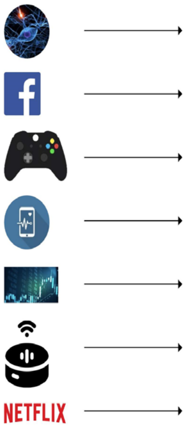
Disconnected cooperation in resilient networks and the algorithmic challenges of local fast re-routing

Stefan Schmid

Communication Networks

Critical infrastructure of digital society

- Popularity of **datacentric applications**: health, business, entertainment, social networking, AI/ML, etc.
- Evident during ongoing **pandemic**: online learning, online conferences, etc.
- Much traffic especially to, from, and inside **datacenters**



Facebook datacenter

Increasingly stringent dependability requirements!

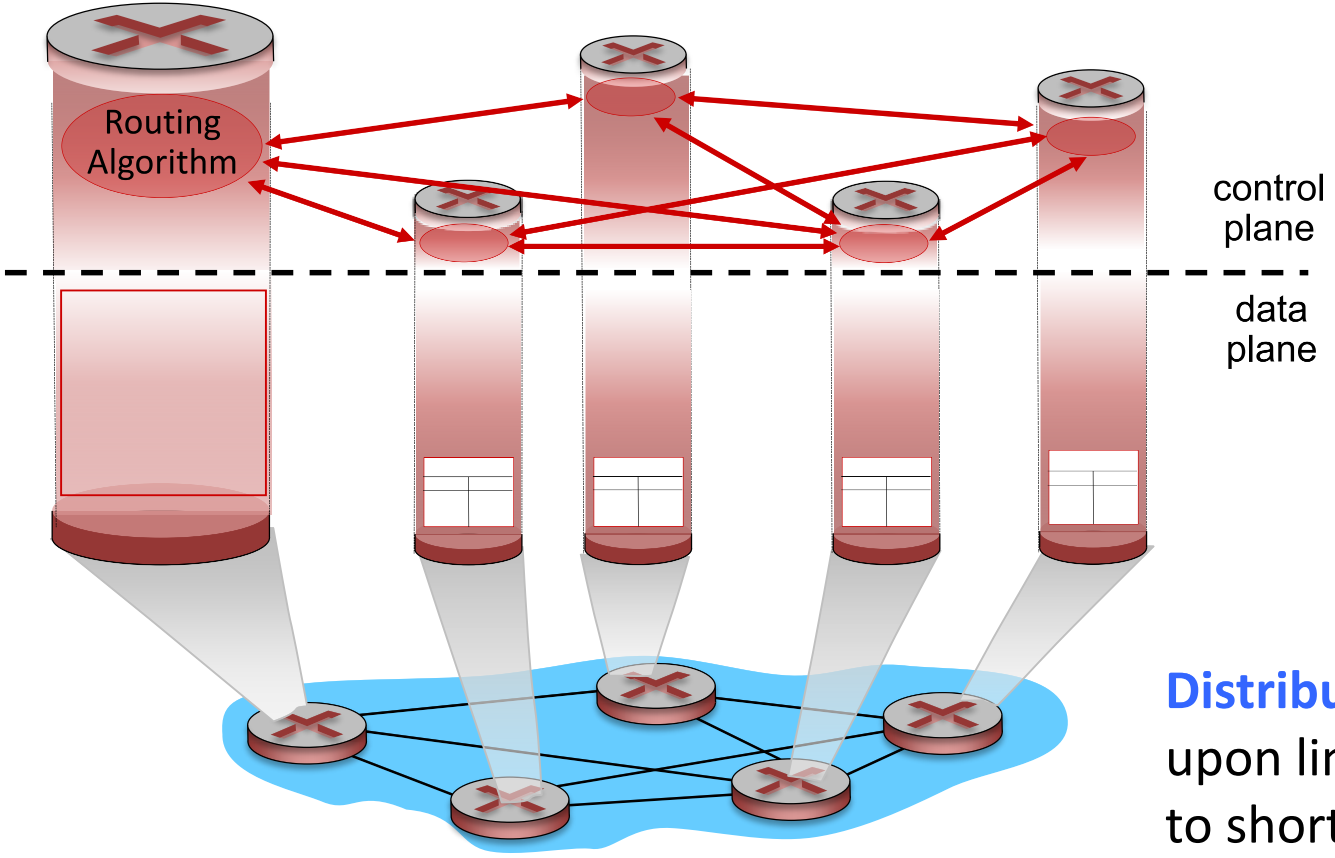
Roadmap

- A Brief Background on Resilient Networking
- Algorithms for Local Fast Re-Routing (FRR)
- Accounting for Congestion
- Accounting for Network Policy

Roadmap

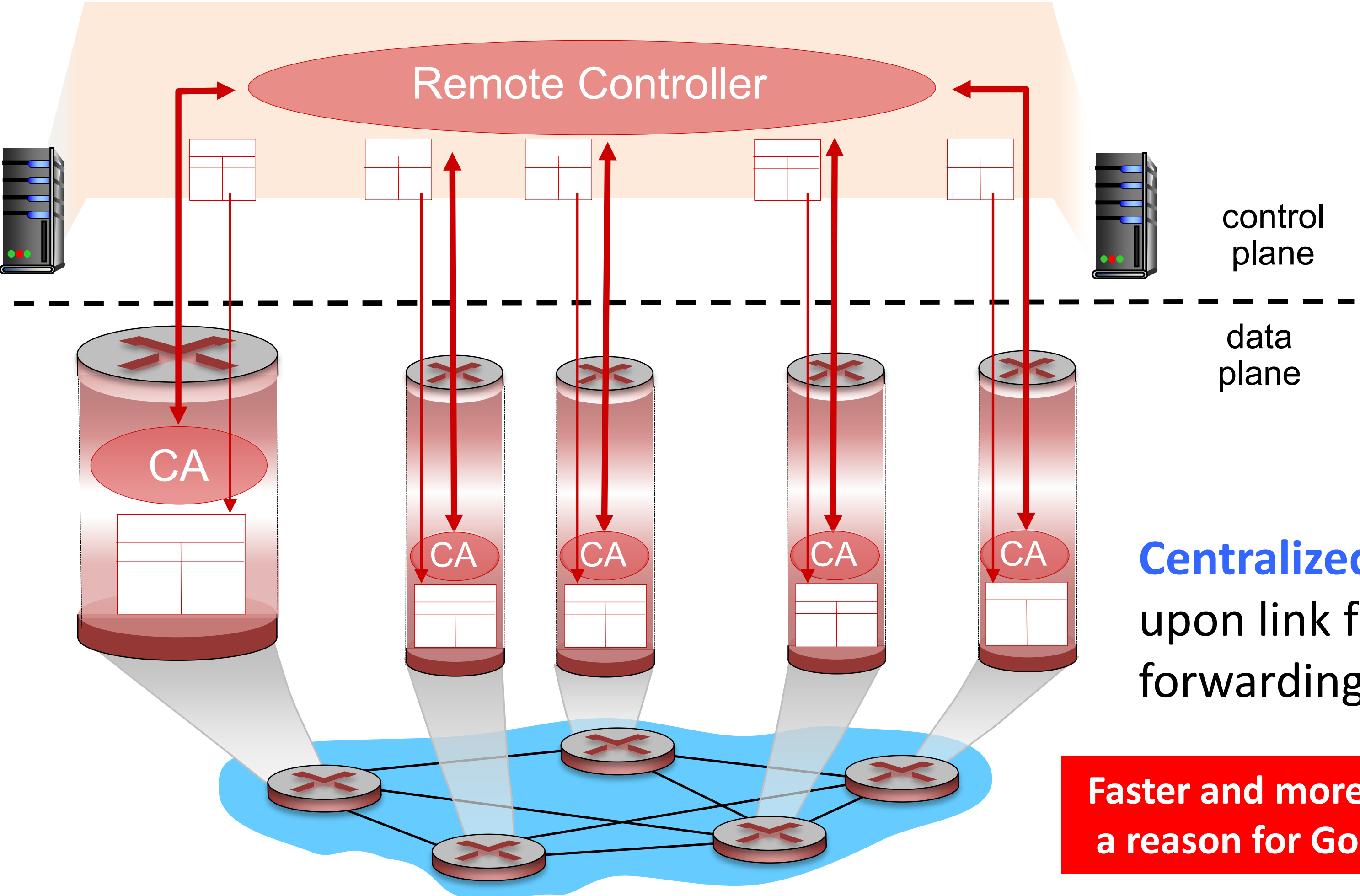
- **A Brief Background on Resilient Networking**
- Algorithms for Local Fast Re-Routing (FRR)
- Accounting for Congestion
- Accounting for Network Policy

