Network Updates

Part 3, Chapter 6

Roger Wattenhofer

Overview

- Software-Defined Networking
- Loop-Free Updates
- Consistent Updates
- Bandwidth
  - Maximization
  - Fairness
  - Updates

Network Updates

- The Internet: Designed for selfish participants
  - Often inefficient (low utilization of links), but robust

- But what happens if the WAN is controlled by a single entity?
  - Examples: Microsoft & Amazon & Google …
  - They spend hundreds of millions of dollars per year

Software-Defined Networking

- Possible solution: **S**oftware-**D**efined **N**etworking (**SDN**s)

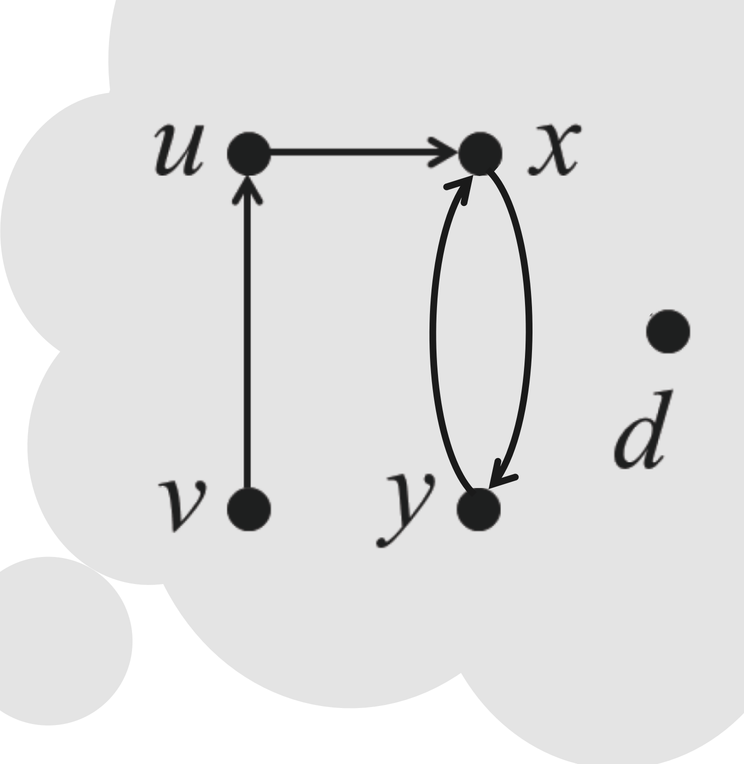
- General Idea: Separate data & control plane in a network
- Centralized controller updates networks rules for optimization
  - Controller (*control plane*) updates the switches/routers (*data plane*)

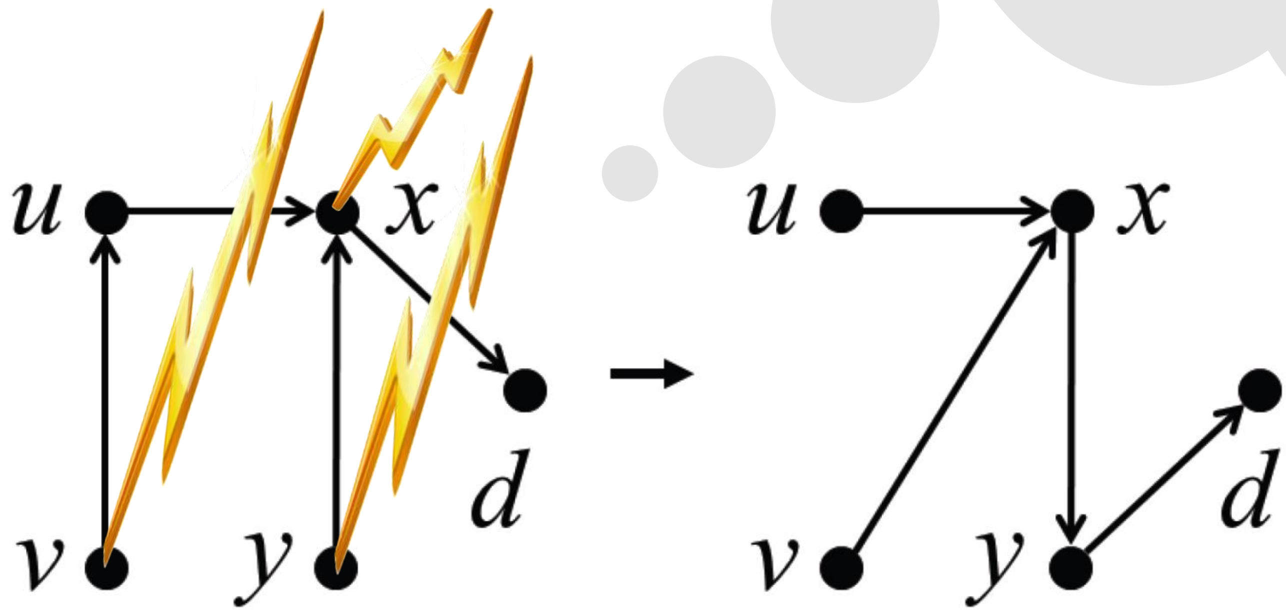
Virtual Services ⟷ Controller ⟷ Physical Network

- Centralized controller implemented with replication, e.g. Paxos

Example

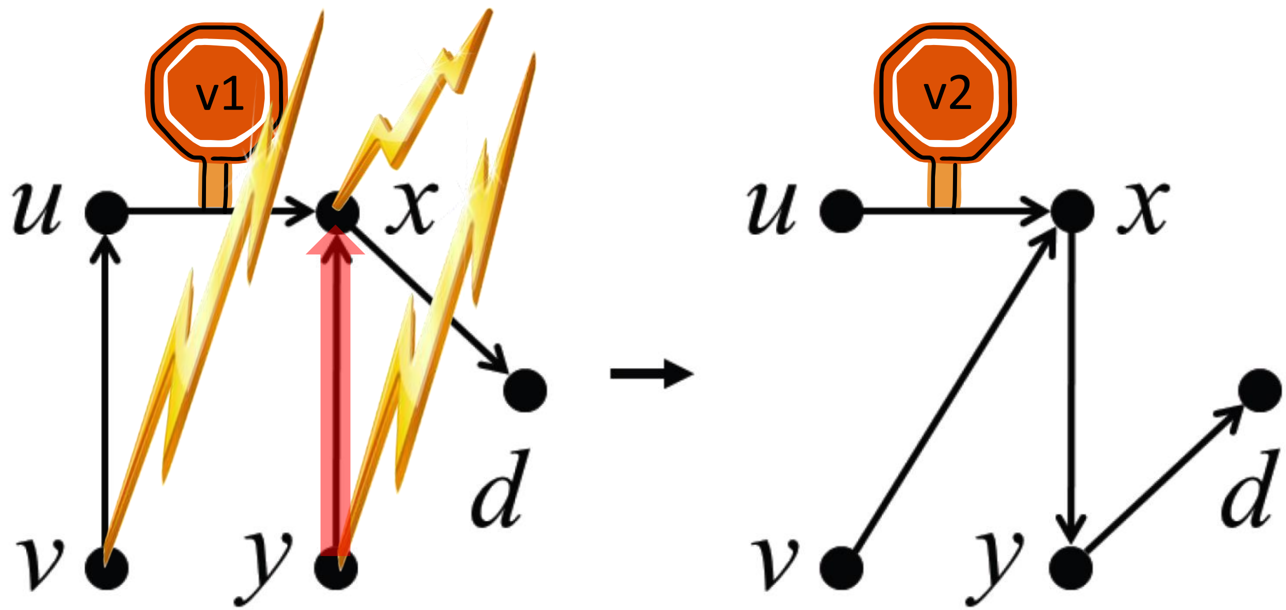
SDN Controller





Example

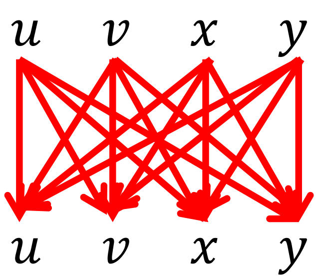
SDN Controller



Dependencies

Version Numbers 

“Better” Solution



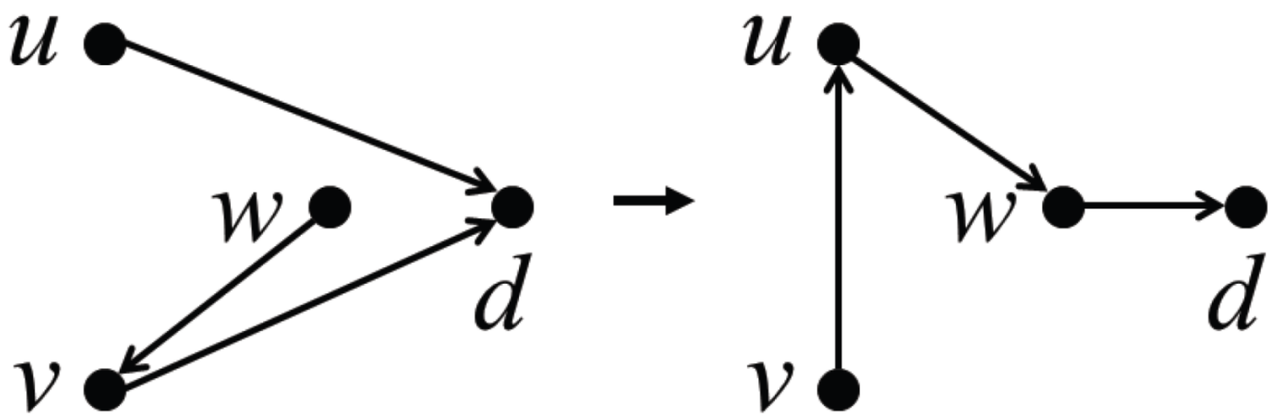
+ stronger packet coherence

– version number in packets

– switches need to store both versions

Minimum SDN Updates?

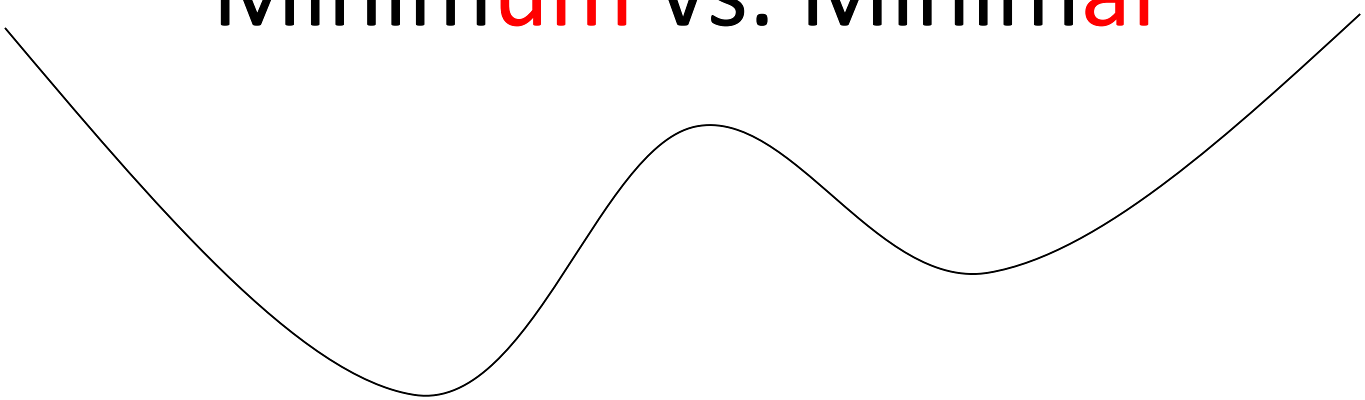
Minimum Updates: Another Example



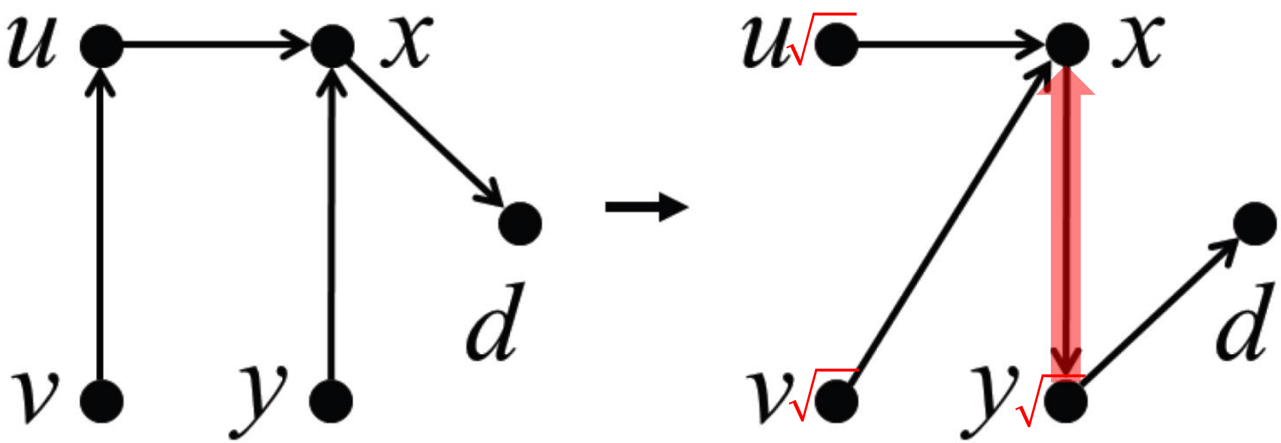
$w \downarrow u$ *or* $w \downarrow v$

No node can improve without hurting another node

Minimum vs. Minimal



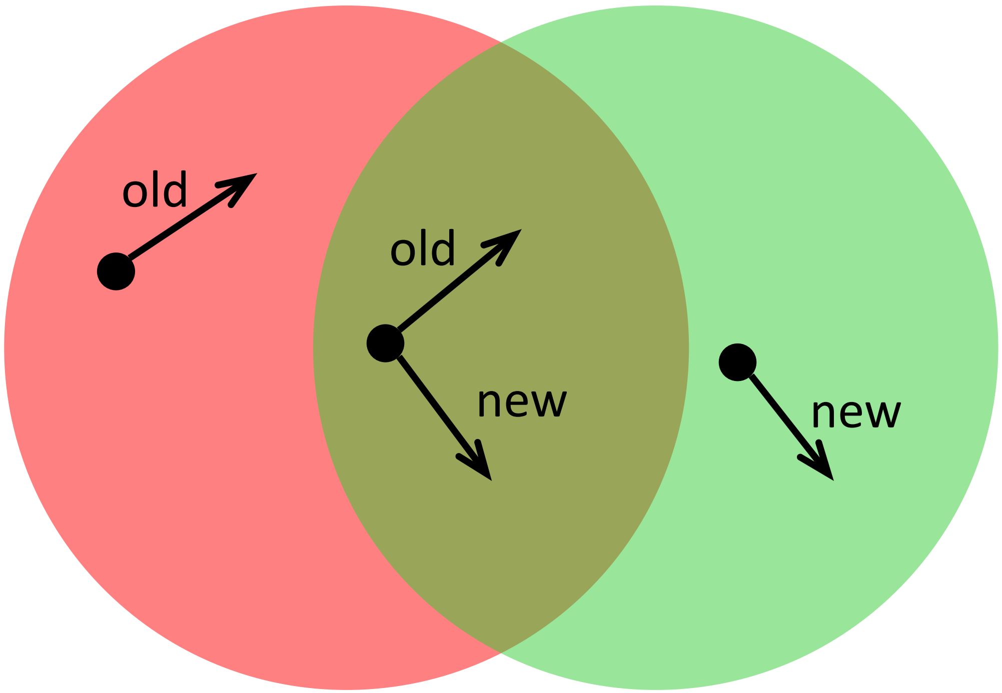
Minimal Dependency Forest



Next: An algorithm to compute minimal dependency forest.

