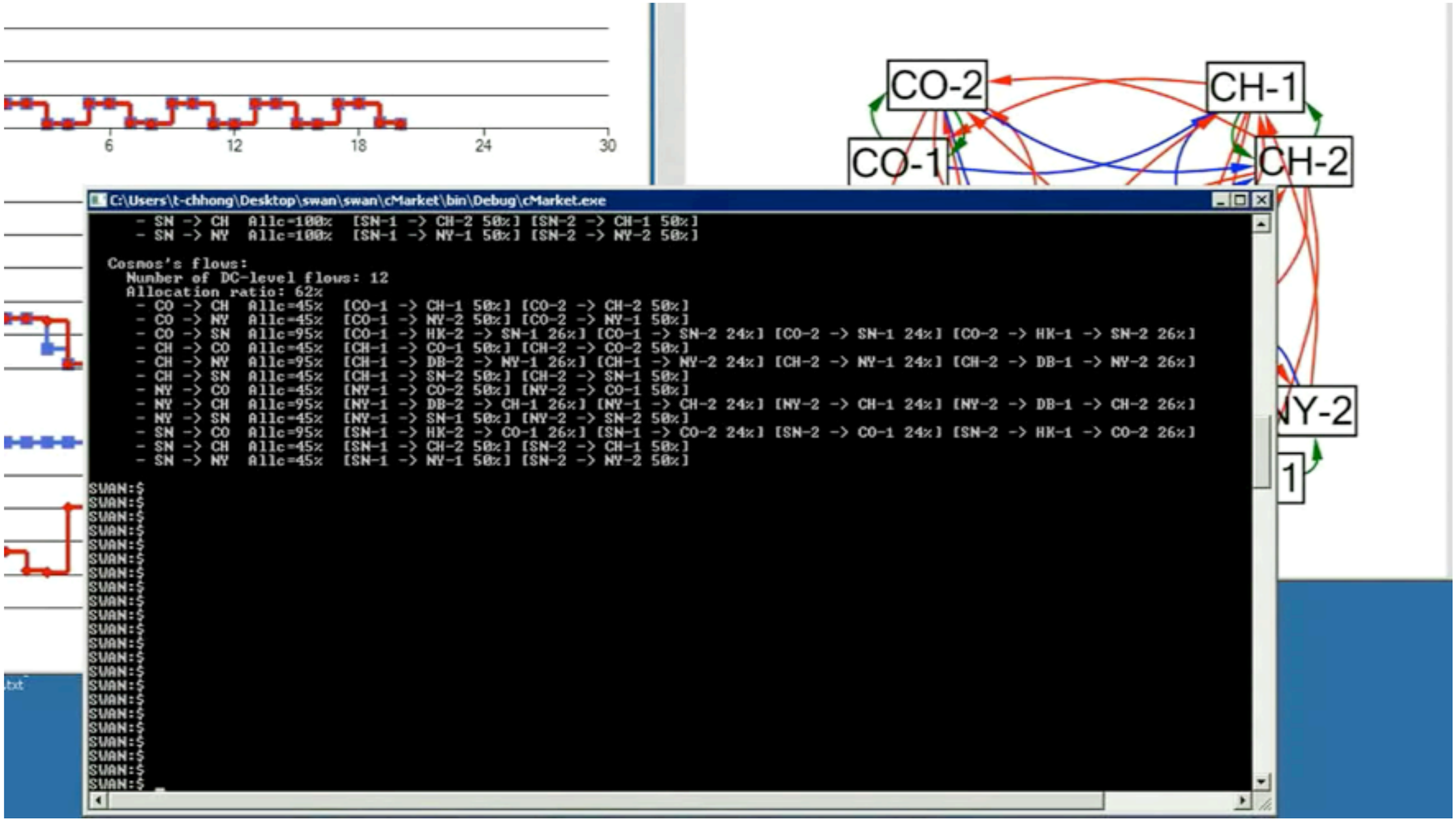
SWAN versus B4

	SWAN	B4
high utilization	yes	yes
scalable rate and route computation	bounded error	heuristic
congestion-free update	in bounded steps	no
using commodity switches with limited # forwarding rules	yes	no

Demo Video: SWAN achieves high utilization



Demo Video: SWAN provides flexible sharing



Failure handling

Link and switch failures

- Network agent notifies SWAN controller
- SWAN controller performs one-time global recomputation