Traditional Networks



Distributed algorithms:
upon link failure, ***reconverge***
to shortest paths

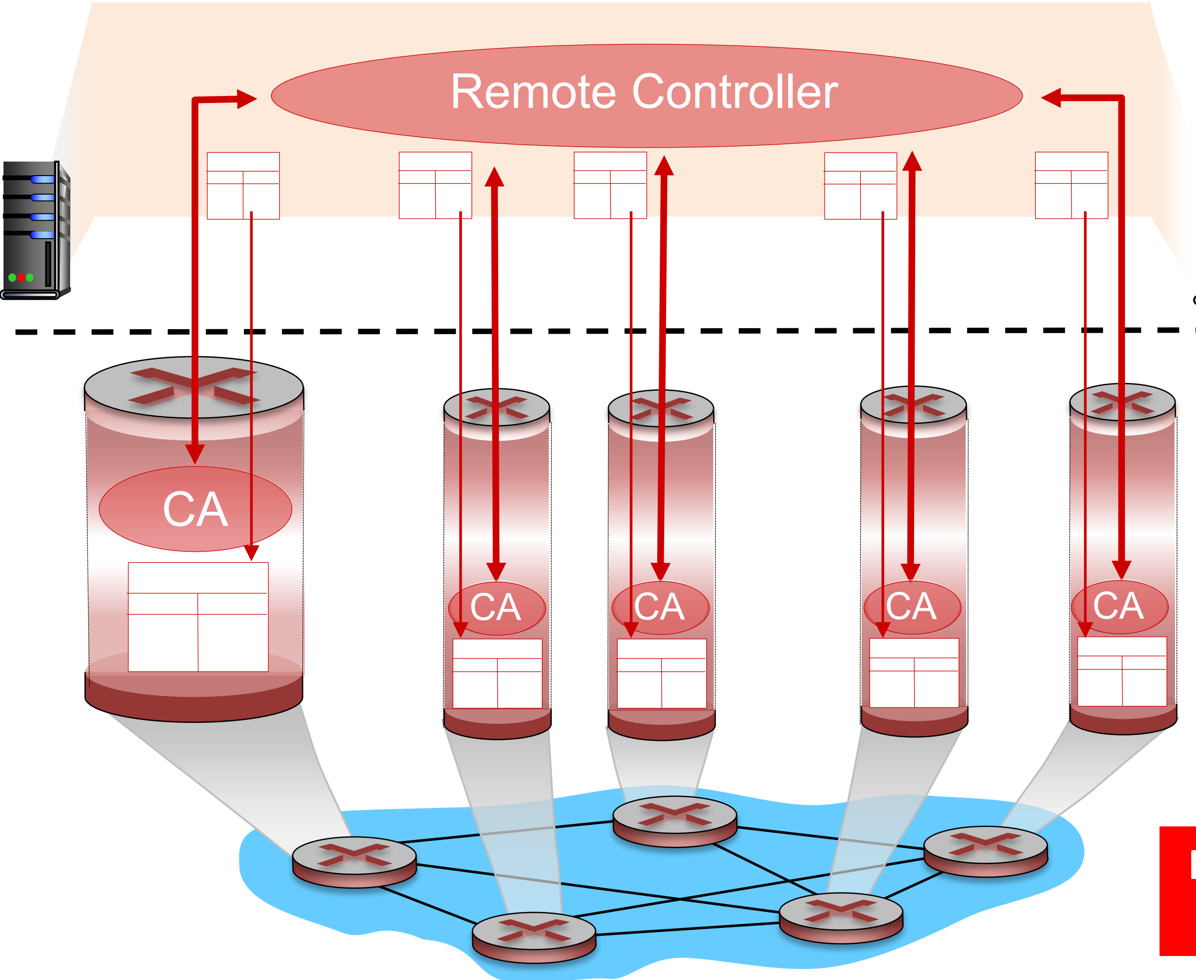
Software-Defined Networks (SDN)



Centralized algorithms: upon link failure, ***push*** new forwarding rules

Faster and more controlled reaction: a reason for Google's move to SDN!

Software-Defined Networks (SDN)



Centralized algorithms: upon link failure, ***push*** new forwarding rules

Faster and more controlled reaction: a reason for Google's move to SDN!

Restoration in control plane takes time -> packet drops!

Video shot taken from “Lemmings”
designed and developed by DMA Design

Failover: Control Plane vs Data Plane

- Slower reaction in the **control plane** than in the **data plane**

Minister of Education

vs

Teacher in the Classroom

Approaches for Failover

In Control Plane

- Distributed recomputation of shortest paths ("**re-convergence**")
- Centralized recomputation of paths (SDN)
- **Link-reversal** algorithms (e.g., Gafni et al.)

vs

In Data Plane

- Static forwarding table
- Rules pre-installed ***before*** failures are known

Approaches for Failover

In Control Plane

Slow but "globally informed".