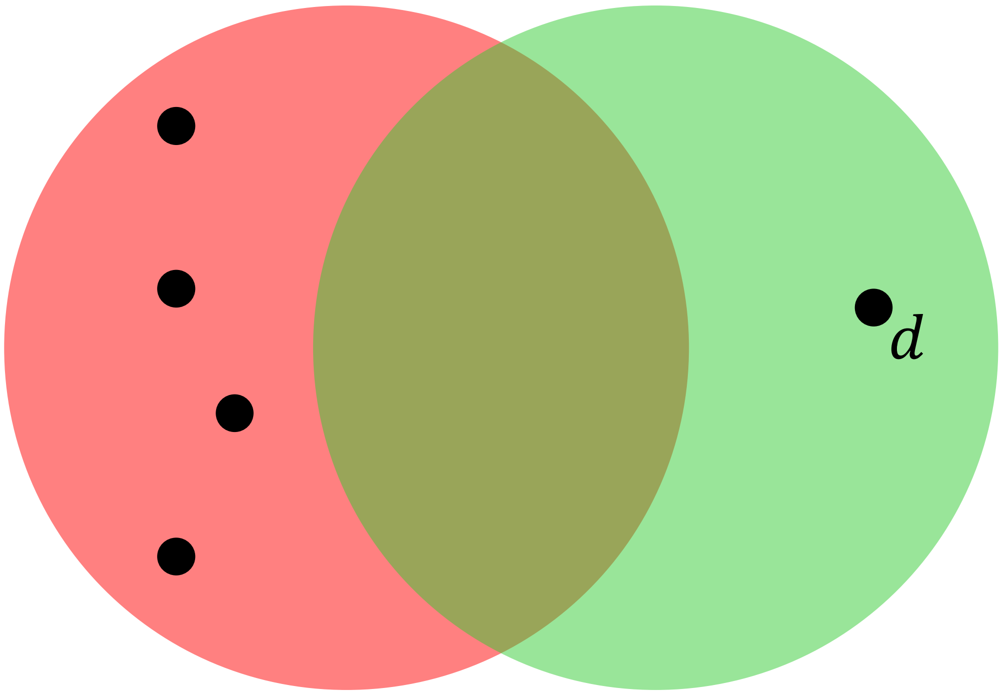
Algorithm for Minimal Dependency Forest

- Each node in one of three states: old, new, and limbo (both old *and* new)



Algorithm for Minimal Dependency Forest

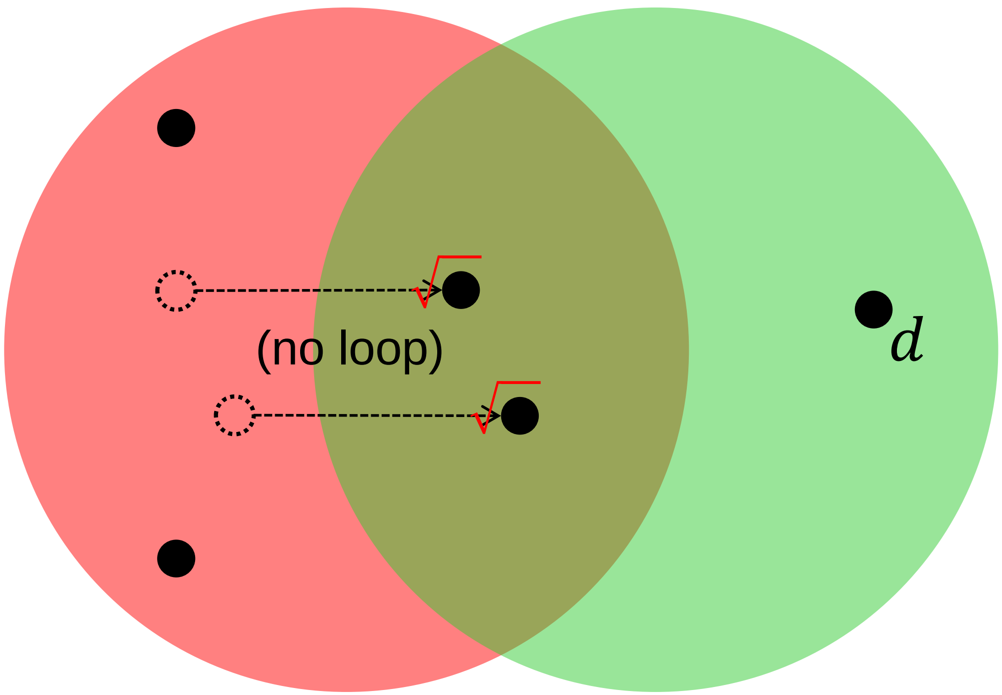
- Each node in one of three states: old, new, and limbo (both old *and* new)
- Originally, destination node in new state, all other nodes in old state
- Invariant: No loop!



Algorithm for Minimal Dependency Forest

Initialization

- Old node u: No loop* when adding new pointer, move node to limbo!
- This node u will be a root in dependency forest



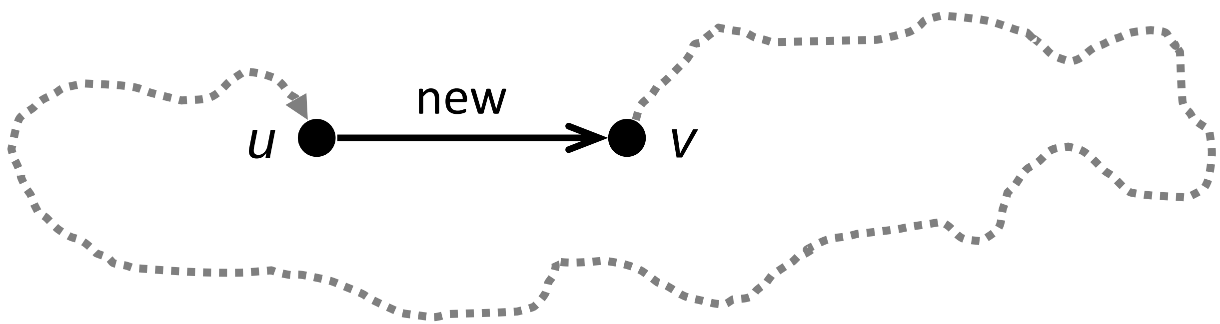
*Loop Detection: Simple procedure, see next slide

Loop Detection

- Will a new rule *u.new = v* induce a loop?

  - We know that the graph so far has no loops
  - Any new loop *must* contain the edge (*u,v*)

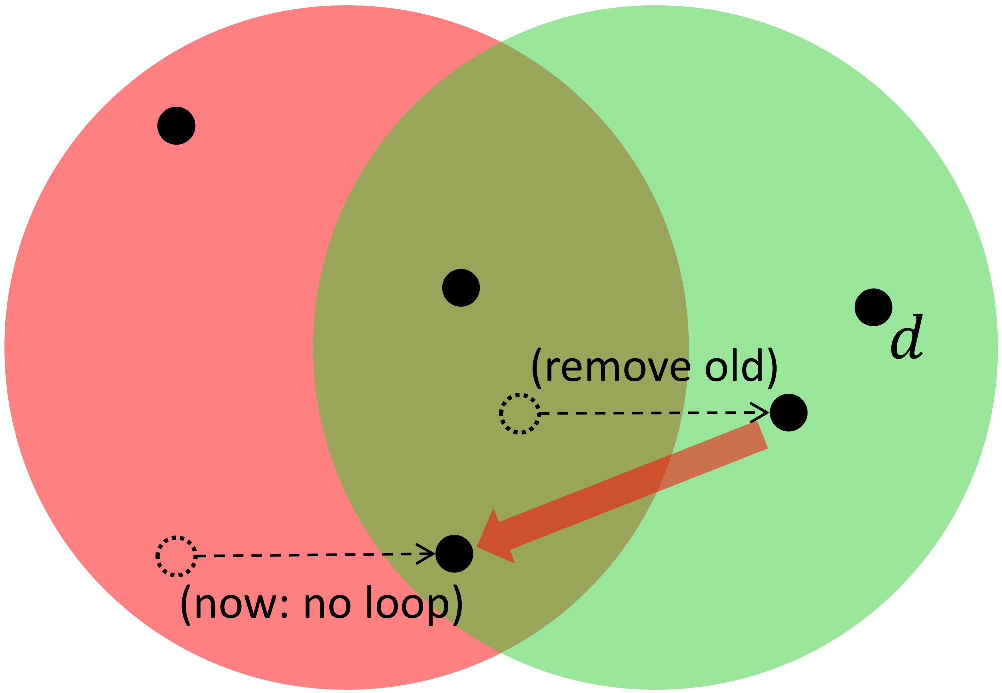
- In other words, is node *u* now *reachable* from node *v*?



- Depth first search (DFS) at node *v*
  - If we visit node *u*: the new rule induces a loop
  - Else: no loop

Algorithm for Minimal Dependency Forest

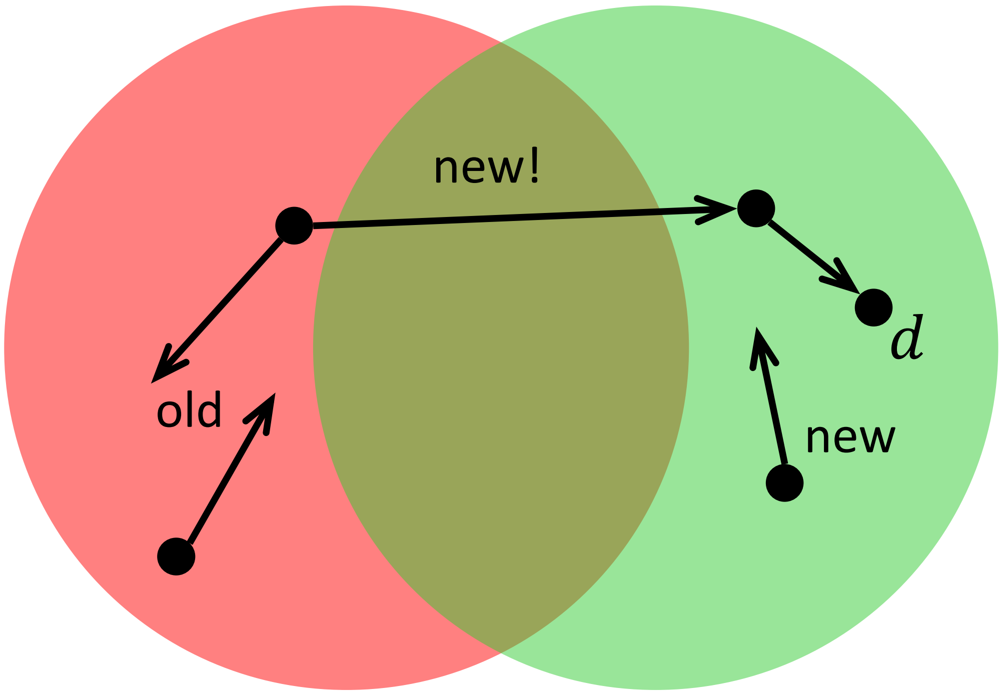
- Limbo node u: Remove old pointer (move node to new)
- Consequence: Some old nodes v might move to limbo!
- Node v will be child of u in dependency forest!



Algorithm for Minimal Dependency Forest

Process terminates

- You can always move a node from limbo to new.
- Can you ever have old nodes but no limbo nodes? No, because…



…one can easily derive a contradiction!

Main Contribution

For a given consistency property, what is the minimal dependency possible?