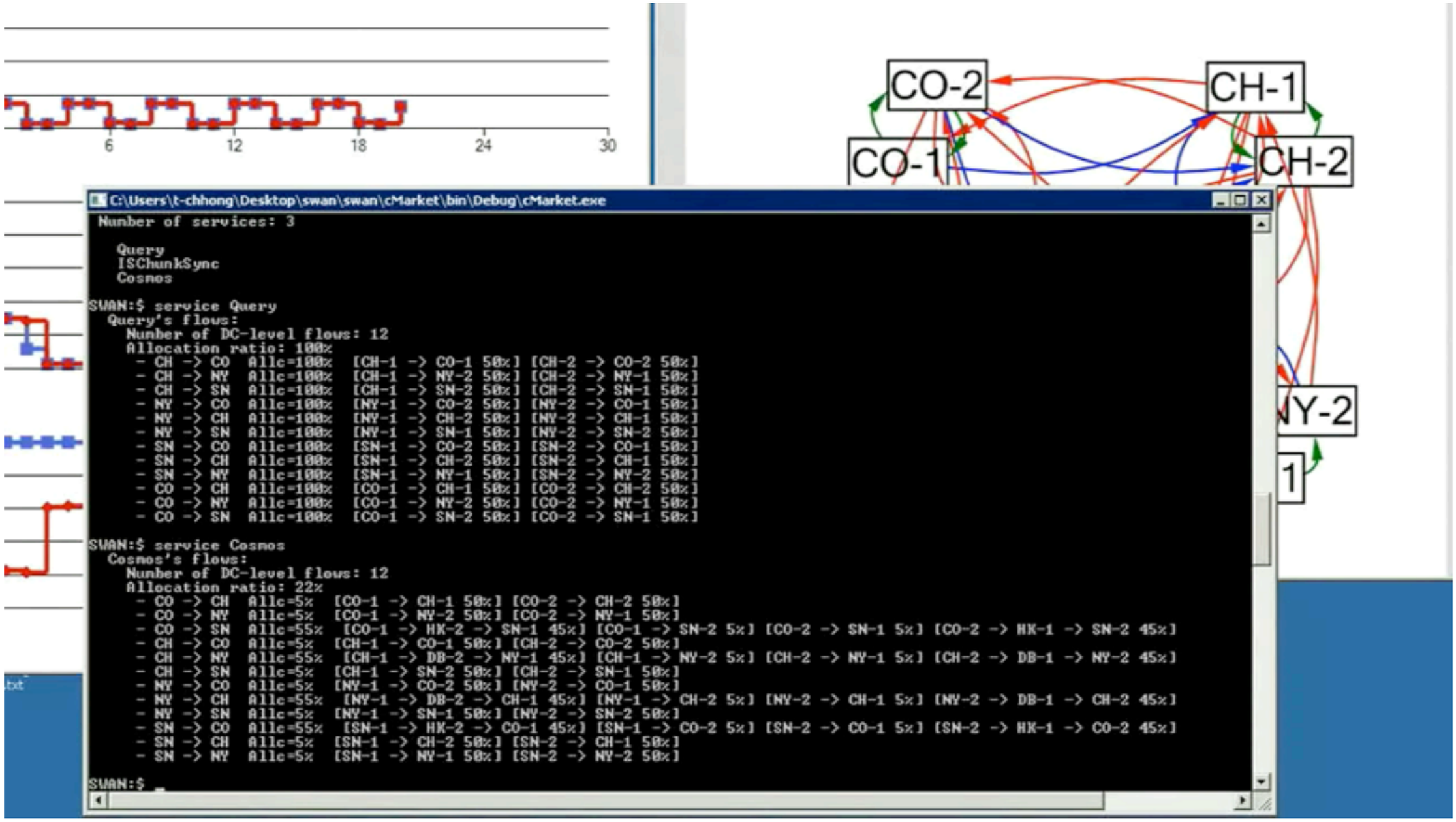
Controller crashes

- Run stateless backup instances
- During the recovery, data plane will still operate