- Distributed recomputation of shortest paths ("**re-convergence**")
- Centralized recomputation of paths (SDN)
- **Link-reversal** algorithms (e.g., Gafni et al.)

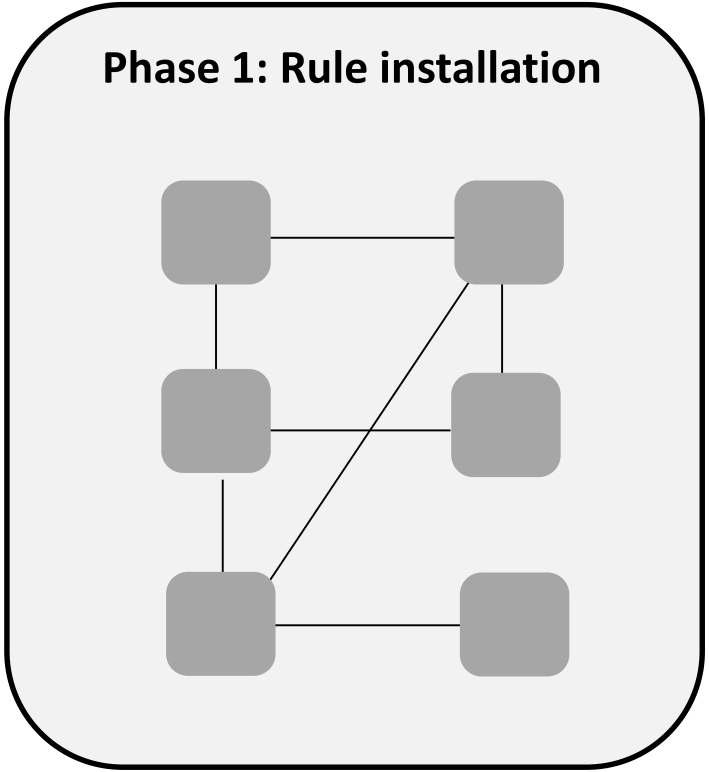
vs

In Data Plane

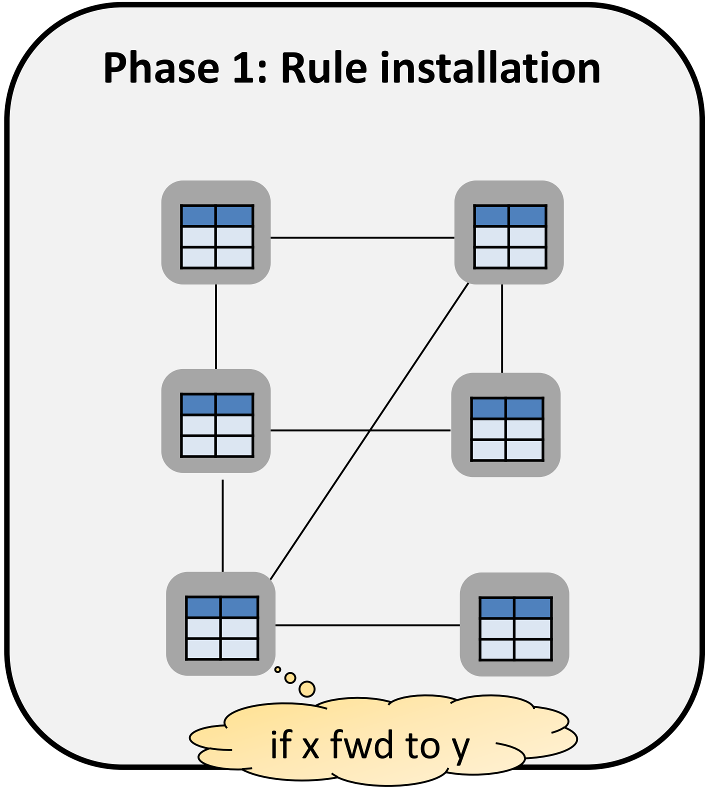
Fast but "local knowledge".