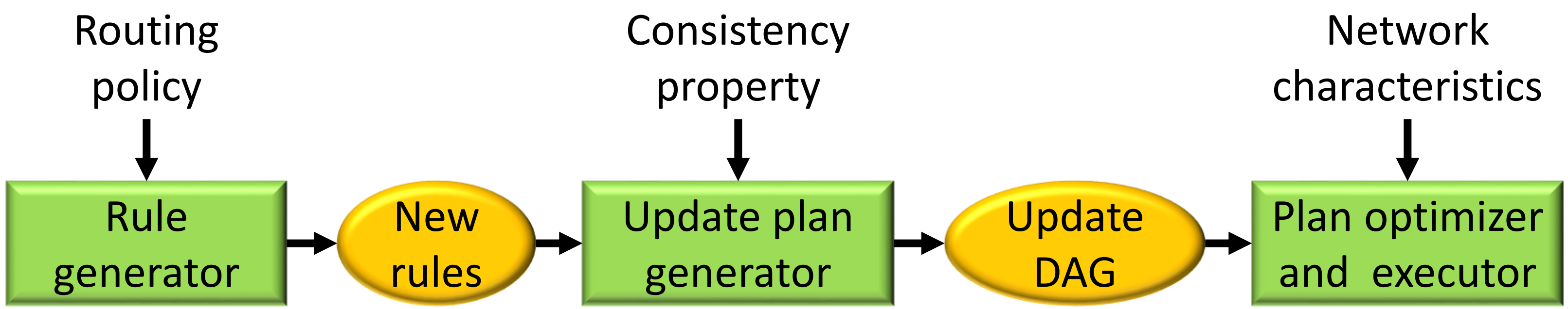
Consistency Space

	None	**Self**	**Downstream subset**	**Downstream all**	**Global**
Eventual consistency	Always guaranteed				
Drop freedom	Impossible	Add before remove			
Memory limit	Impossible	Remove before add			
Loop freedom	Impossible		Rule dep. forest	Rule dep. tree	
Packet coherence	Impossible			Per-flow ver. numbers	Global ver. numbers
Bandwidth limit	Impossible				Staged partial moves

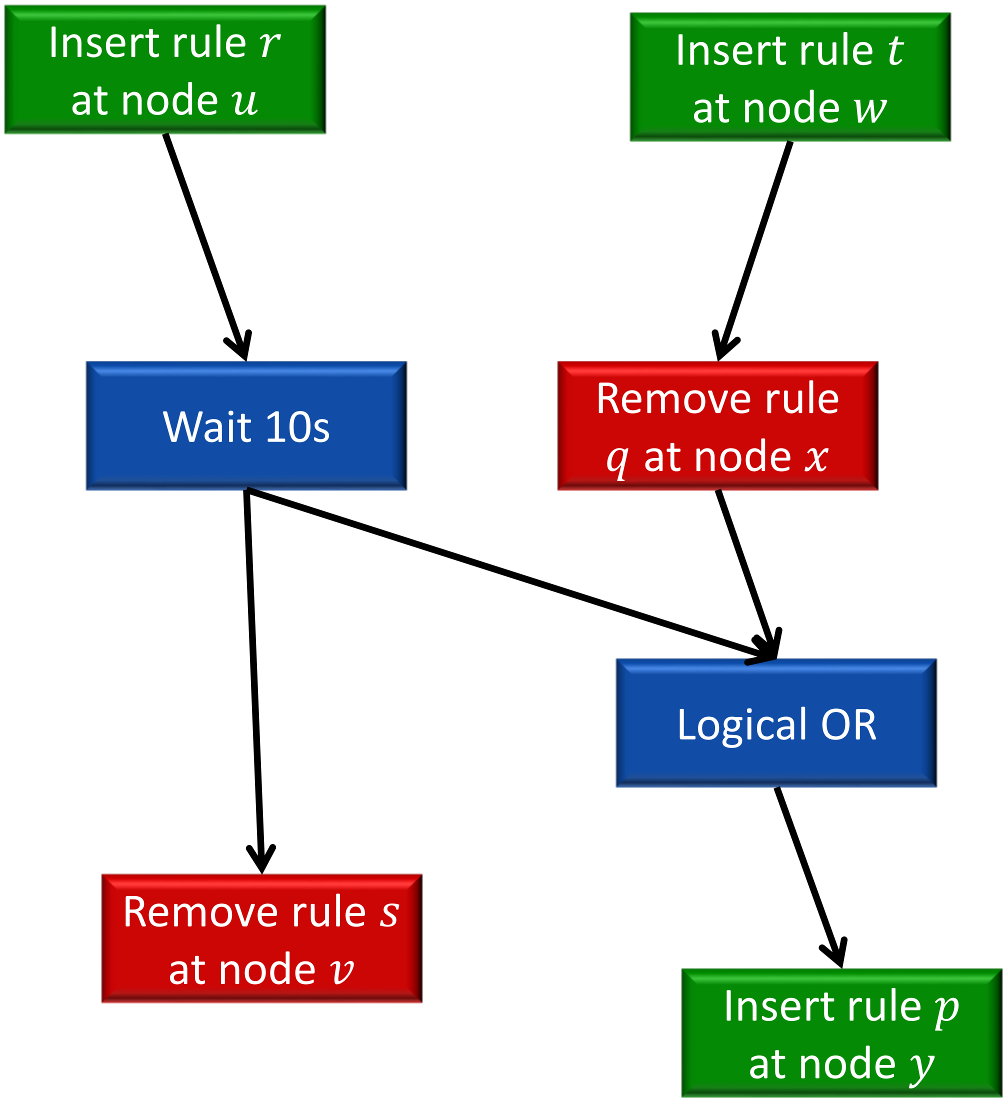
It's *not* just how to compute new rules.

It is also how to gracefully get from current to new configuration, respecting consistency.

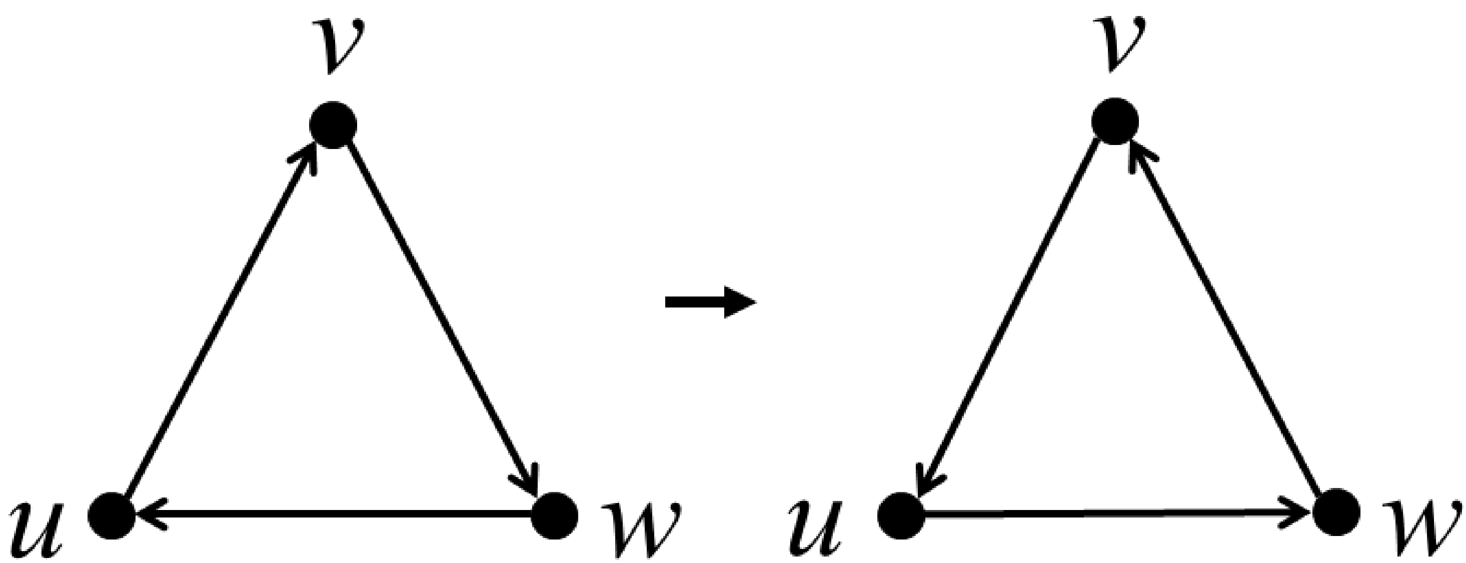
Architecture



Update DAG

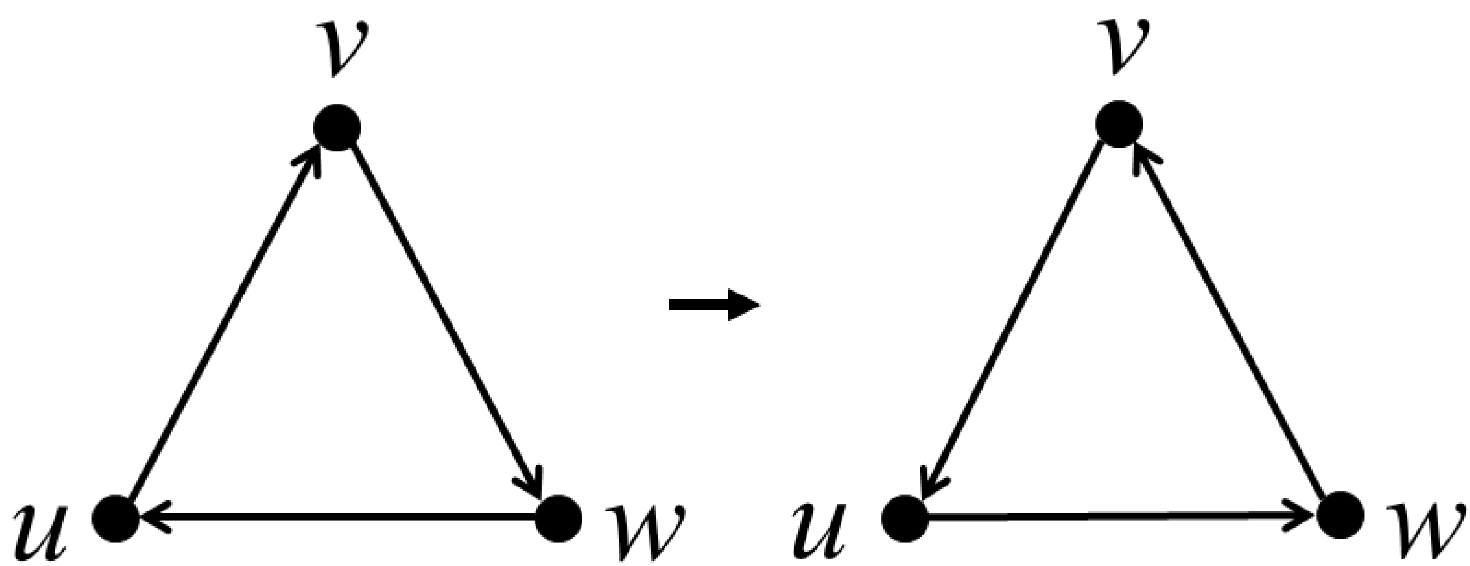

Insert rule r at node u
Insert rule t at node w
Wait 10s
Remove rule q at node x
Logical OR
Remove rule s at node v
Insert rule p at node y

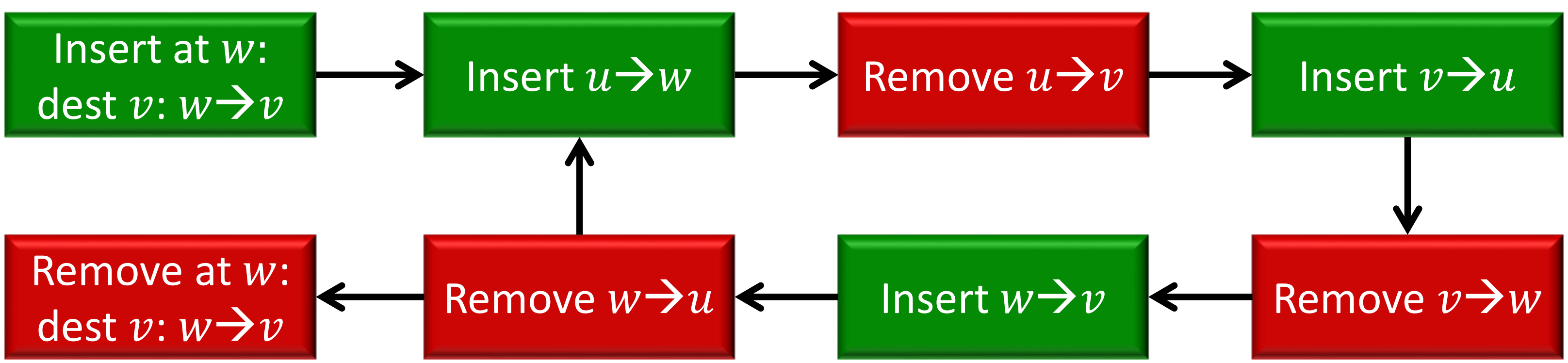
Multiple Destinations using Prefix-Based Routing



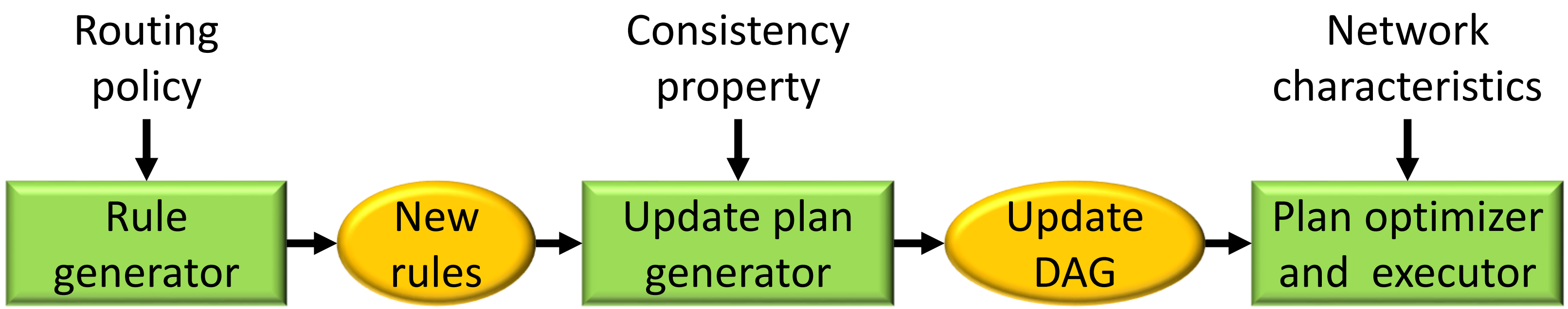
- No new “default” rule can be introduced without causing loops
- Solution: Rule-Dependency Graphs!
- Deciding if simple update schedule exists is hard!