Controller bugs

- Work in progress

Demo Video: SWAN handles link failures gracefully



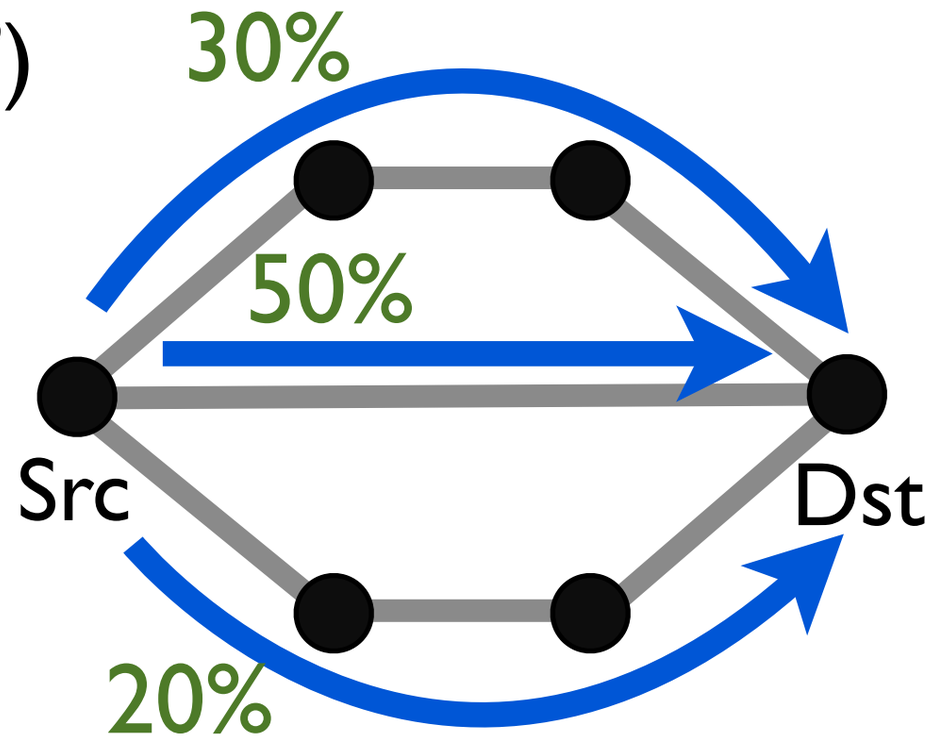
SWAN controller

Global allocation at {SrcDC, DstDC, ServicePriority}-level

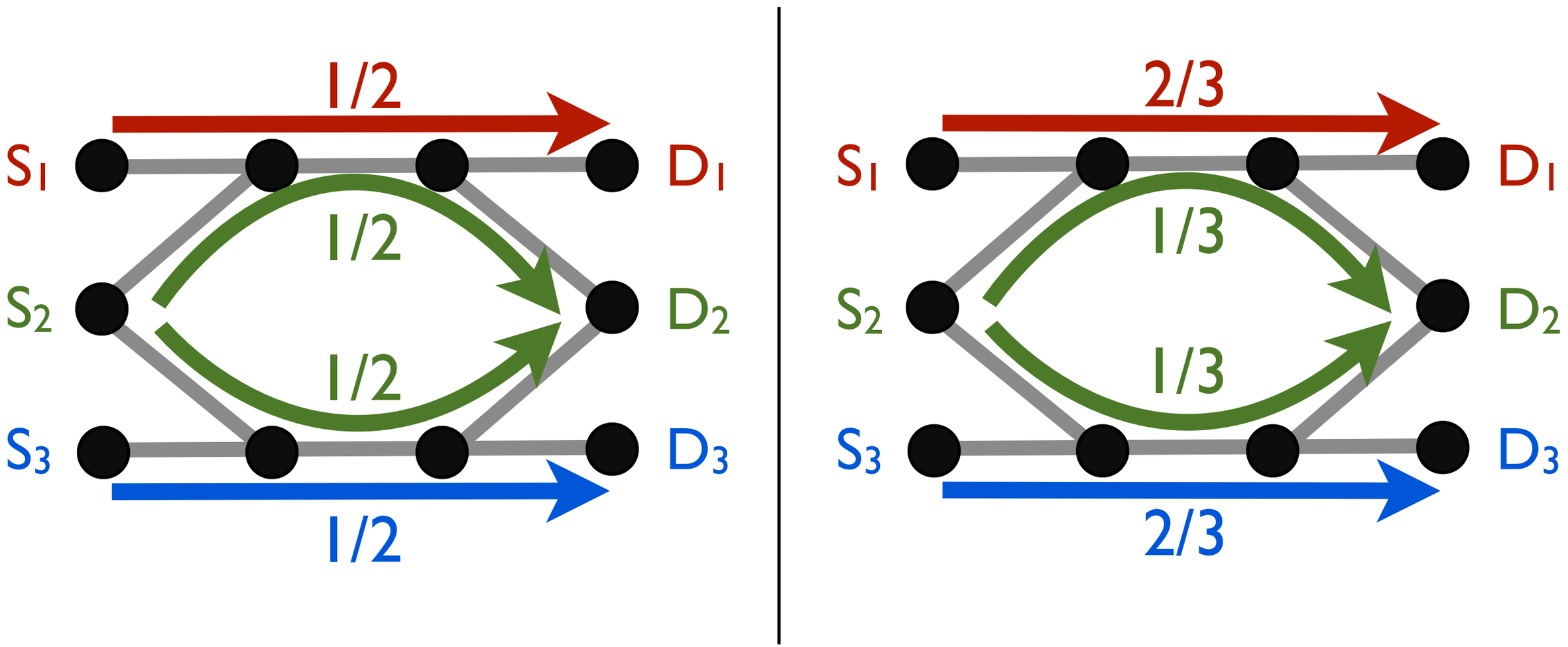
- support a few priorities (e.g., background < elastic < interactive)
- map flows to priority queues at switches (via DSCP bits)

Label-based forwarding ("tunnels")

- by tagging VLAN IDs
- SWAN controller globally computes how to split traffic at ingress switches