- Static forwarding table
- Rules pre-installed ***before*** failures are known

The FRR Problem

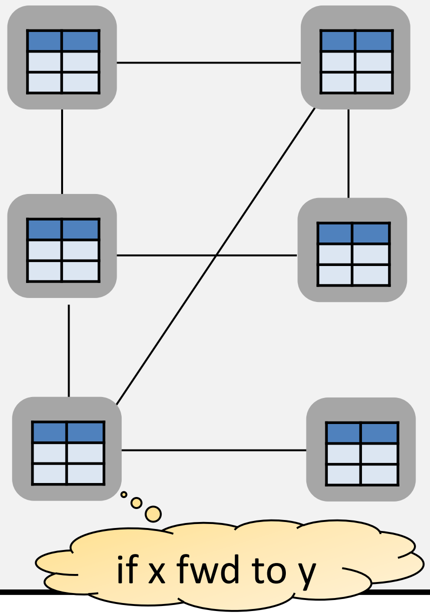


The FRR Problem

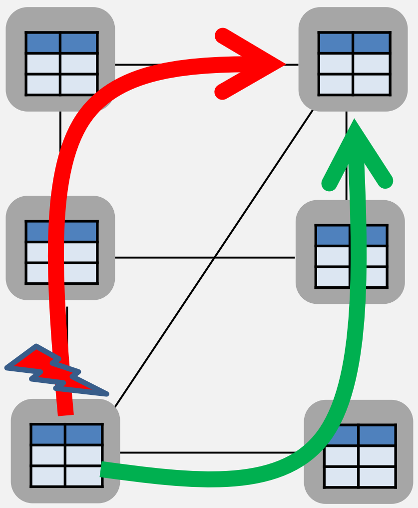


The FRR Problem

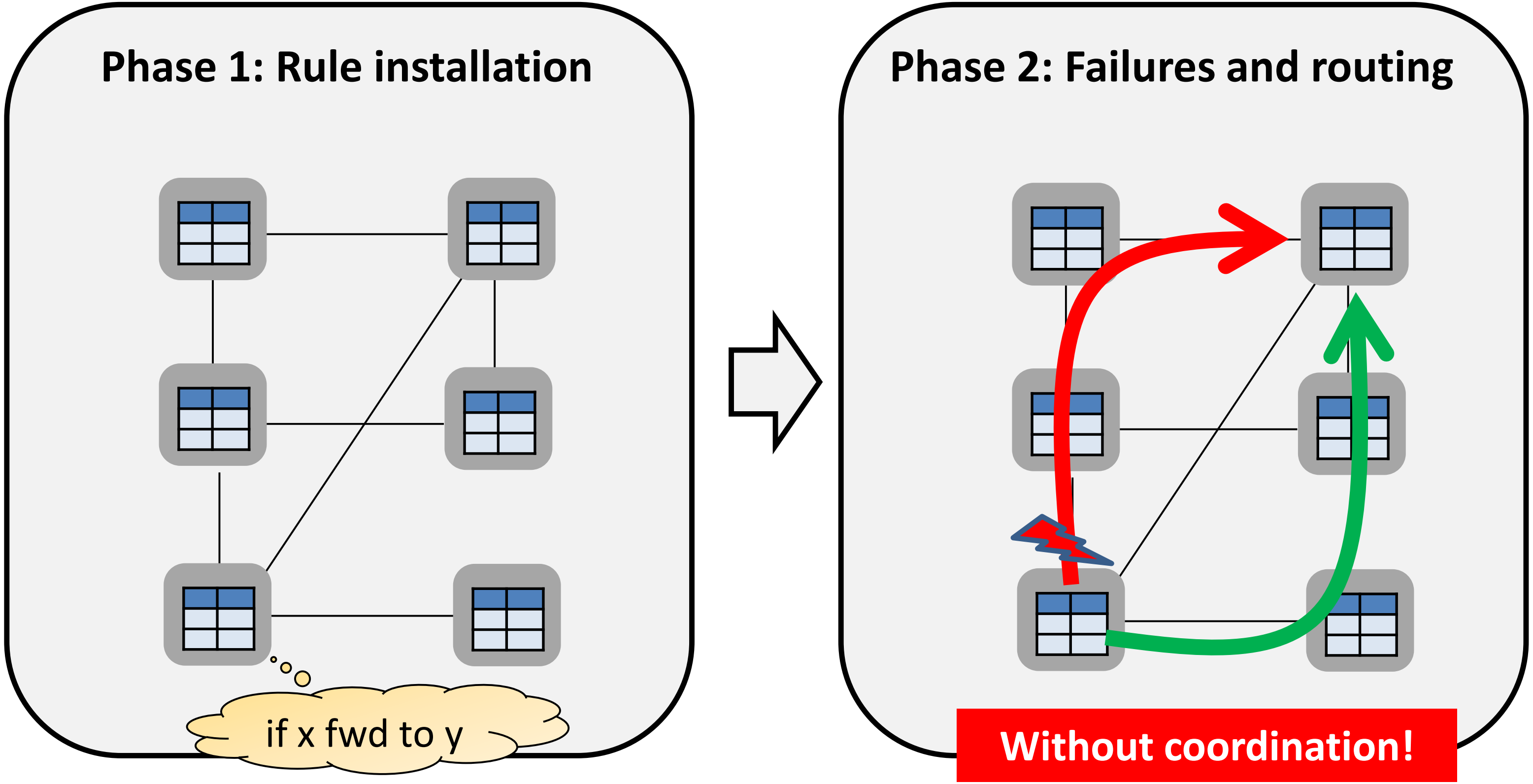
Phase 1: Rule installation



Phase 2: Failures and routing



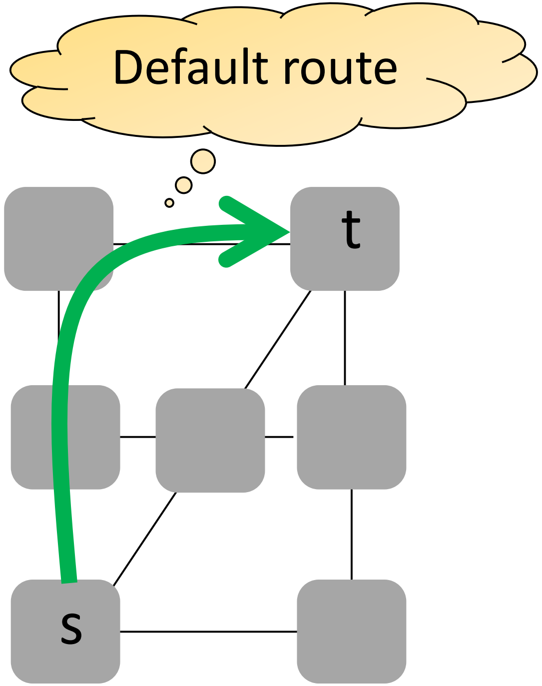
The FRR Problem



The FRR Problem

- **Pre-installed** local-fast failover rules
  - *Can depend on local failures and, e.g., destination, inport, source*
- **At runtime**, rules are just *"executed"*

Advantage: no need to wait for reconvergence.



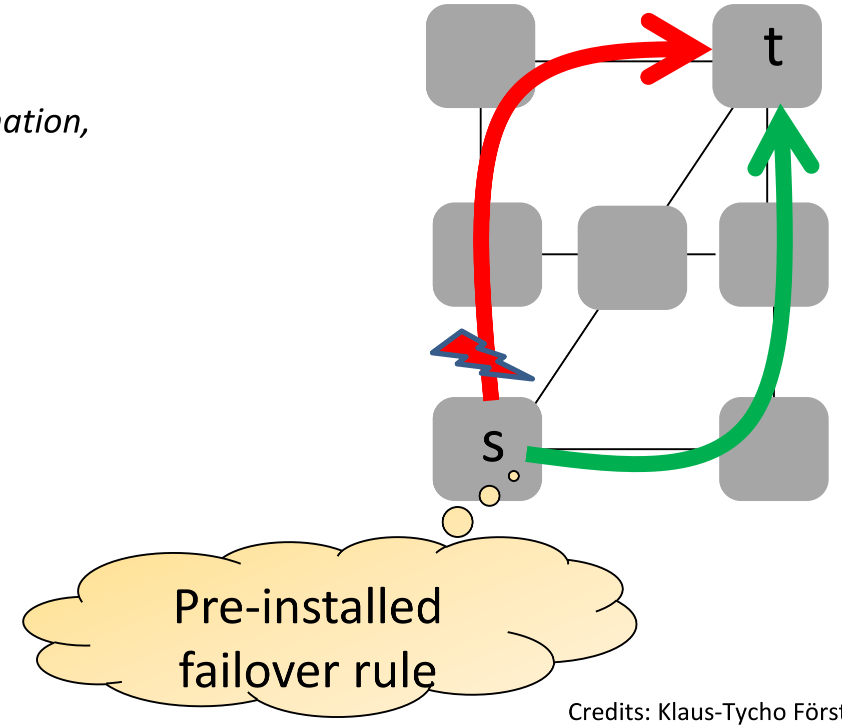
Credits: Klaus-Tycho Förster

The FRR Problem

- **Pre-installed** local-fast failover rules
  - *Can depend on local failures and, e.g., destination, inport, source*
- **At runtime**, rules are just *"executed"*

Advantage: no need to wait for reconvergence.

Good alternative under 1 failure!