Breaking Cycles

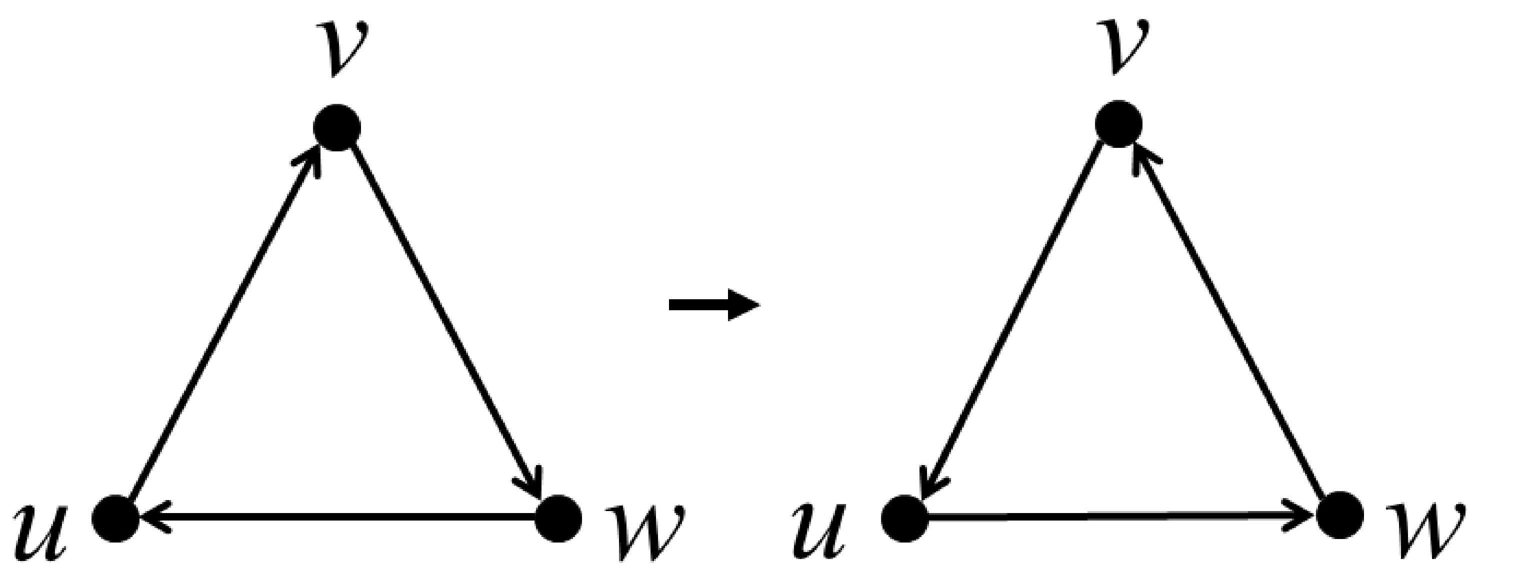




Architecture



Breaking Cycles



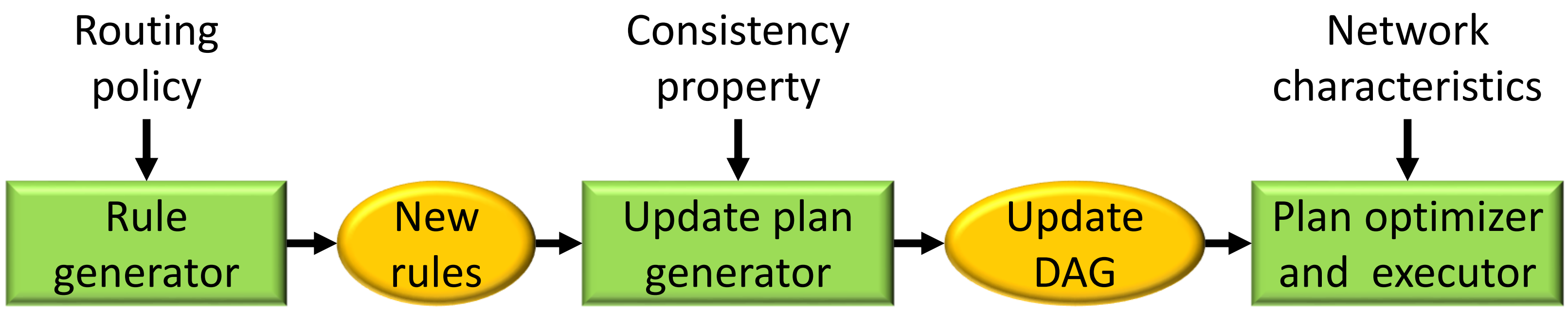
Insert at w: dest v: $w \rightarrow v$ → Insert $u \rightarrow w$ → Remove $u \rightarrow v$ → Insert $v \rightarrow u$ → Remove $v \rightarrow w$ → Insert $w \rightarrow v$ → Remove $w \rightarrow u$ → Remove at w: dest v: $w \rightarrow v$

Are Minimal Dependencies Good?

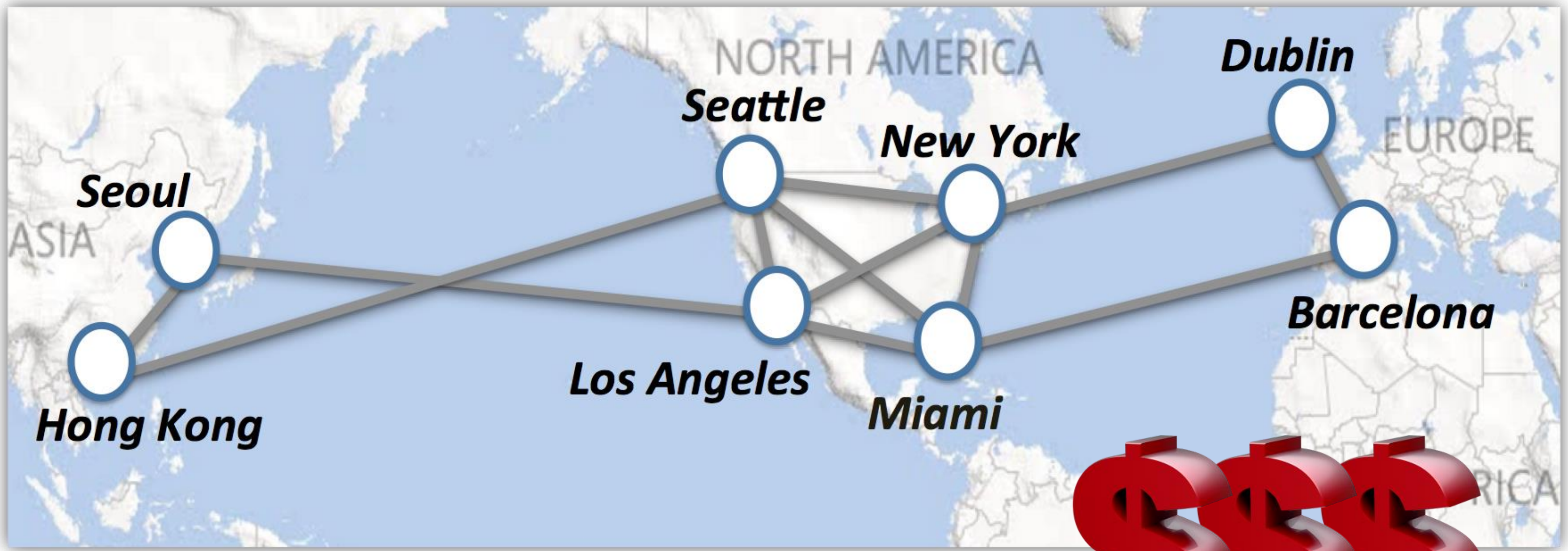
…it depends

(But Plan optimizer and executor will fix it.)