Link-level fairness != network-wide fairness



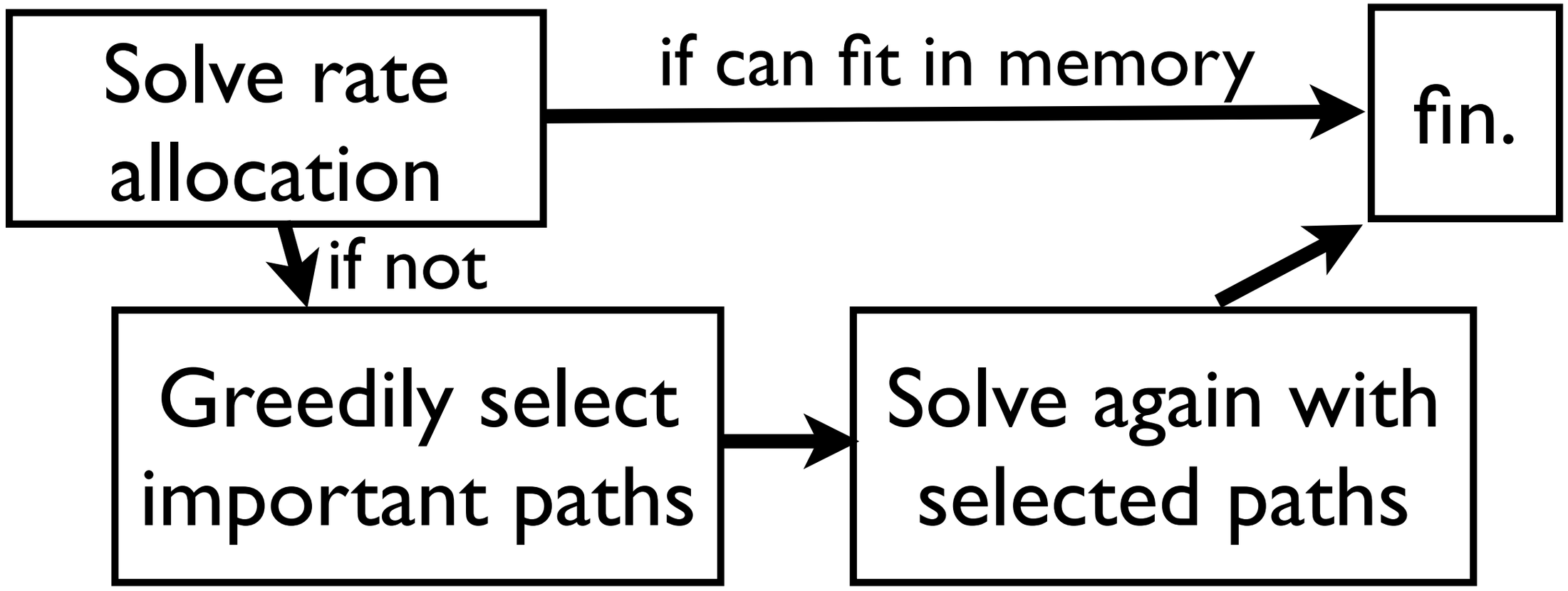
Time for network update

How much stretch capacity is needed?

	s	max steps		99th pctl.	goodput
	50%	1		1	79%
	30%	3	>	2	91%
✔	10%	9	>>	3	100%
	0%	∞	>>	6	100%

[data-driven evaluation]

Our heuristic: dynamic path selection

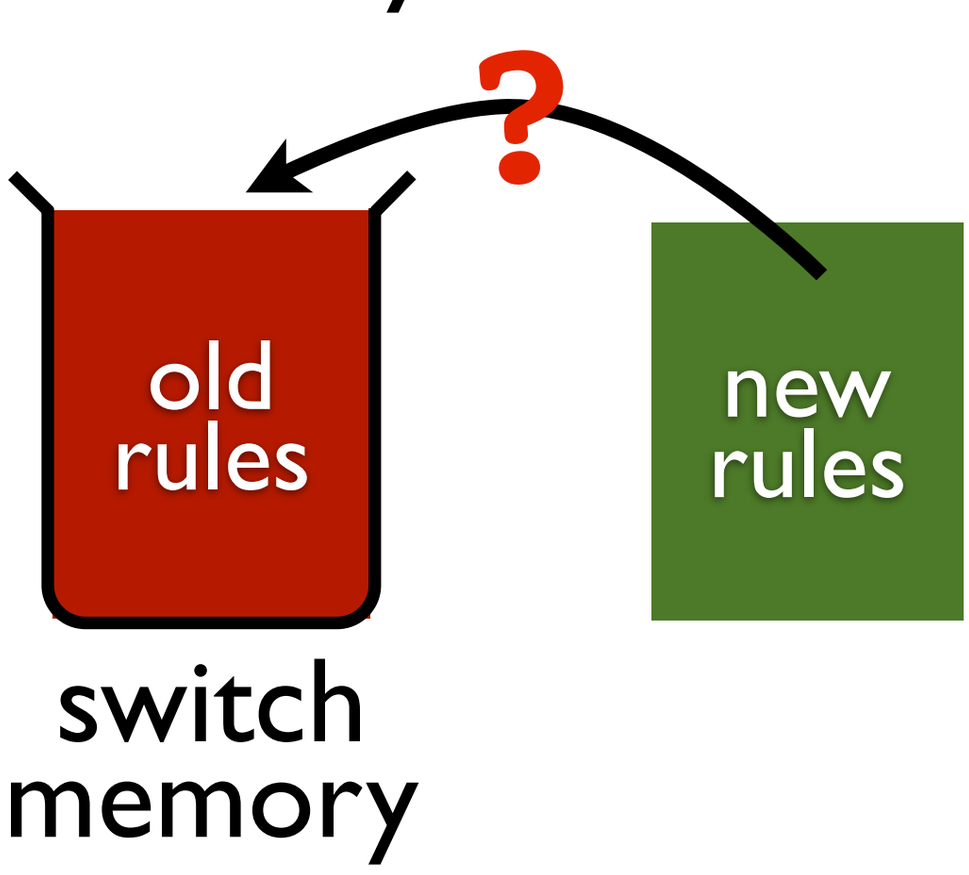


Using 10x fewer paths than static k-shortest path routing

Rule update with memory constraints

Option #1:

Fully utilize all the switch memory

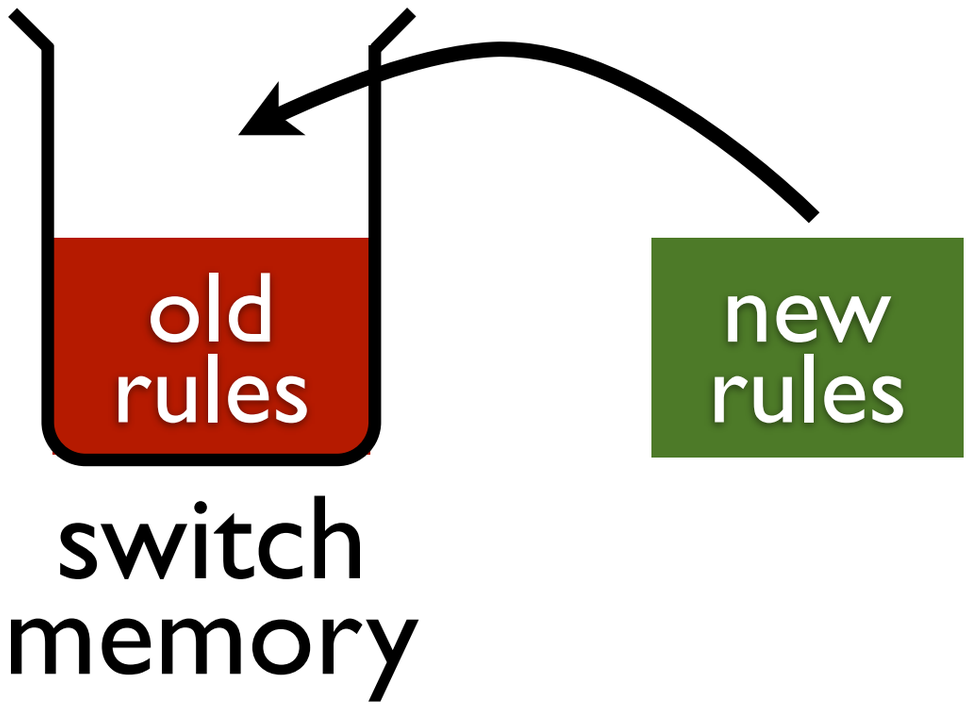


Rule update may disrupt traffic

Rule update with memory constraints

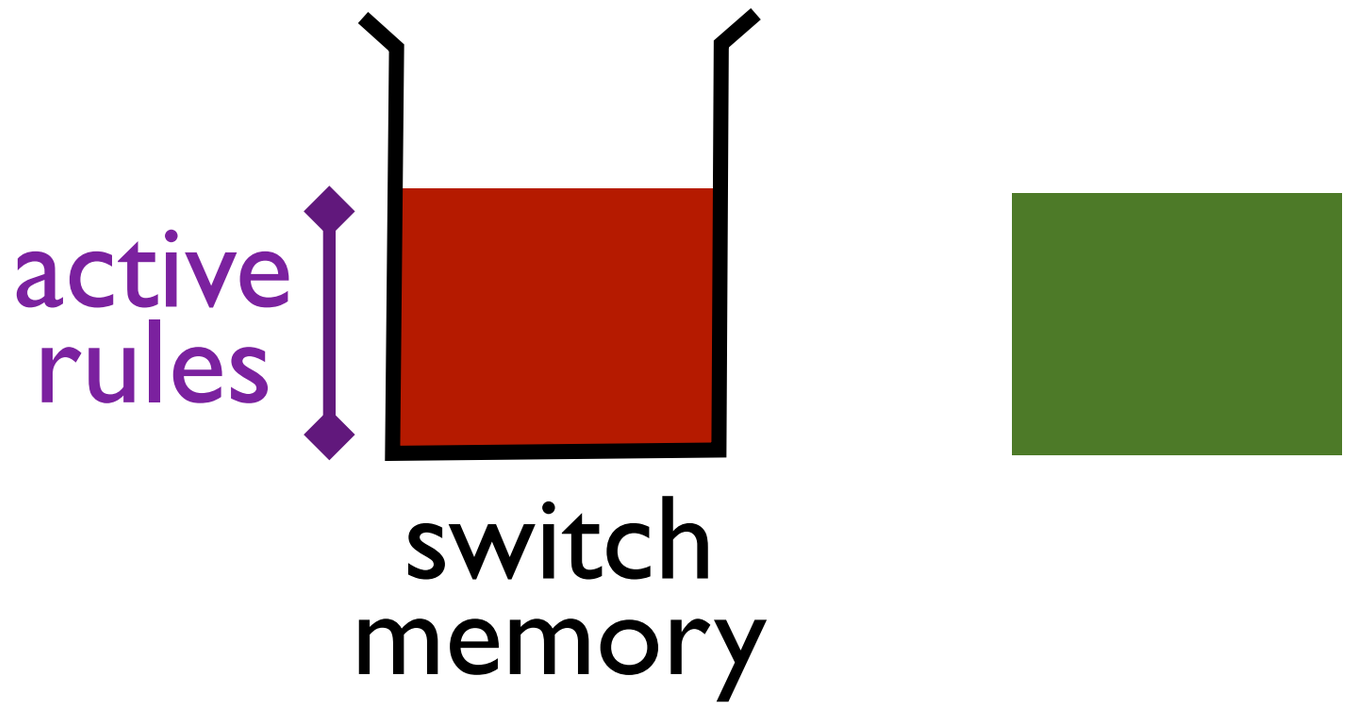
Option #2:

Leave 50% slack [Reitblatt et al.; SIGCOMM'12]