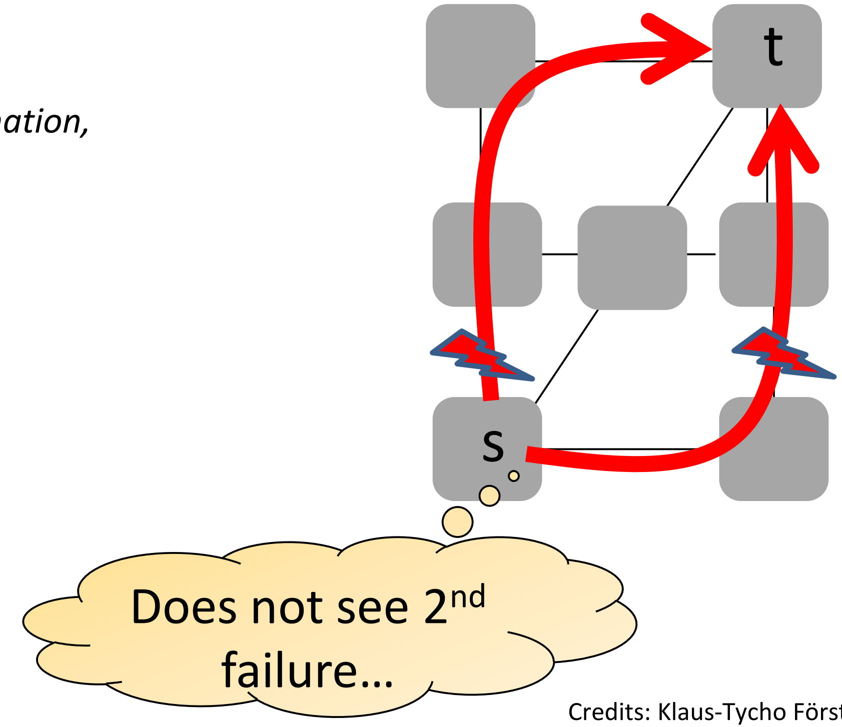
Credits: Klaus-Tycho Förster

The FRR Problem

- **Pre-installed** local-fast failover rules
  - *Can depend on local failures and, e.g., destination, inport, source*
- **At runtime**, rules are just *"executed"*

Advantage: no need to wait for reconvergence.

Good alternative under 1 failure!



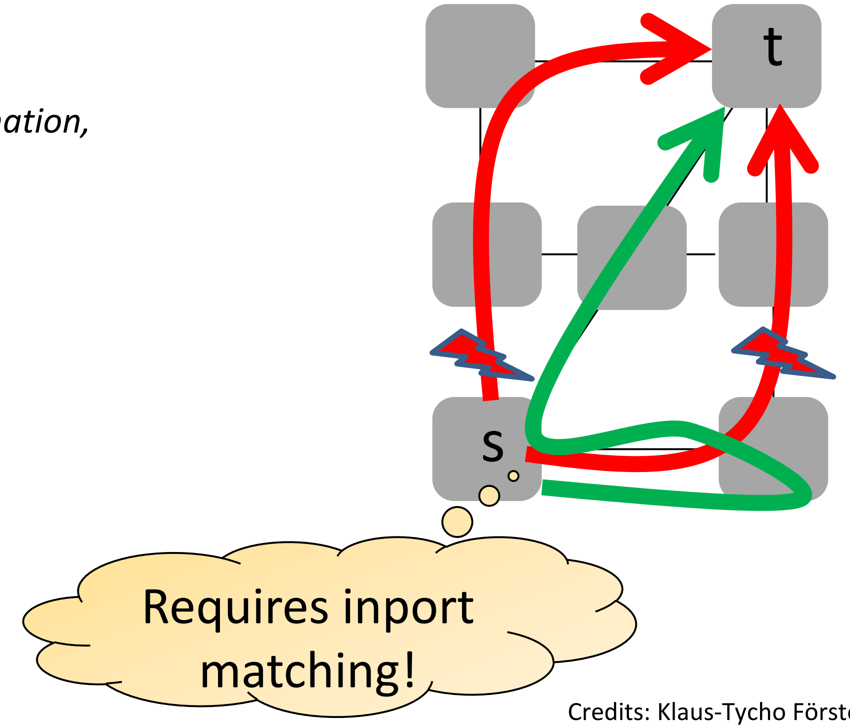
Credits: Klaus-Tycho Förster

The FRR Problem

Can get complex under multiple failures..

- **Pre-installed** local-fast failover rules
  - *Can depend on local failures and, e.g., destination, inport, source*
- **At runtime**, rules are just *"executed"*

Advantage: no need to wait for reconvergence.



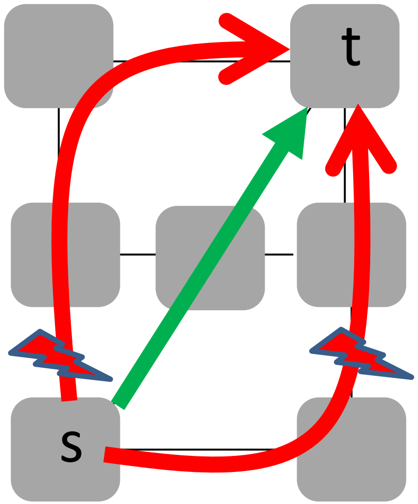
Credits: Klaus-Tycho Förster

The FRR Problem

With global knowledge: simpler!

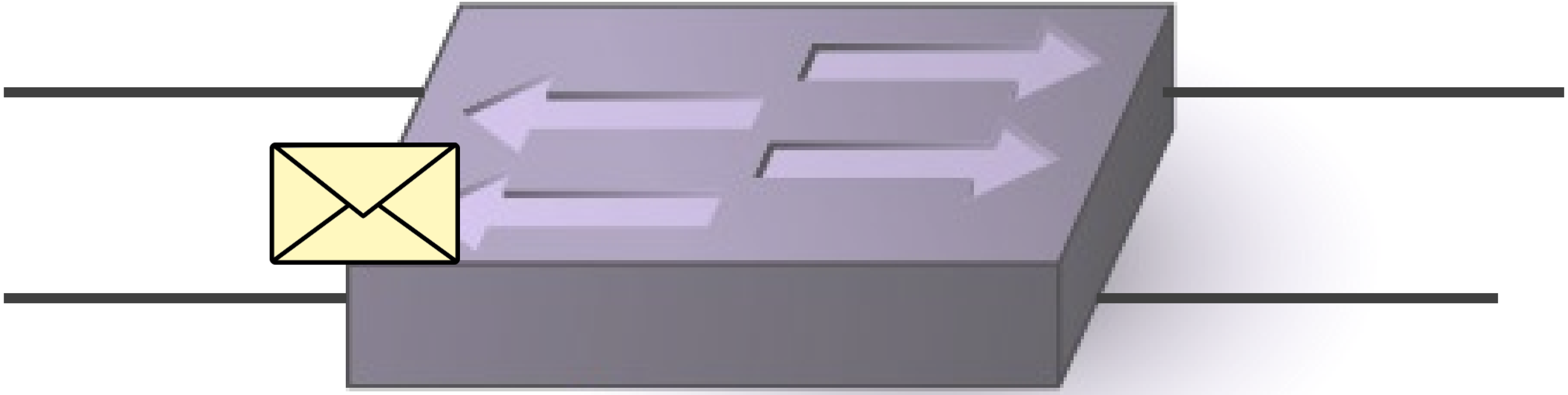
- **Pre-installed** local-fast failover rules
  - *Can depend on local failures and, e.g., destination, inport, source*
- **At runtime**, rules are just *"executed"*

Advantage: no need to wait for reconvergence.