Architecture

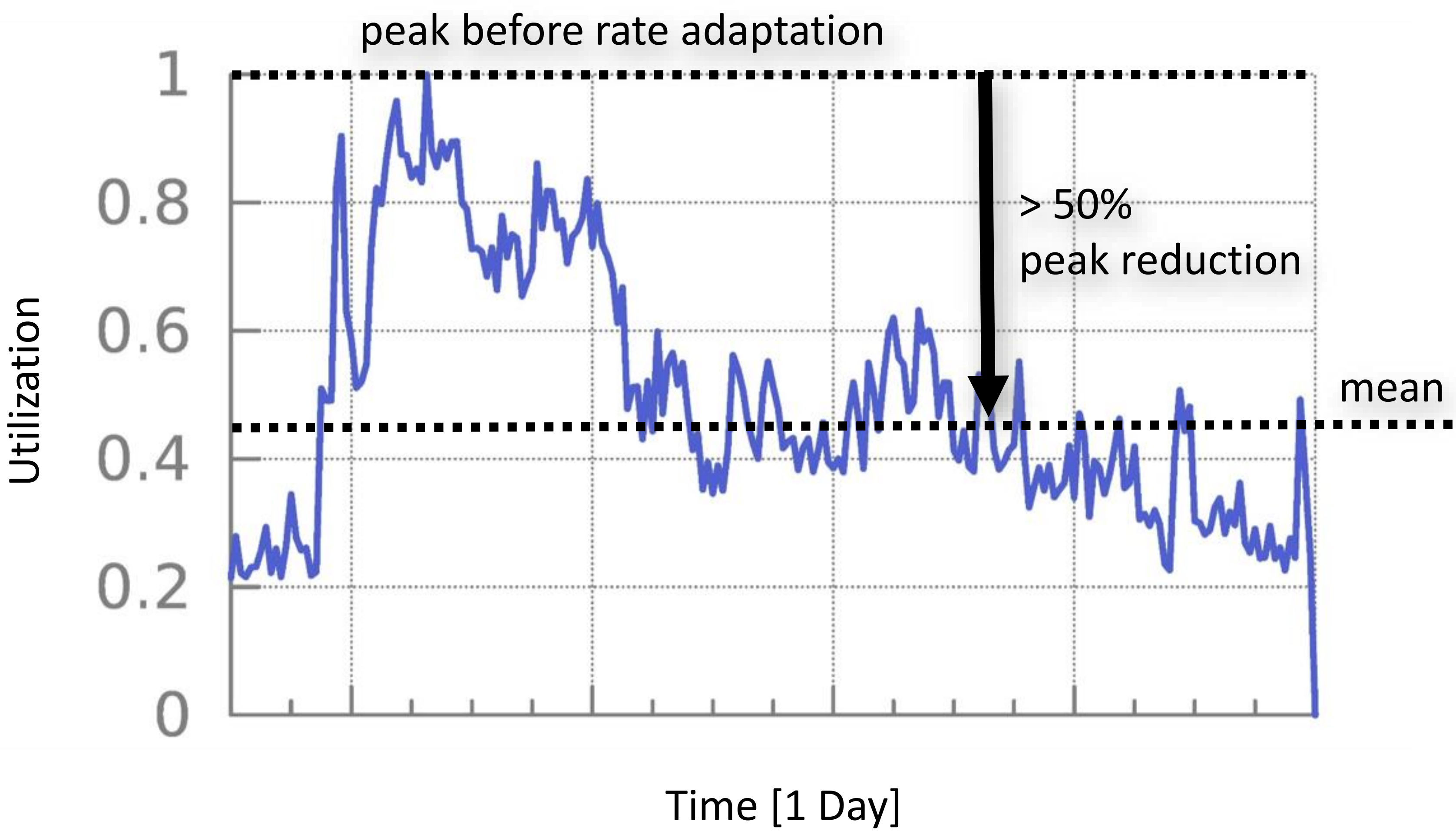


Real Application: Inter-Data Center WANs

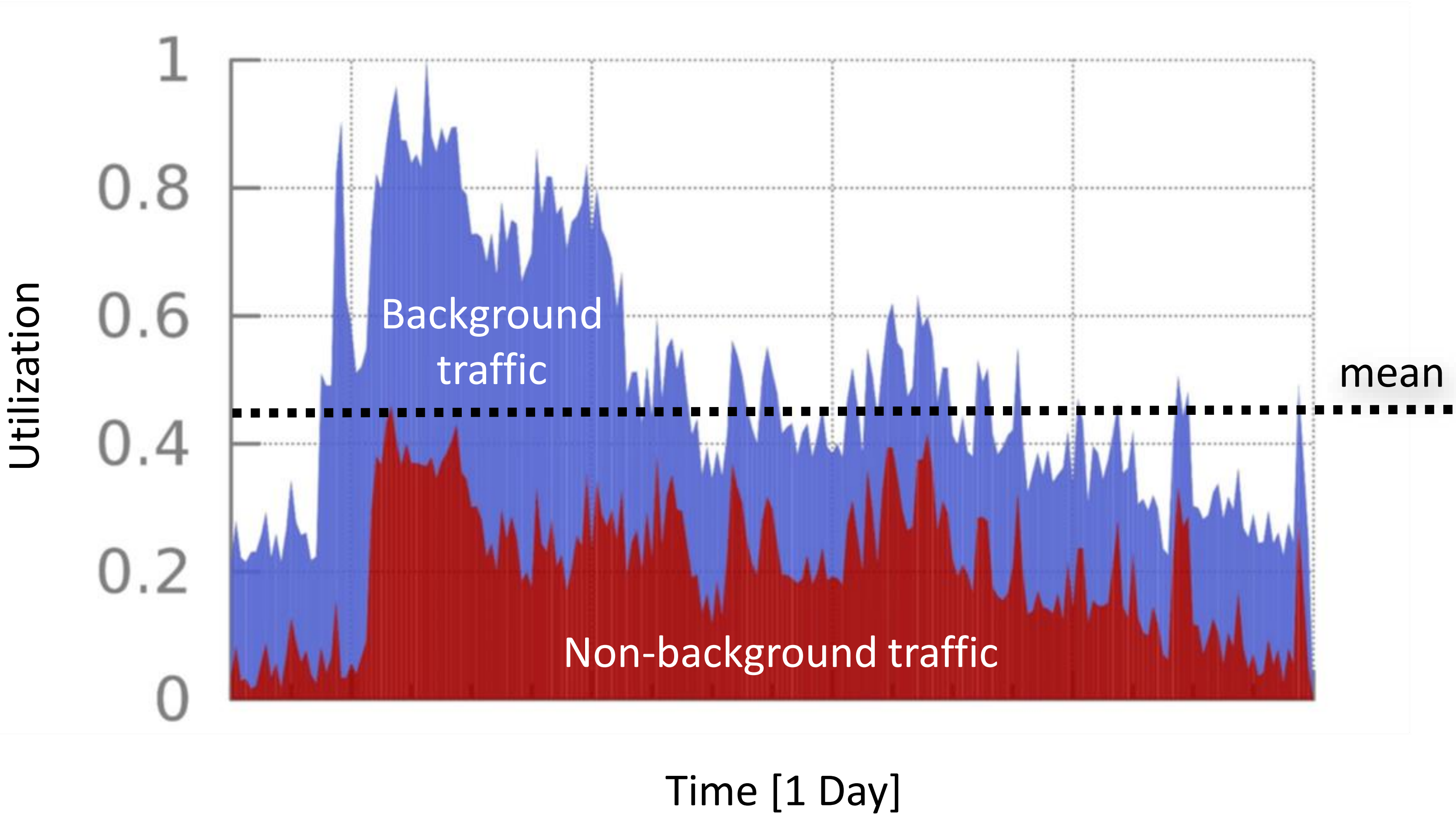


Think: Google, Amazon, Microsoft

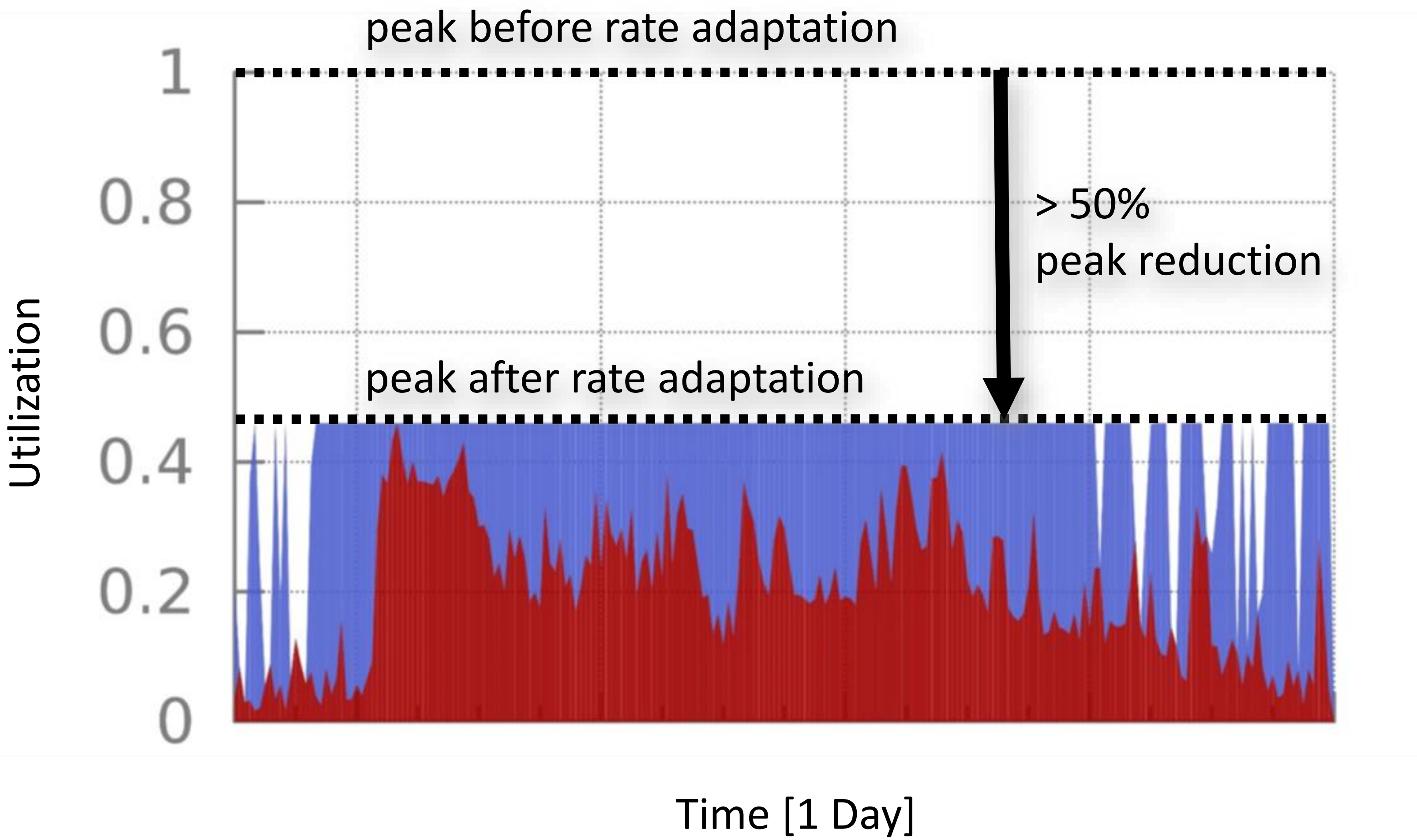
Problem: Typical Network Utilization



Problem: Typical Network Utilization



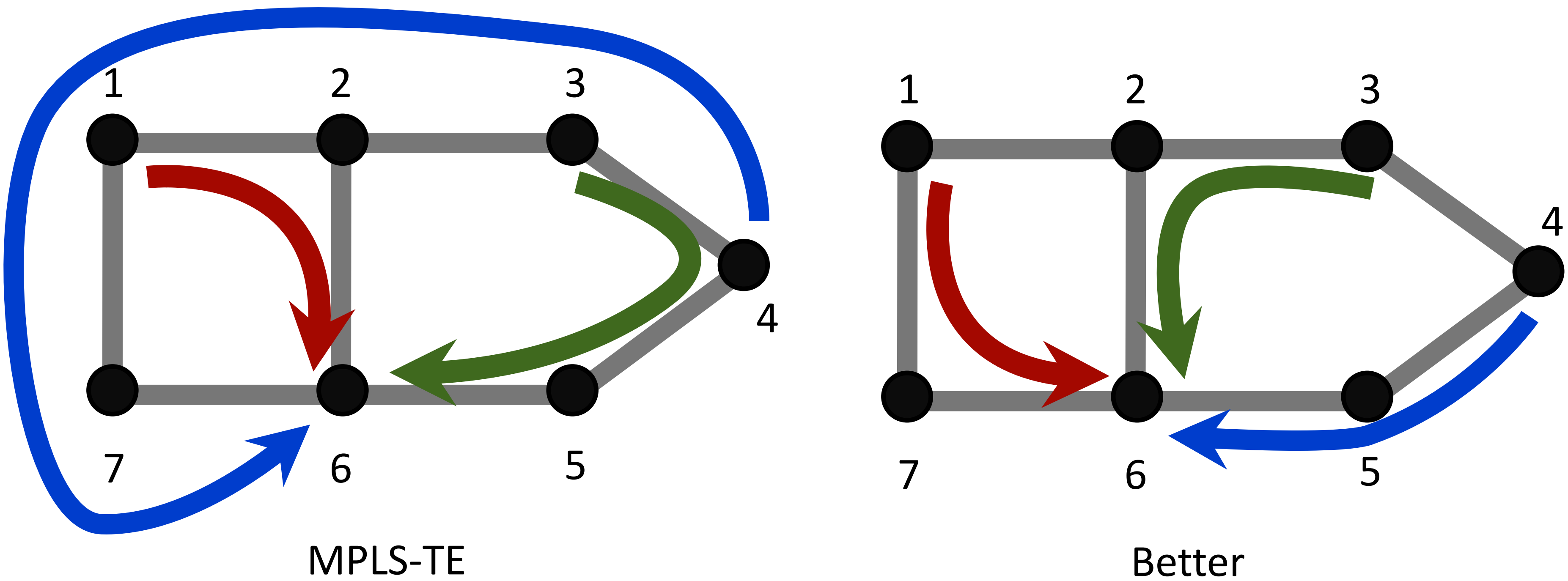
Problem: Typical Network Utilization



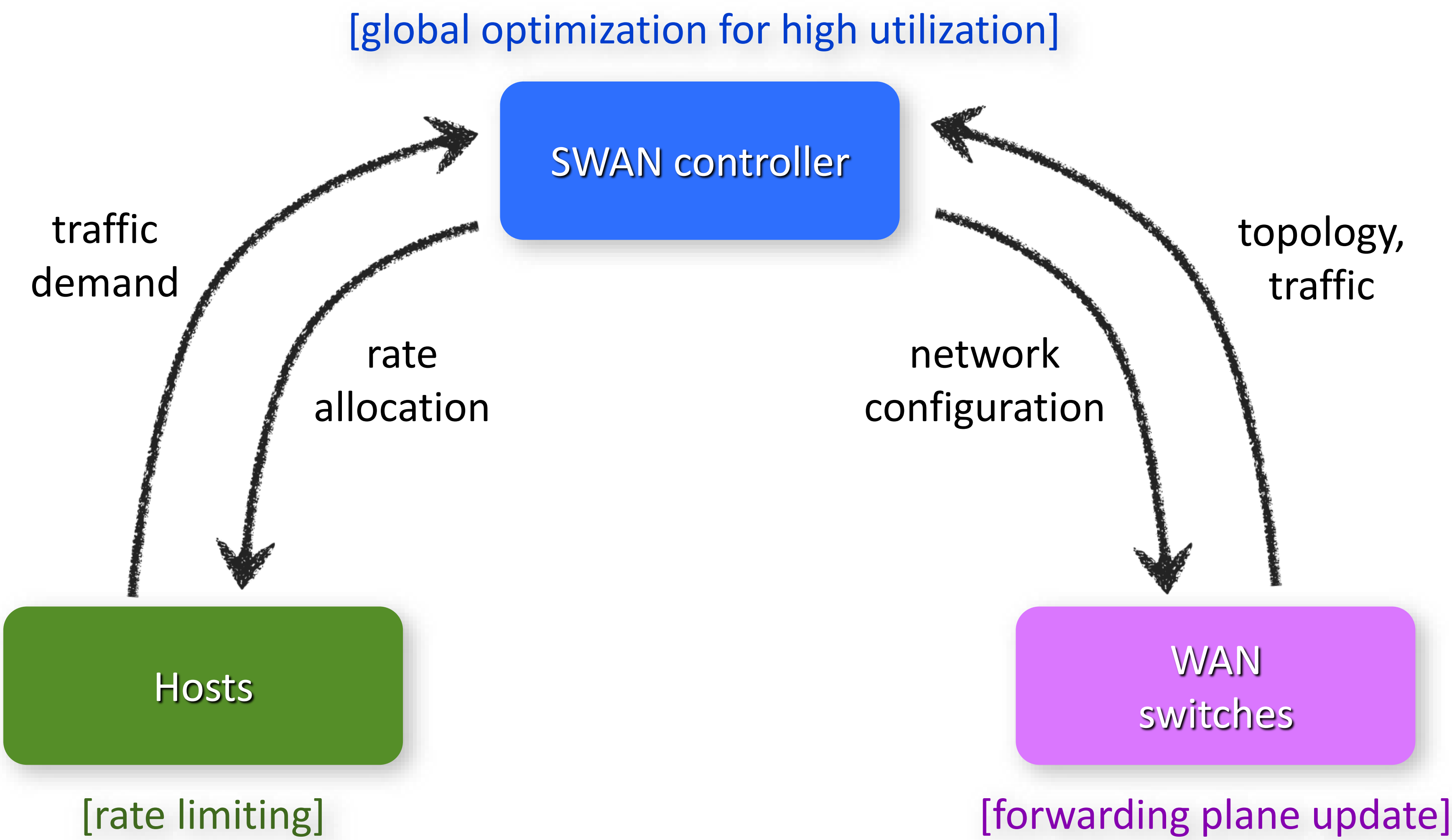
Another Problem: Online Routing Decisions

flow arrival order: A, B, C

each link can carry at most one flow (in both directions)



The SWAN Project



Algorithms?

- Priority classes (2-3)
- Allocate highest priority first

- Solve with multi-commodity flow (LP) within each class
  - Flows are splittable
  - Well understood, fast enough for our input (seconds)

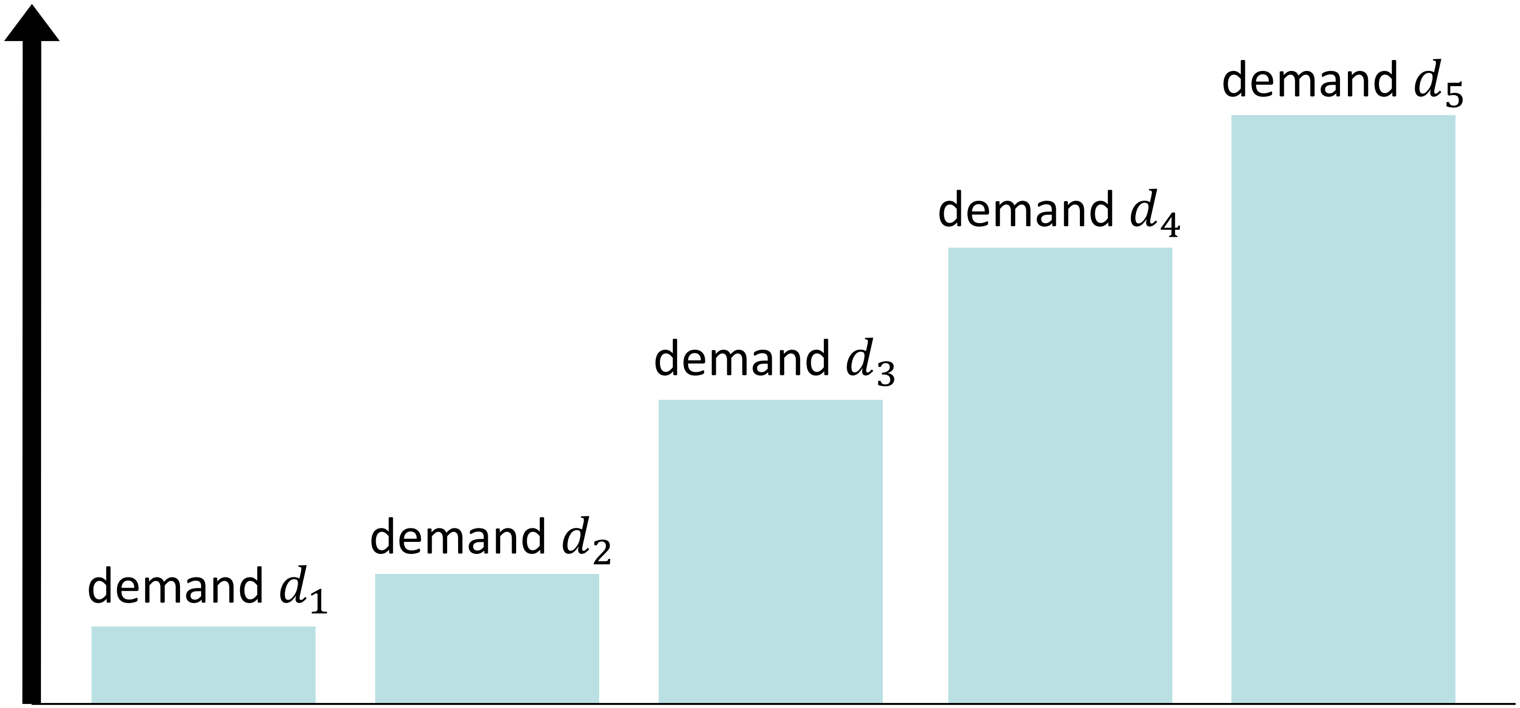
- But: Within a priority class we want max-min fairness (“$f_i \geq f, \max f$”)
  - Definition: Make nobody richer at cost of someone poorer
  - Works, but now one has to solve linearly many LPs, which is too slow (hours)
  - A perfect example of algorithm engineering?

- Solution: Fairness approximation!