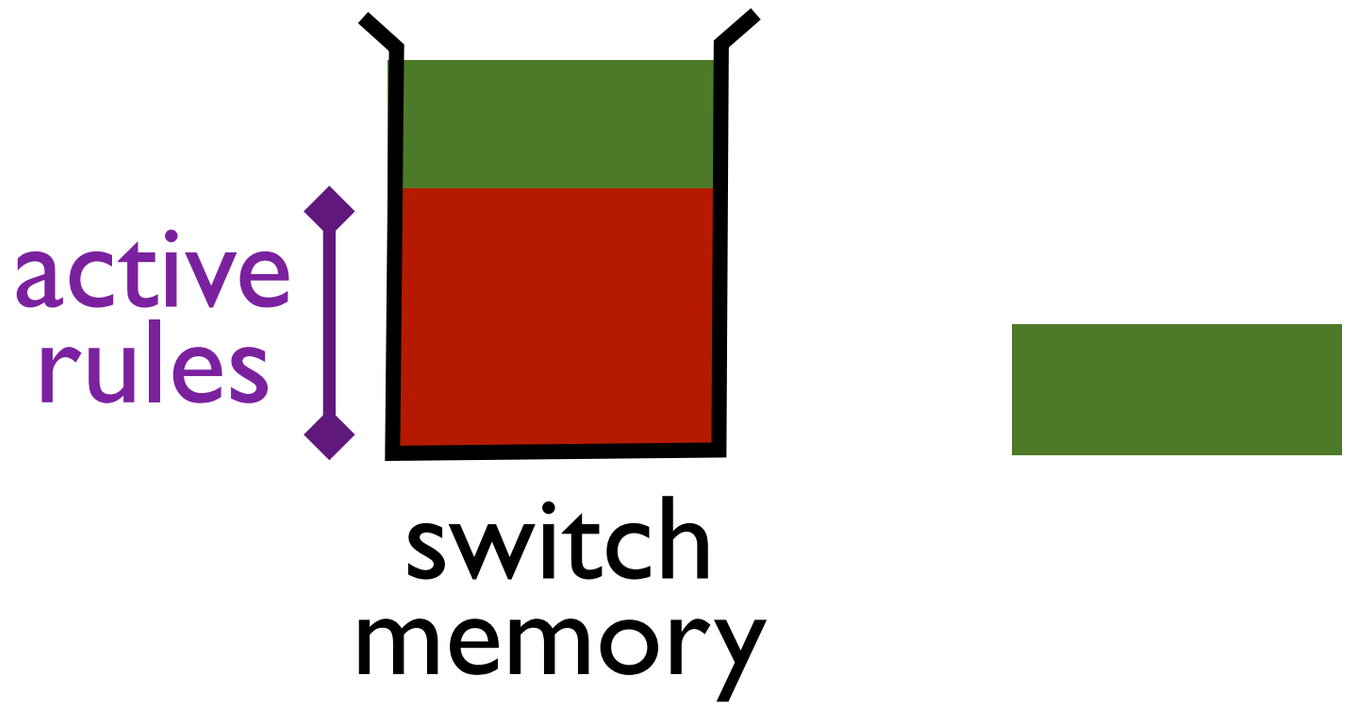


Waste a half switch memory

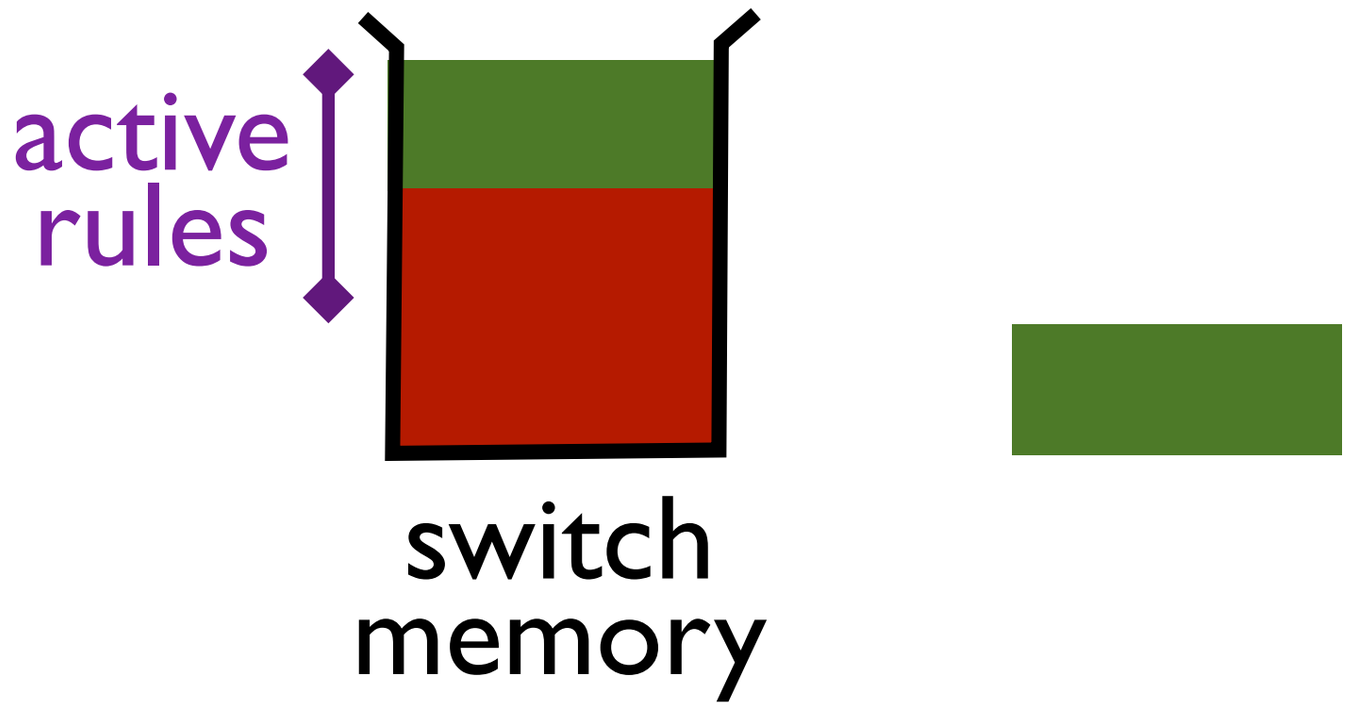
Multi-stage rule update



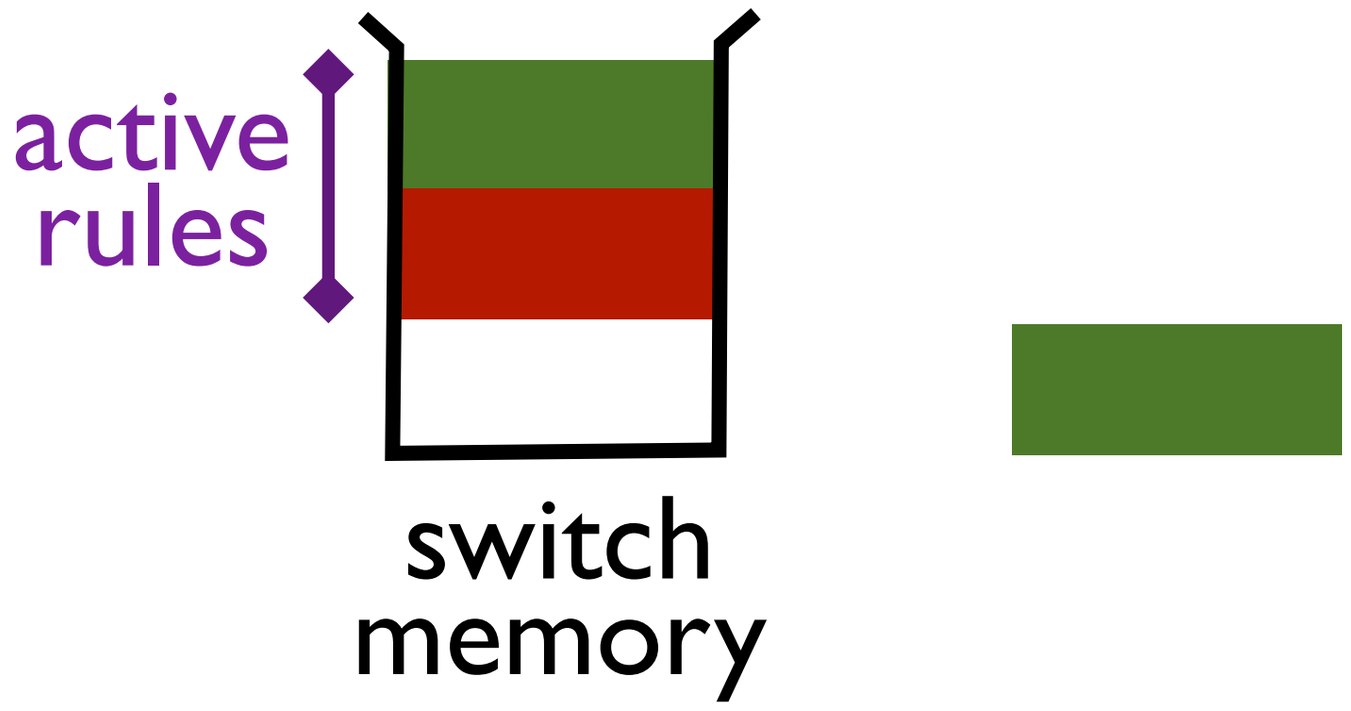
Multi-stage rule update