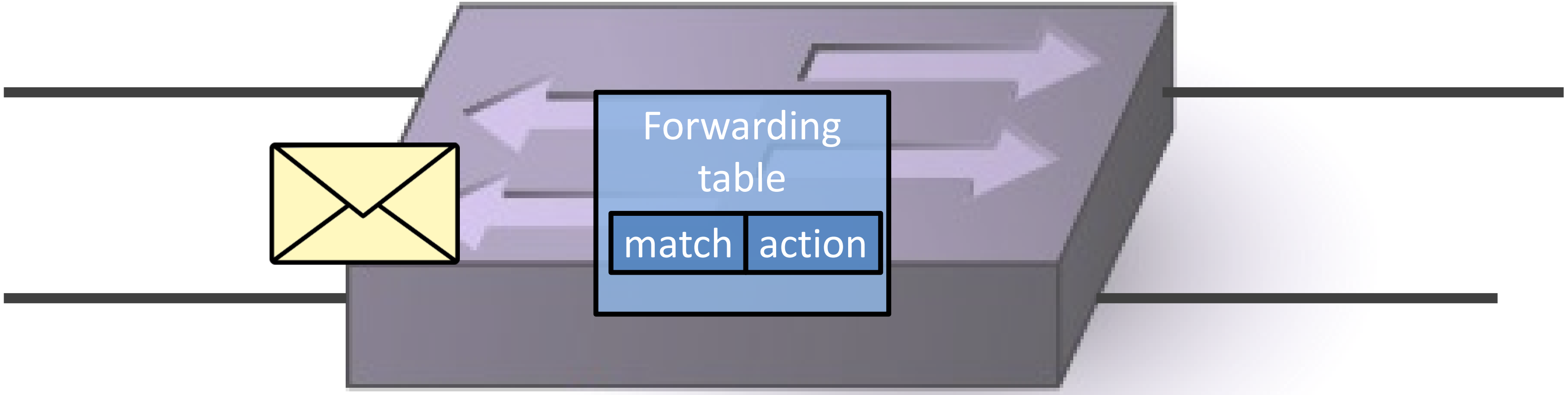
Credits: Klaus-Tycho Förster

What information is locally available in a switch for handling a packet?