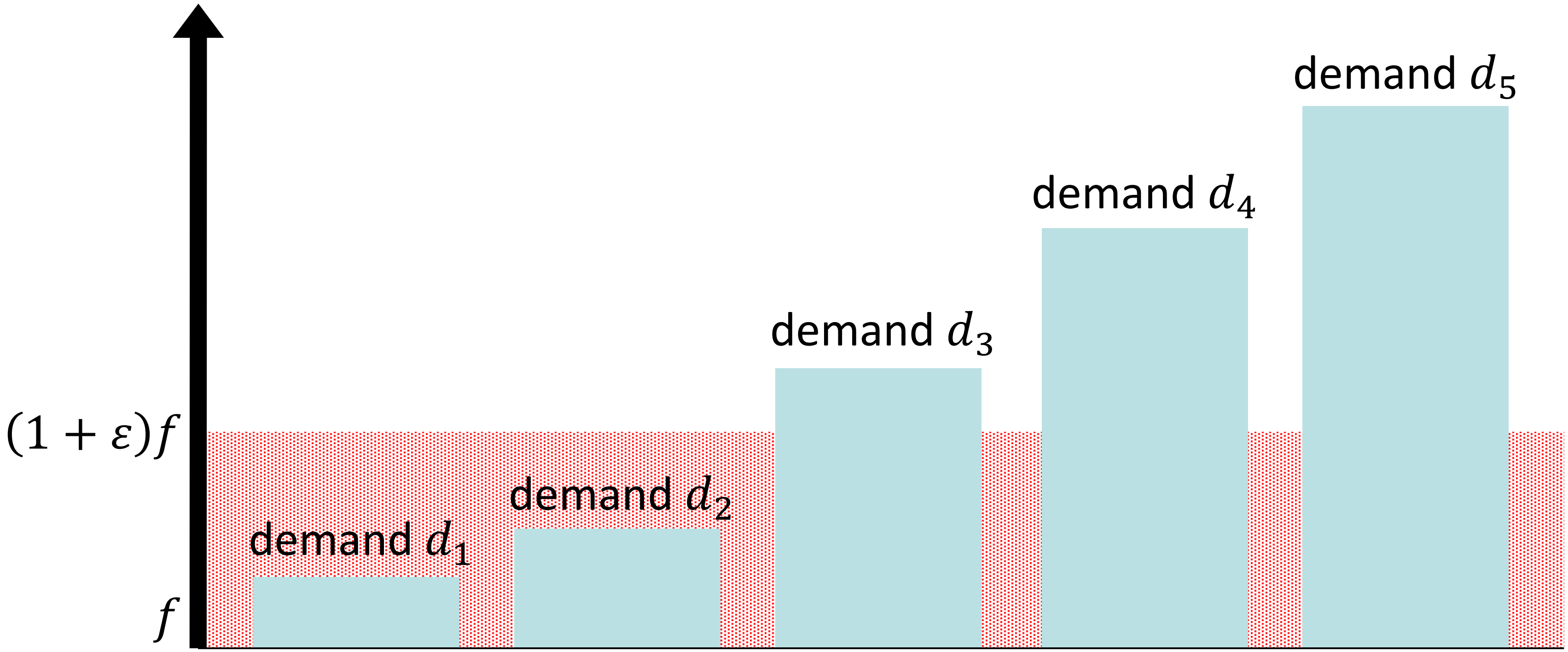
Multicommodity Flow LP

Maximize throughput	$\max \sum_i f_i$
Flow less than demand	$0 \leq f_i \leq d_i$
Flow less than capacity	$\sum_i f_i(e) \leq c(e)$
Flow conservation on inner nodes	$\sum_u f_i(u,v) = \sum_w f_i(v,w)$
Flow definition on source, destination	$\sum_v f_i(s_i,v) = \sum_u f_i(u,t_i) = f_i$