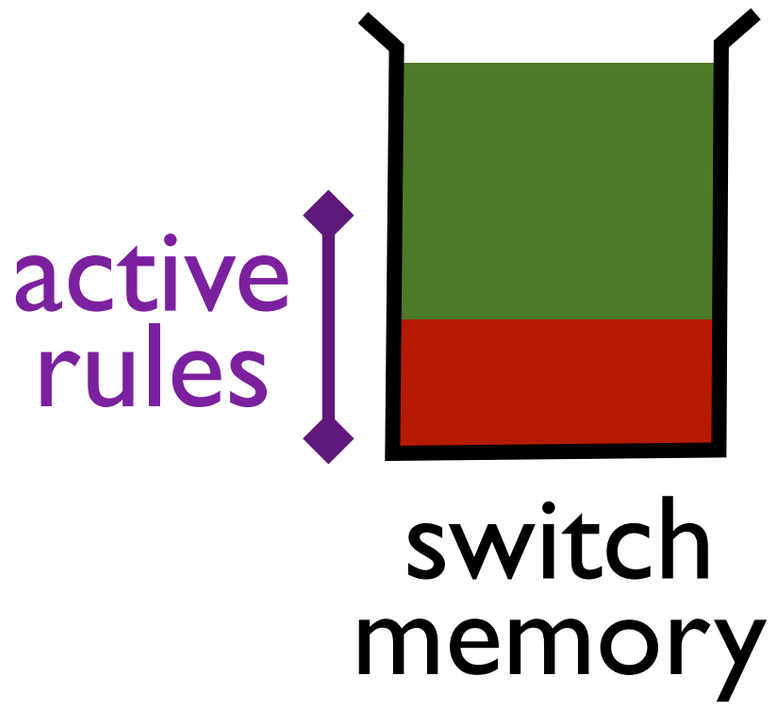
Multi-stage rule update



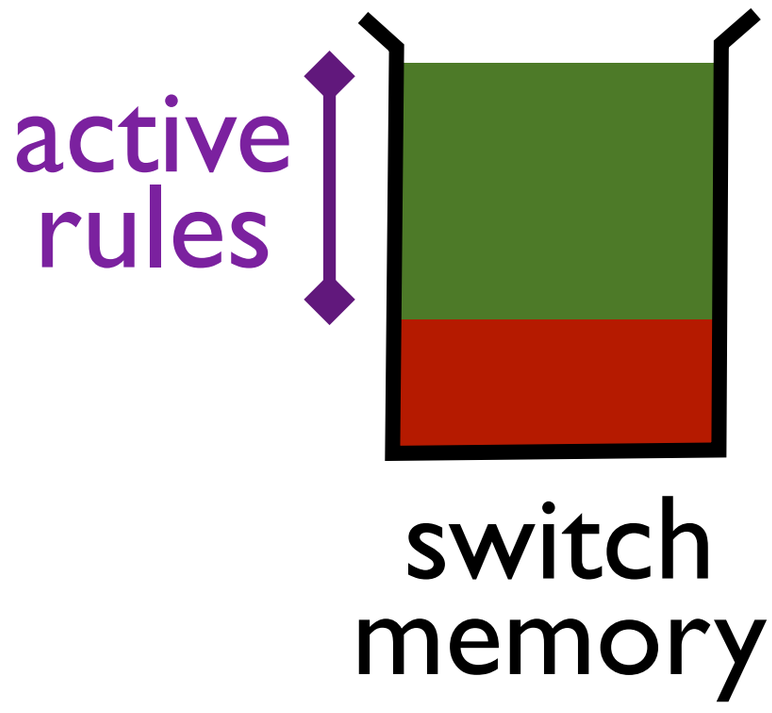
Multi-stage rule update



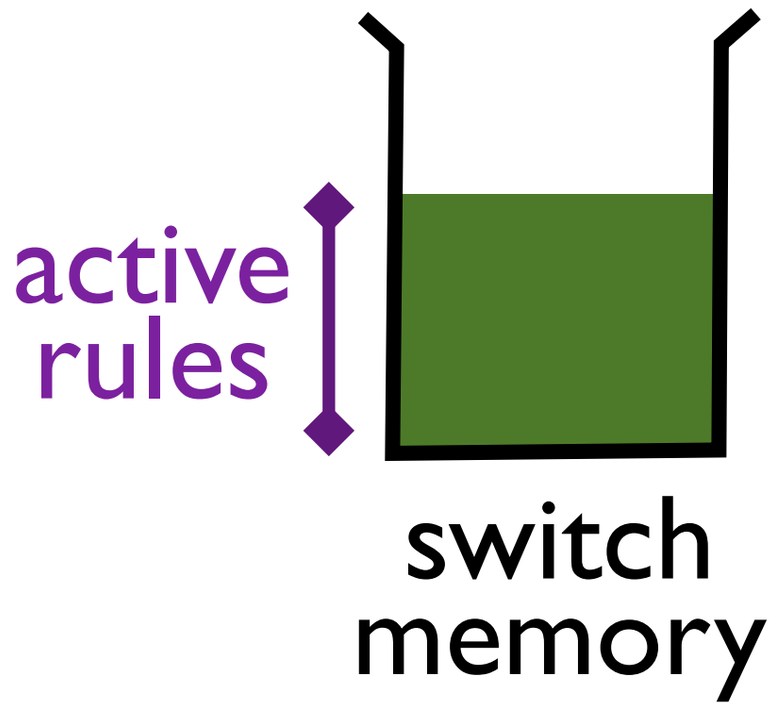
Multi-stage rule update



Multi-stage rule update