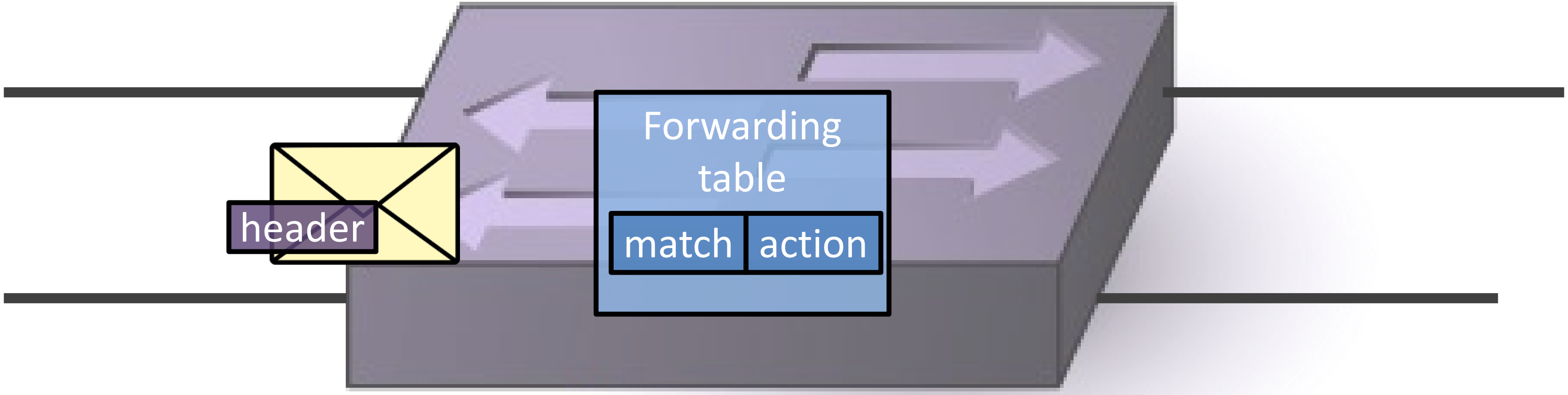
Credits: Marco Chiesa

Locally Available Information:
The Forwarding Table: Match -> Action



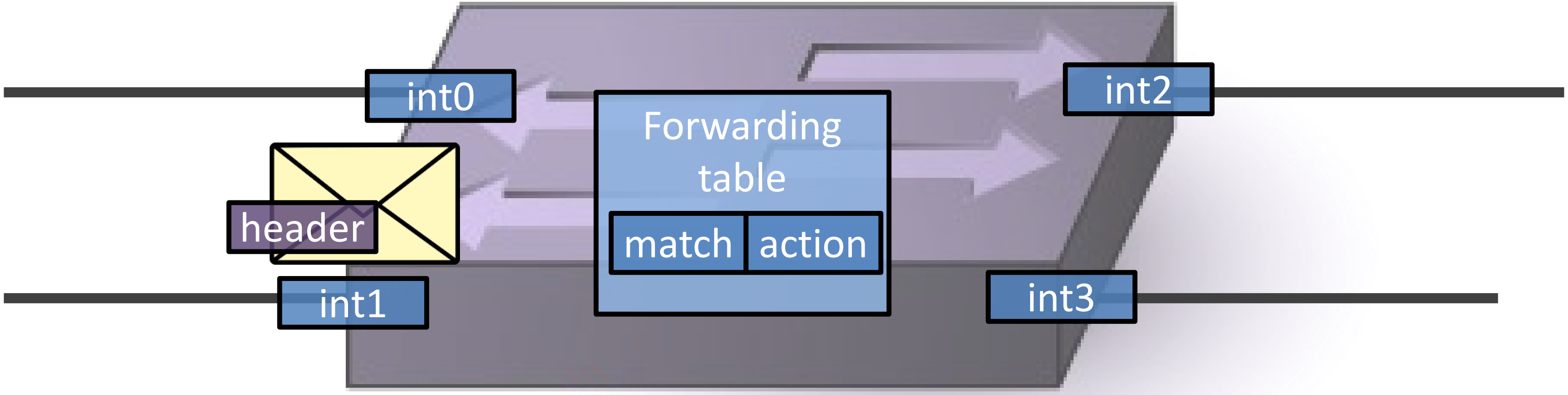
Credits: Marco Chiesa

Locally Available Information:
The Packet Header



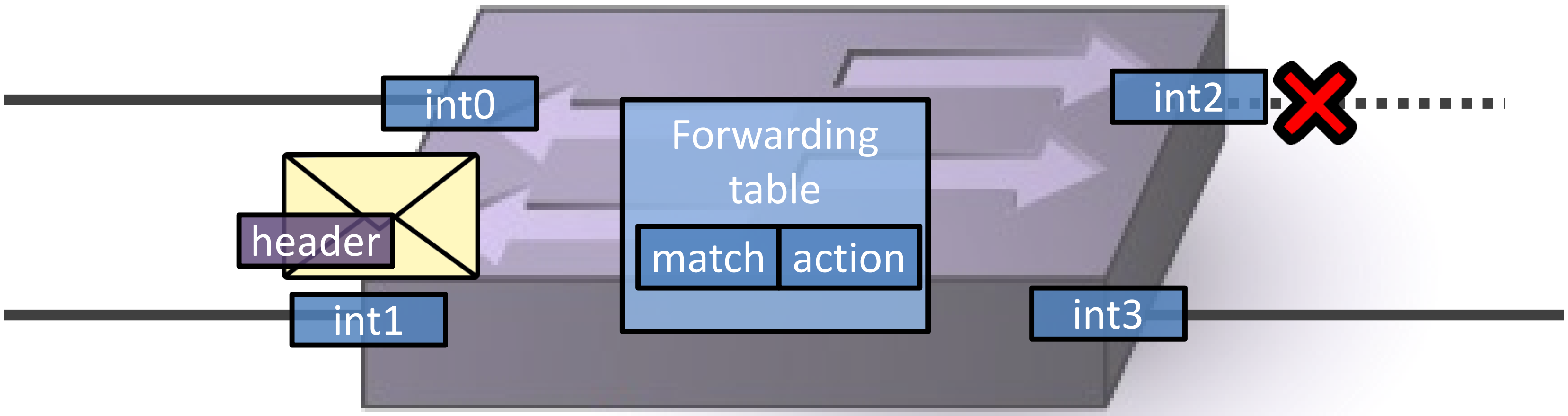
Credits: Marco Chiesa

Locally Available Information:
The Inport of the Received Packet



Credits: Marco Chiesa

Locally Available Information: The Outgoing Port Depends on Failed Links



Credits: Marco Chiesa

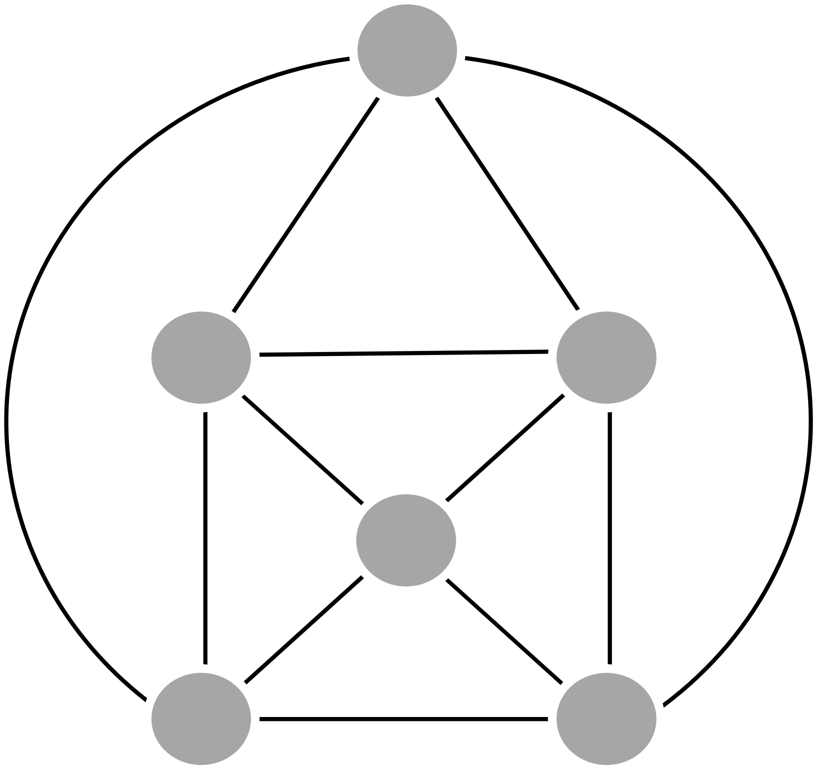
Raises an Interesting Question

Can we pre-install local fast failover rules which ensure reachability under multiple failures? *In particular: How many failures* can be tolerated by static forwarding tables?

Roadmap

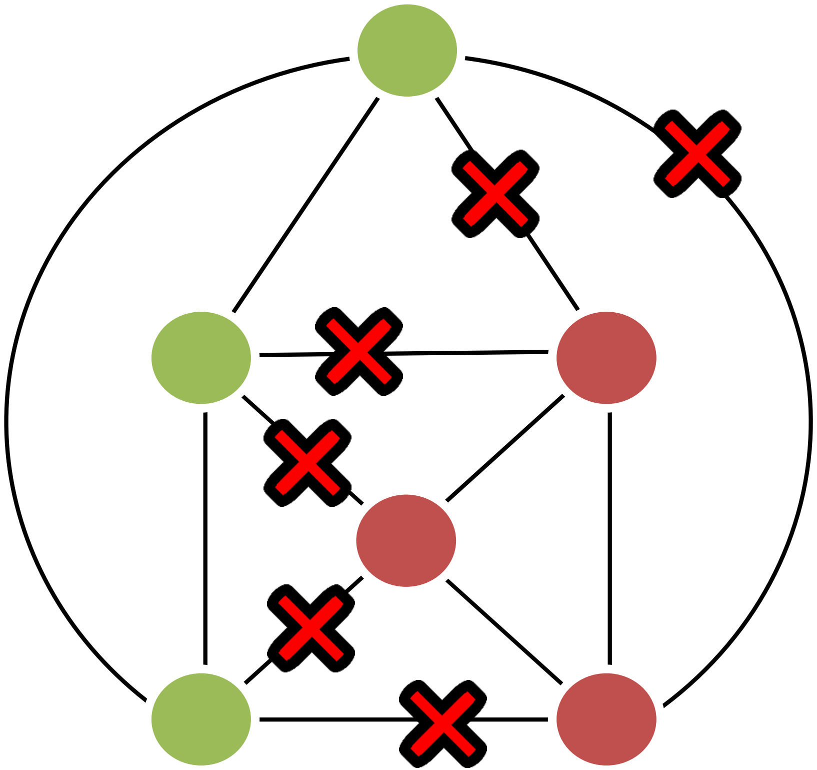
- A Brief Background on Resilient Networking
- **Algorithms for Local Fast Re-Routing (FRR)**
- Accounting for Congestion
- Accounting for Network Policy

So: How many failures can be tolerated by static forwarding tables?