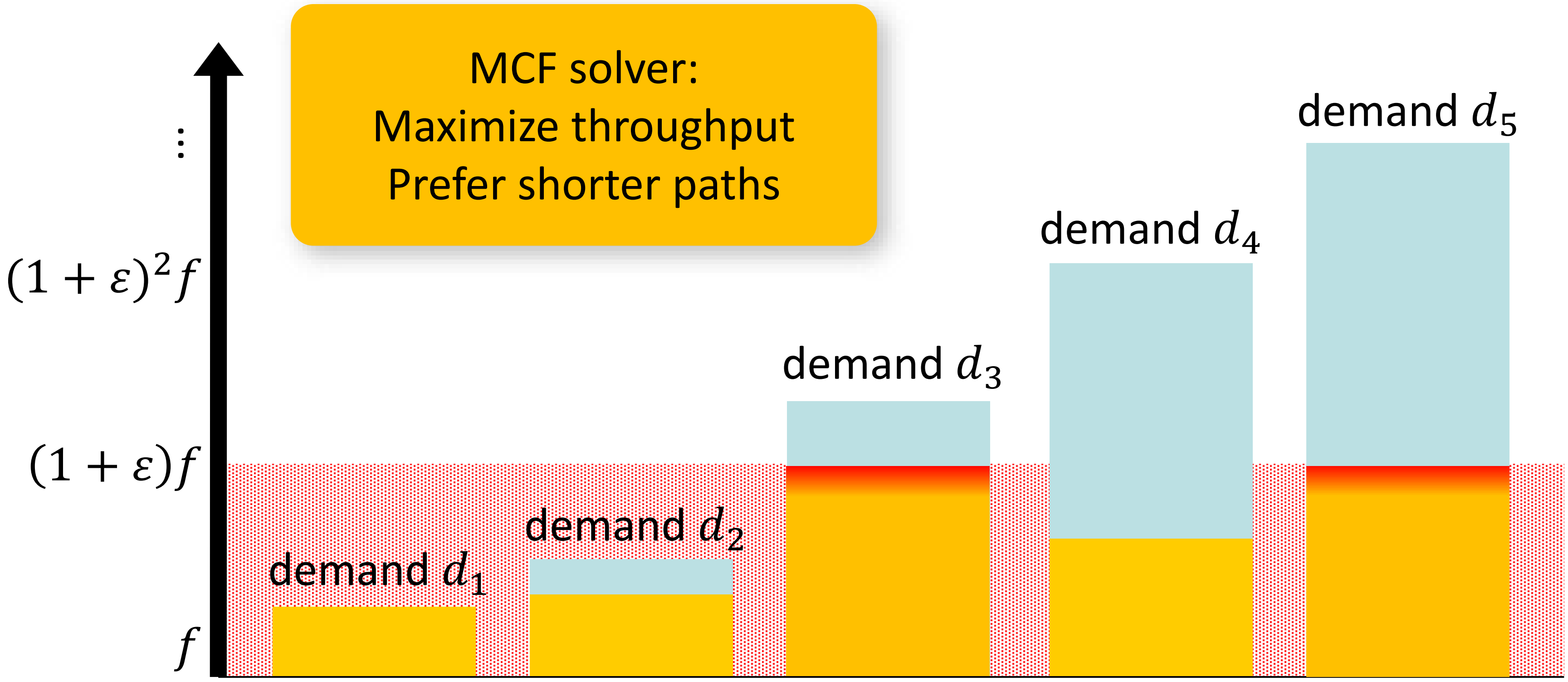
Approximated max-min fairness



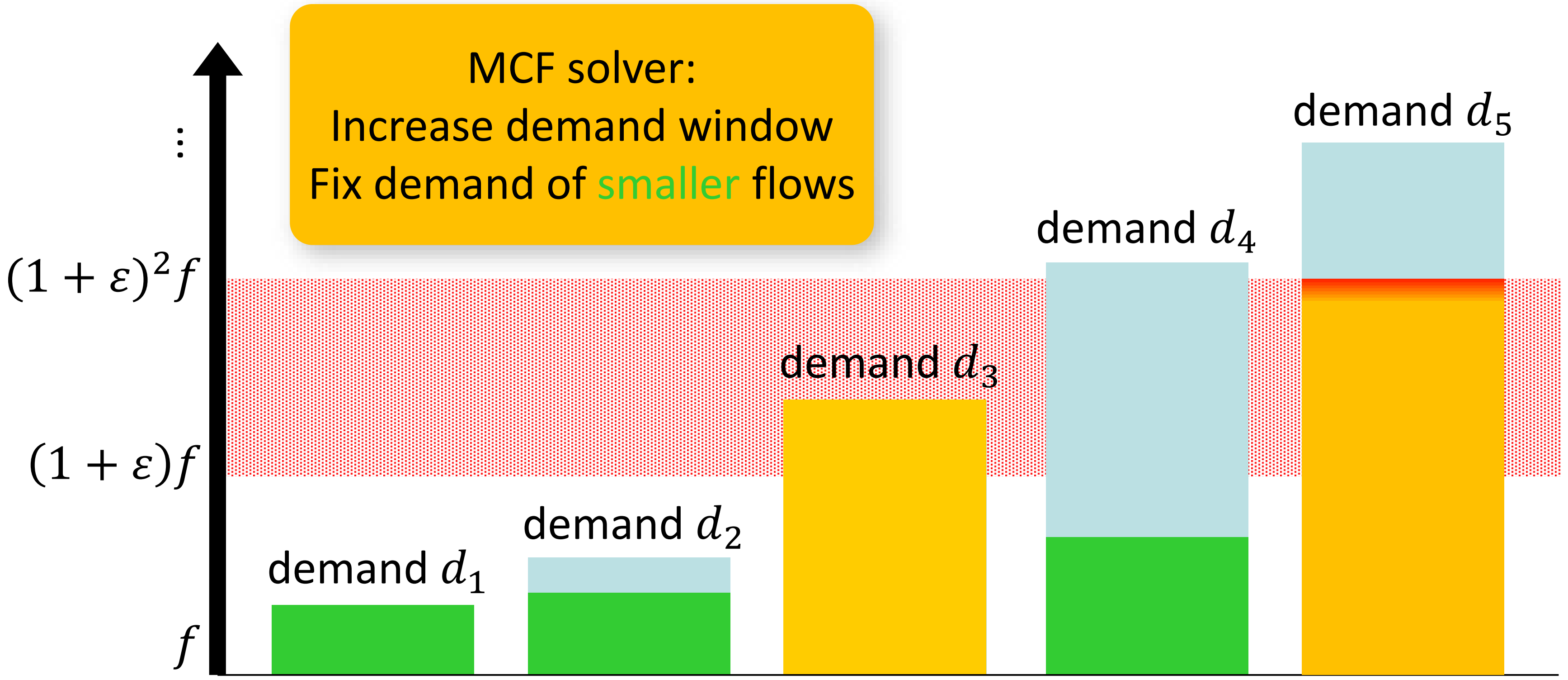
Approximated max-min fairness



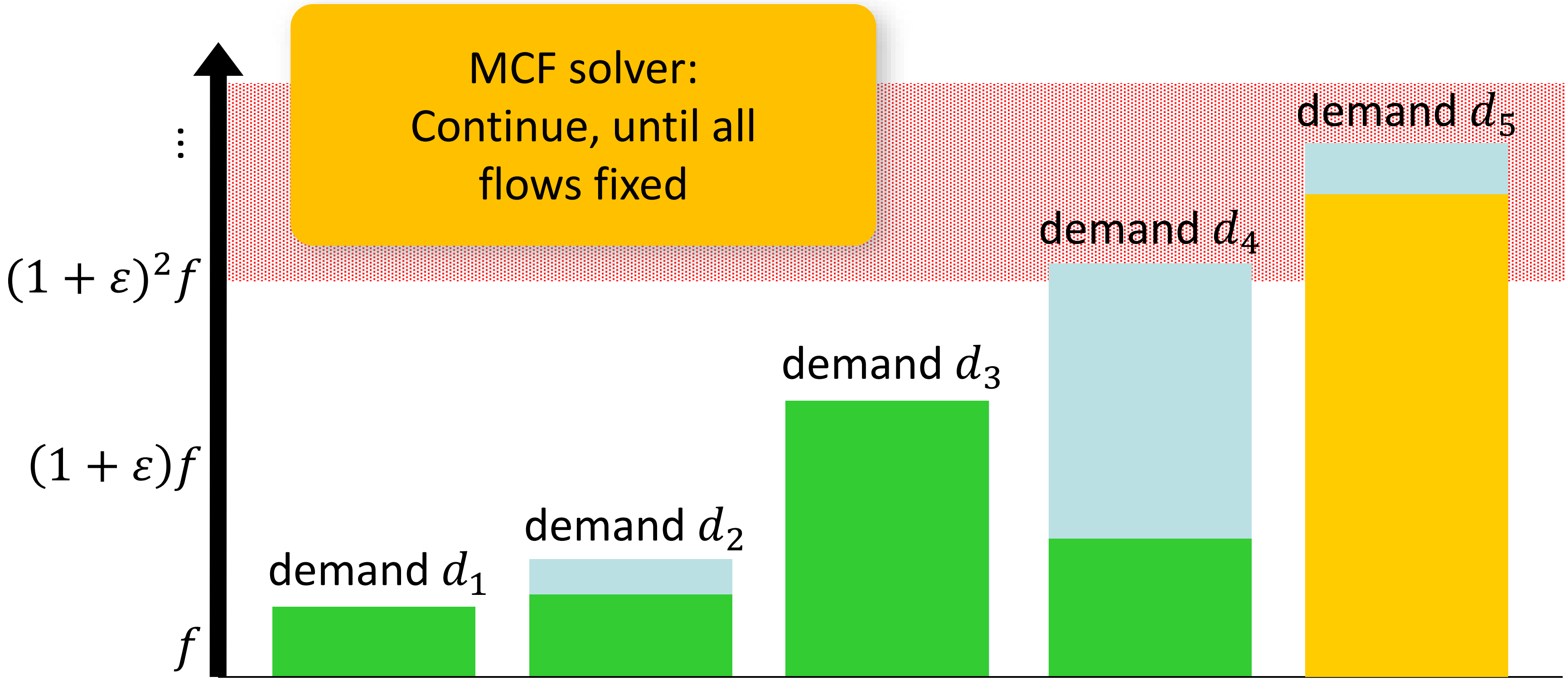
Approximated max-min fairness



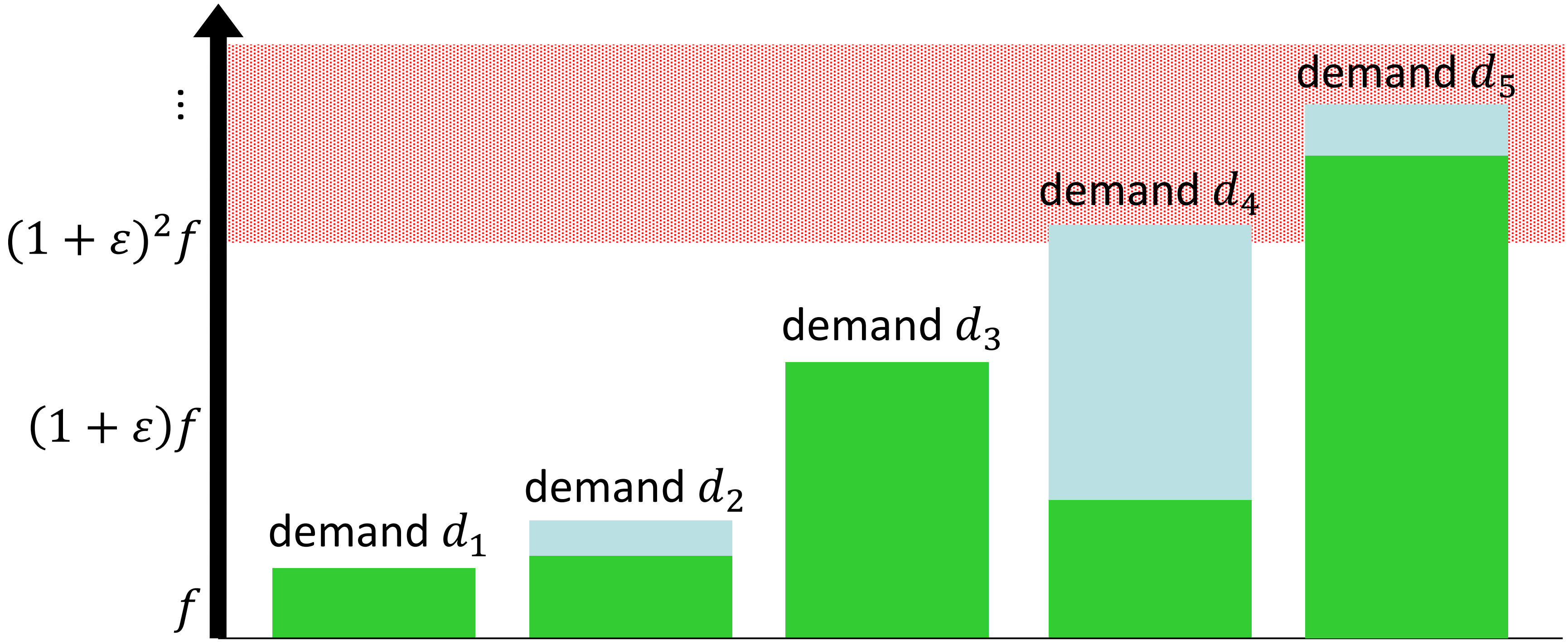
Approximated max-min fairness



Approximated max-min fairness

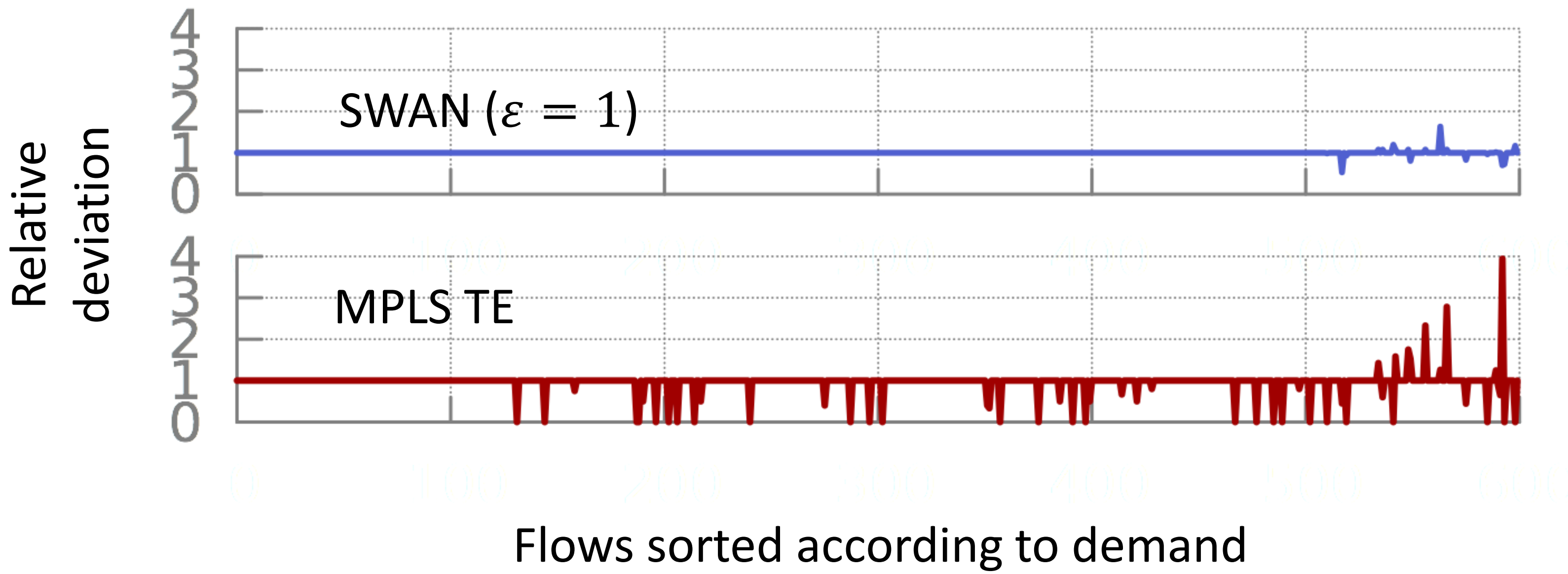


Approximated max-min fairness

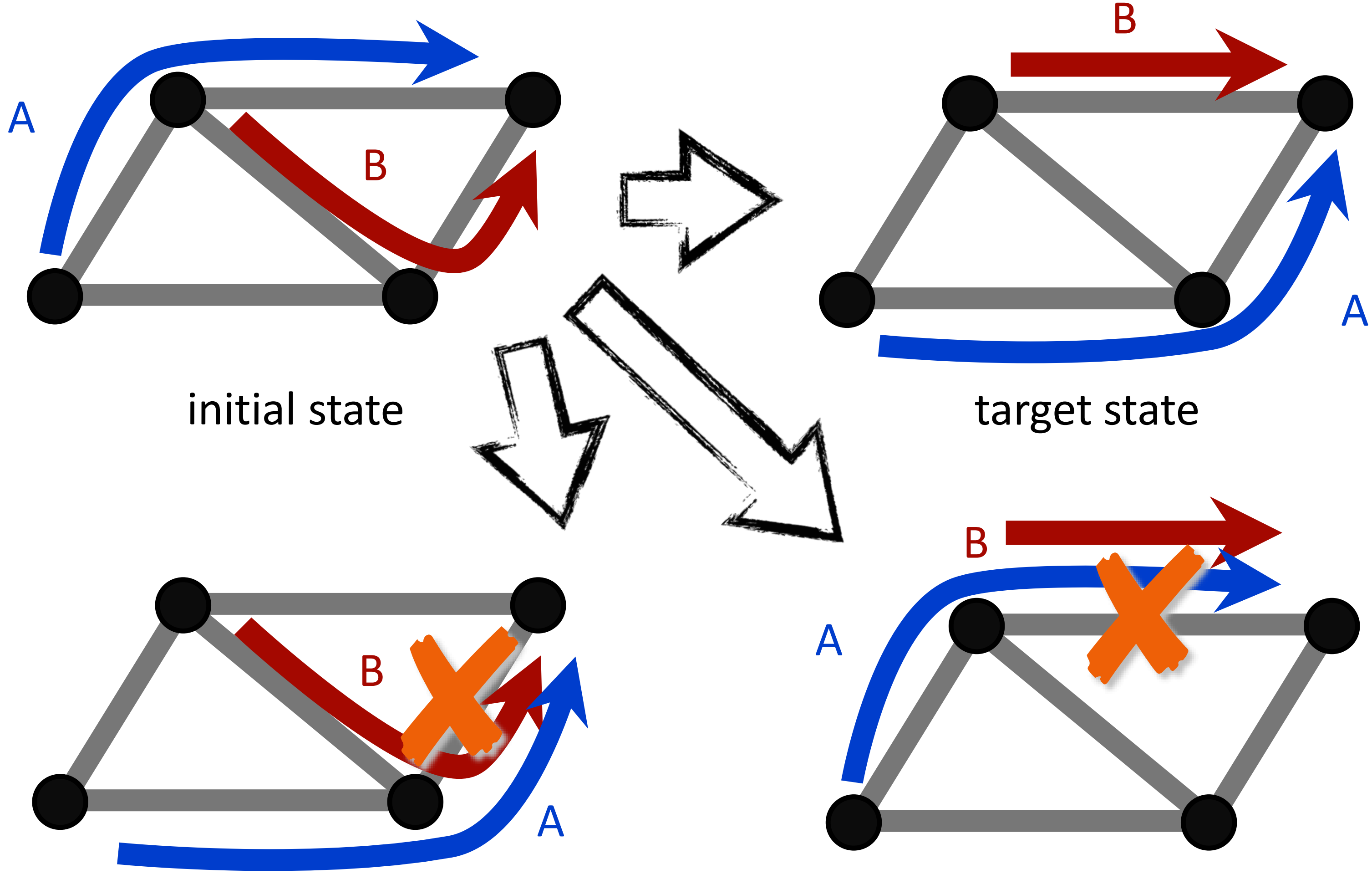


- In theory, this process is $(1+\varepsilon)$ competitive
- In practice, with $\varepsilon = 1$, only 4% of flows deviate over 5% from their fair share

Fairness: SWAN vs. MPLS TE



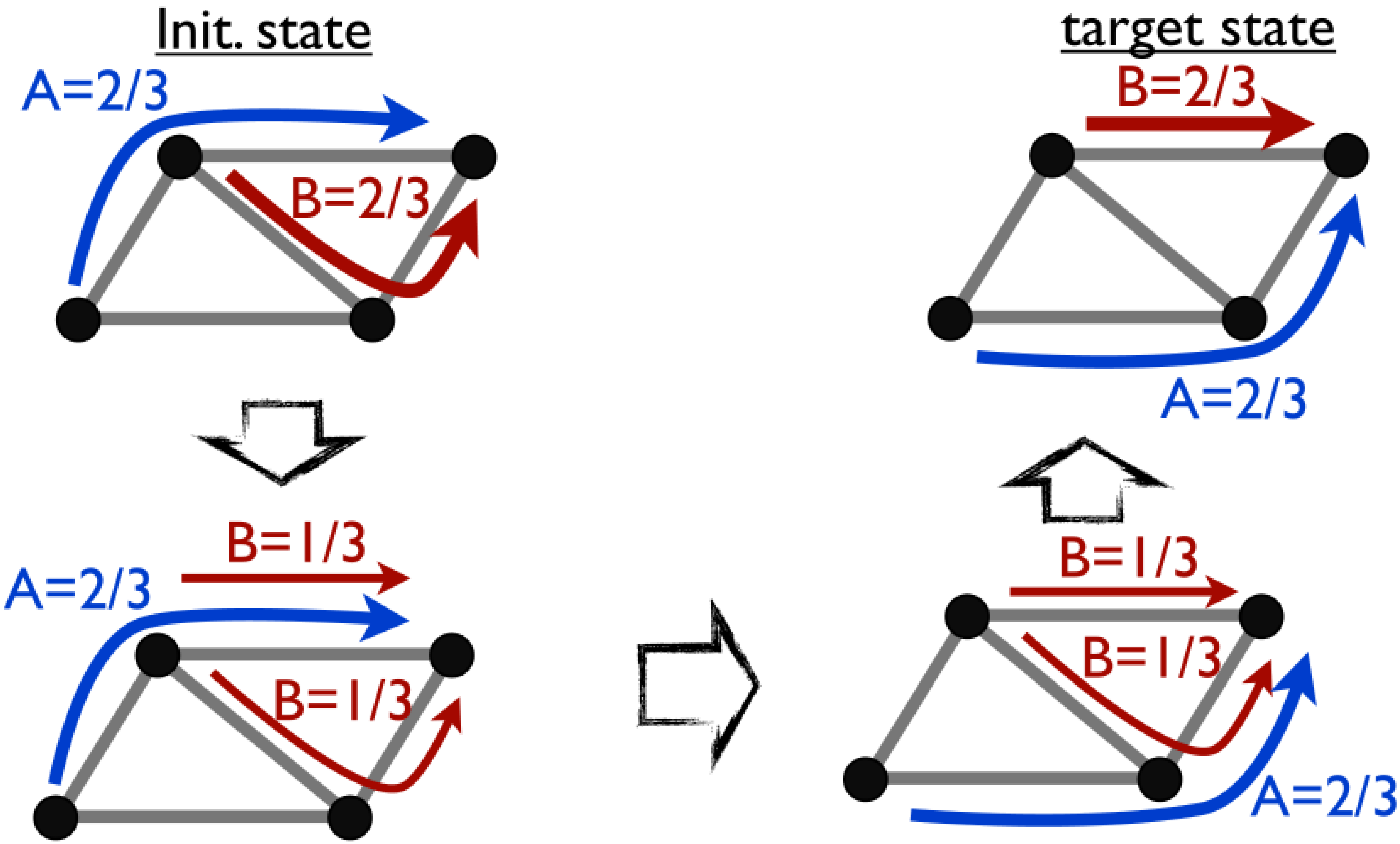
Problem: Consistent Updates



Capacity-Consistent Updates

- Not directly, but maybe through intermediate states?
- Solution: Leave a fraction s slack on each edge, less than $1/s$ steps
- Example: Slack = 1/3 of link capacity

Example: Slack = 1/3 of link capacity



Capacity-Consistent Updates

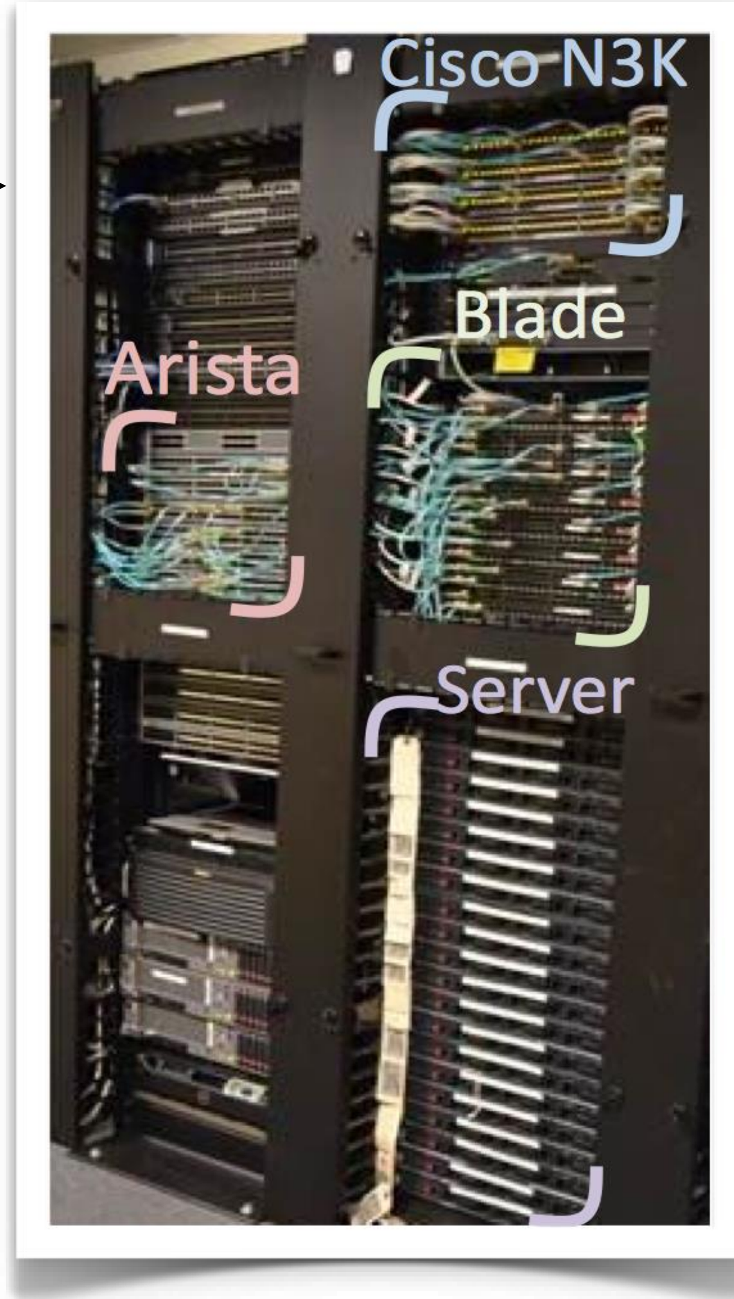
- Alternatively: Try whether a solvable LP with k steps exist, for $k = 1, 2, 3$ ...
  - Sum of flows in steps j and $j + 1$, together, must be less than capacity limit