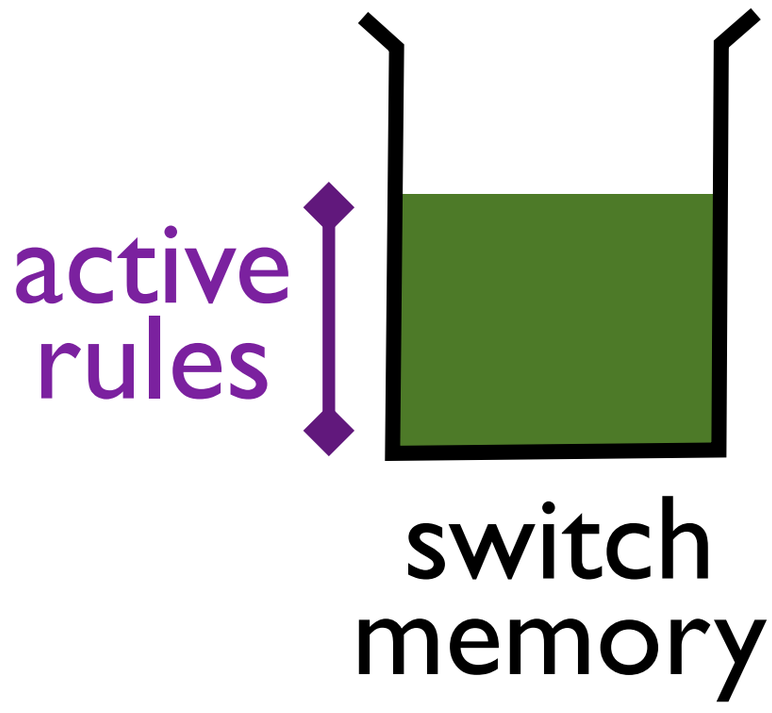


Multi-stage rule update



Multi-stage rule update

# stages bound:
f(memory slack)



Multi-stage rule update

# stages bound:
f(memory slack)

When slack=10%,
2 stages for 95% of time