Only growing flows	$f_i^0 \leq f_i^k$
Flow less than capacity	$\sum_i \max\left(f_i^j(e), f_i^{j+1}(e)\right) \leq c(e)$
Flow conservation on inner nodes	$\sum_u f_i^j(u,v) = \sum_w f_i^j(v,w)$
Flow definition on source, destination	$\sum_v f_i^j\,(s_i, v) = \sum_u f_i^j\,(u, t_i) = f_i^j$

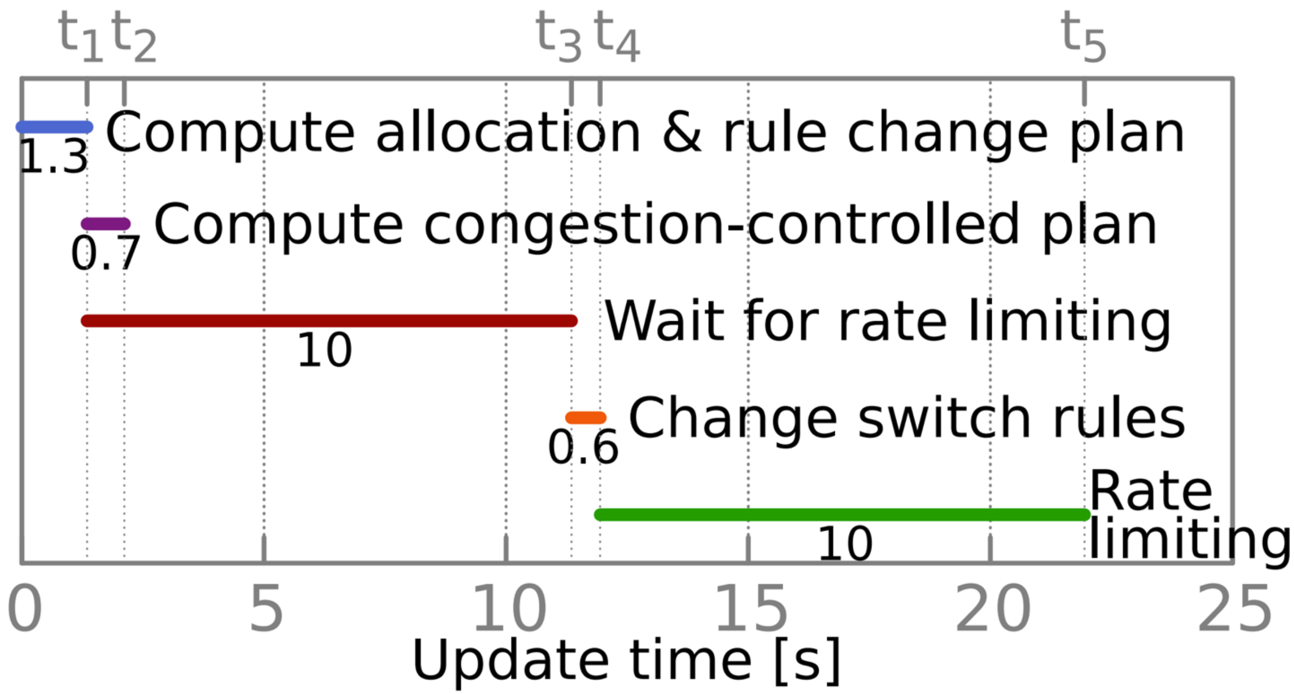
Evaluation platforms

- Prototype
  - 5 DCs across 3 continents
  - 10 switches

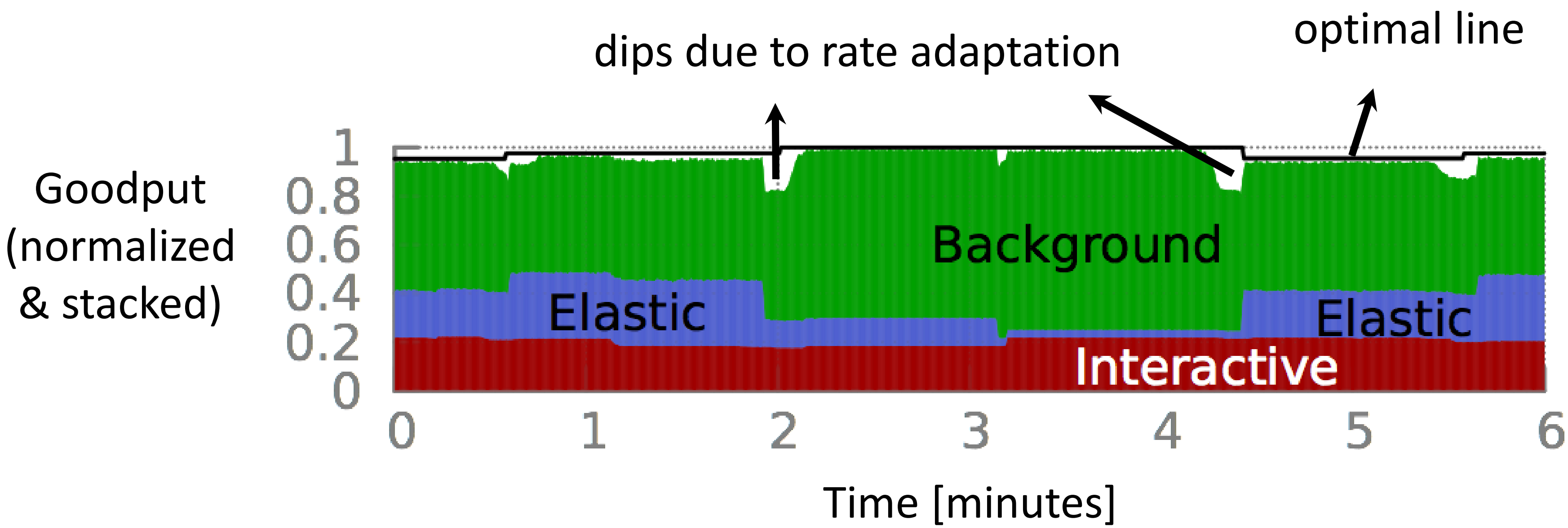
- Data-driven evaluation
  - 40+ DCs across 3 continents
  - 80+ switches



Time for One Network Update



Prototype Evaluation

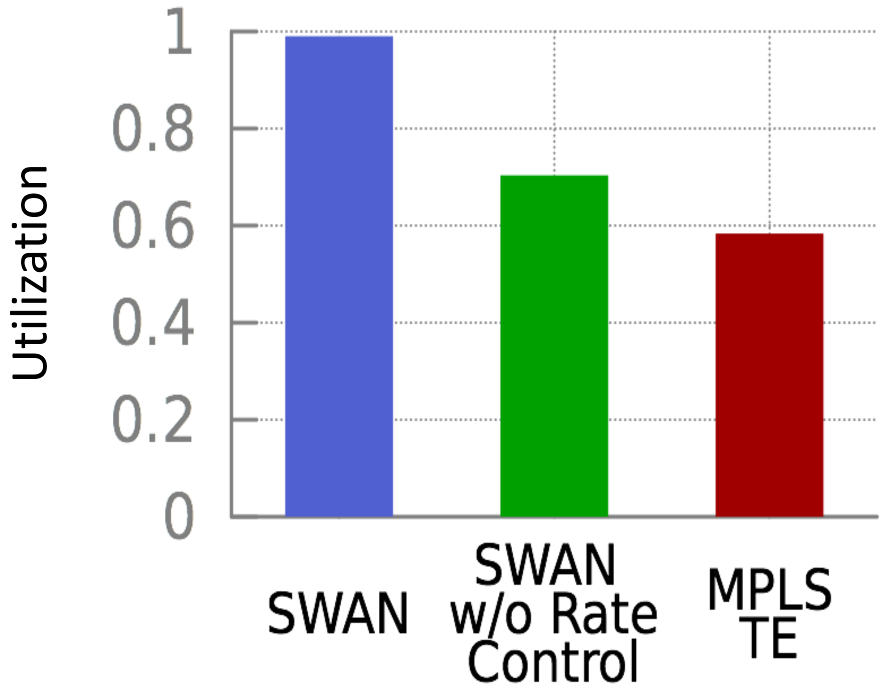


Traffic: (∀DC-pair) 125 TCP flows per class

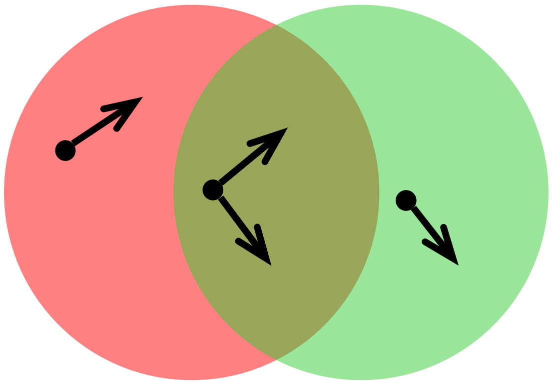
High utilization
SWAN's goodput:
98% of an optimal method

Flexible sharing
Interactive protected;
background rate-adapted

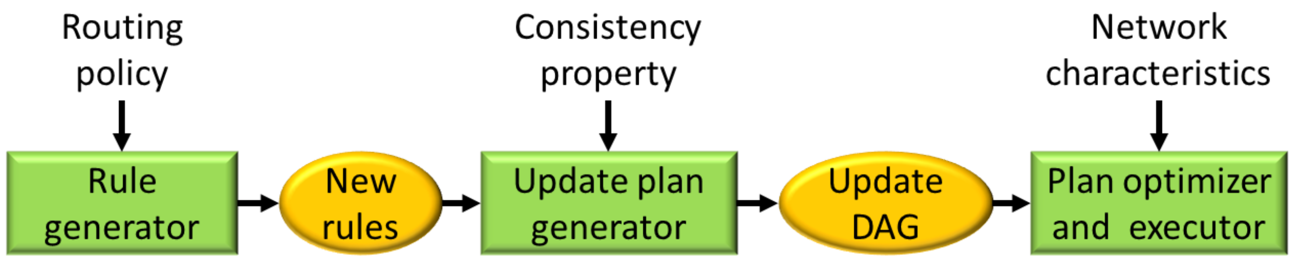
Data-driven Evaluation of 40+ DCs

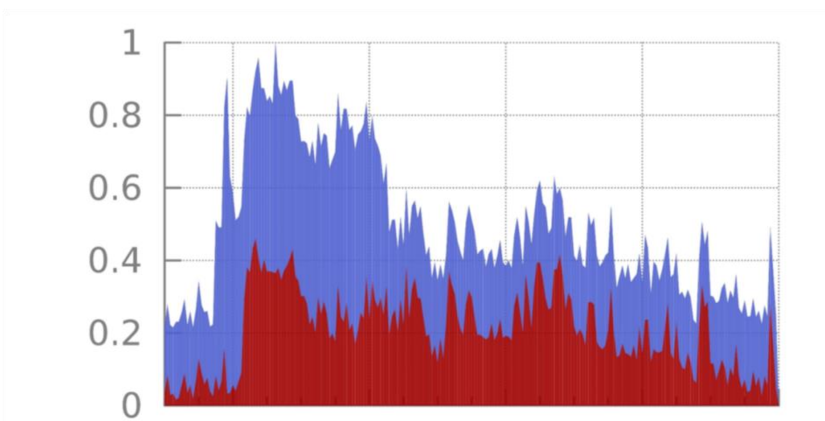


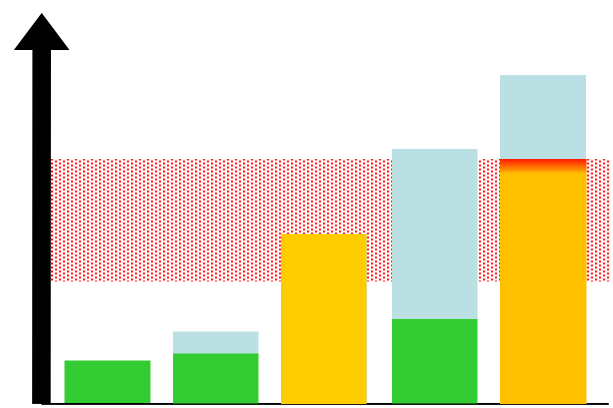
Summary



	None	Self	Downstream subset	Downstream all	Global
Eventual consistency	Always guaranteed				
Drop freedom	Impossible	Add before remove			
Memory limit	Impossible	Remove before add			
Loop freedom	Impossible		Rule dep. forest	Rule dep. tree	
Packet coherence	Impossible			Per-flow ver. numbers	Global ver. numbers
Bandwidth limit	Impossible				Staged partial moves







References

- Introducing consistent network updates was done in Mark Reitblatt et. al., SIGCOMM 2012
- For minimal loop-free updates and more see Ratul Mahajan et. al., HotNets 2013
- Deciding if a simple update schedule exists is hard was proven in Laurent Vanbever et. al., IEEE/ACM Trans. Netw. 2012
- For one of the first papers on loop-detection you can look at Robert Tarjan, Depth-first search and linear graph algorithms, 1972
- For more on the SWAN-project see Chi-Yao Hong et. al., SIGCOMM 2013

Thank You!

Questions & Comments?

www.disco.ethz.ch

Evaluation