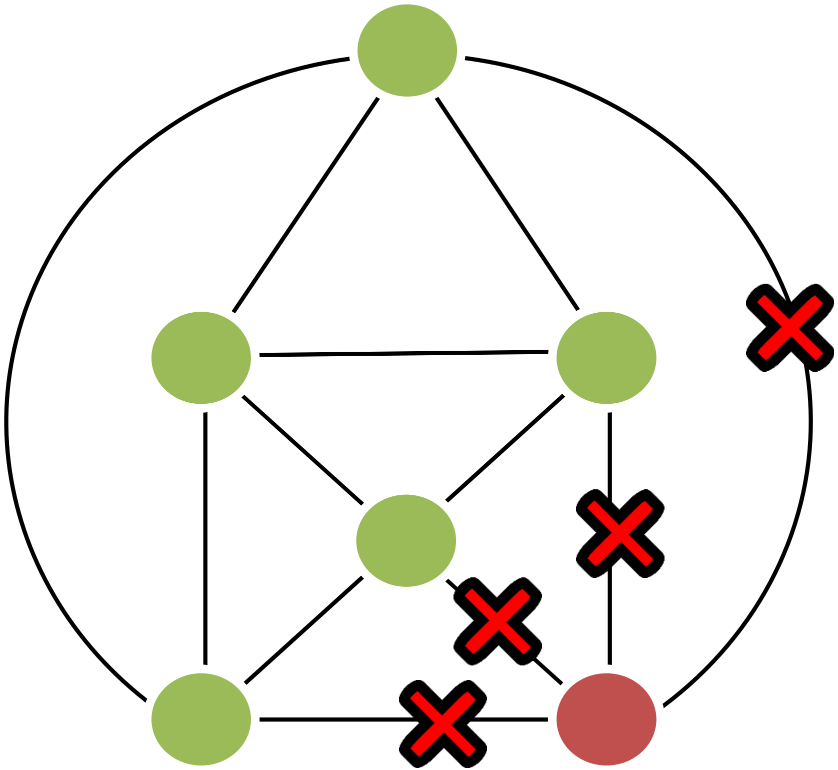
Credits: Marco Chiesa

If we partition the network, there is not much to do



Credits: Marco Chiesa

The connectivity k of a network N: the minimum number of link deletions that partitions N



The connectivity of this network is *four*

Credits: Marco Chiesa

Resilience Criteria

Ideal resilience

Given a k-connected graphs, we can tolerate *any k-1 link failures.*

Perfect resilience

Any source s can always reach any destination t as long as the unterlying network is *physically connected*.

Can this be achieved? Assume undirected link failures.

Resilience Criteria

Ideal resilience

Given a *k*-connected graphs, we can tolerate *any k-1 link failures.*

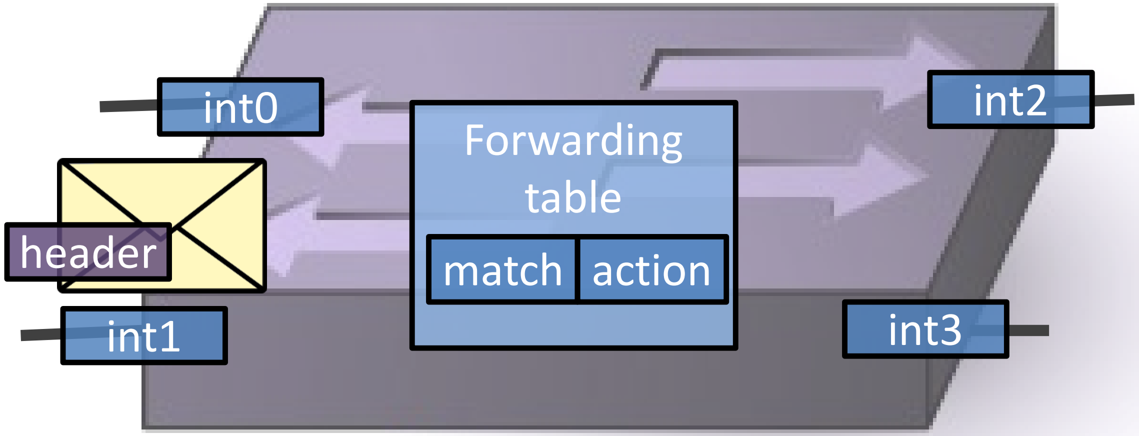
Perfect resilience

Any source *s* can always reach any destination *t* as long as the unterlying network is *physically connected*.

Can this be achieved? Assume undirected link failures.

Spectrum of Models

Recall our switch model:



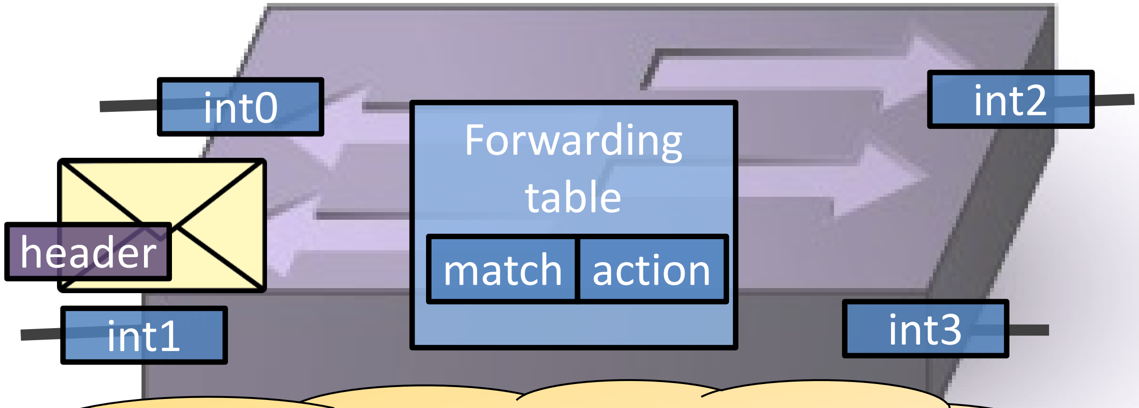
Achievable resilience depnds on *what can be matched*:

Per-destination	Per source	Incoming port	Probabilistic forwarding	Packet header rewriting

Credits: Marco Chiesa

Spectrum of Models

Recall our switch model:



Can carry global information, but often undesirable

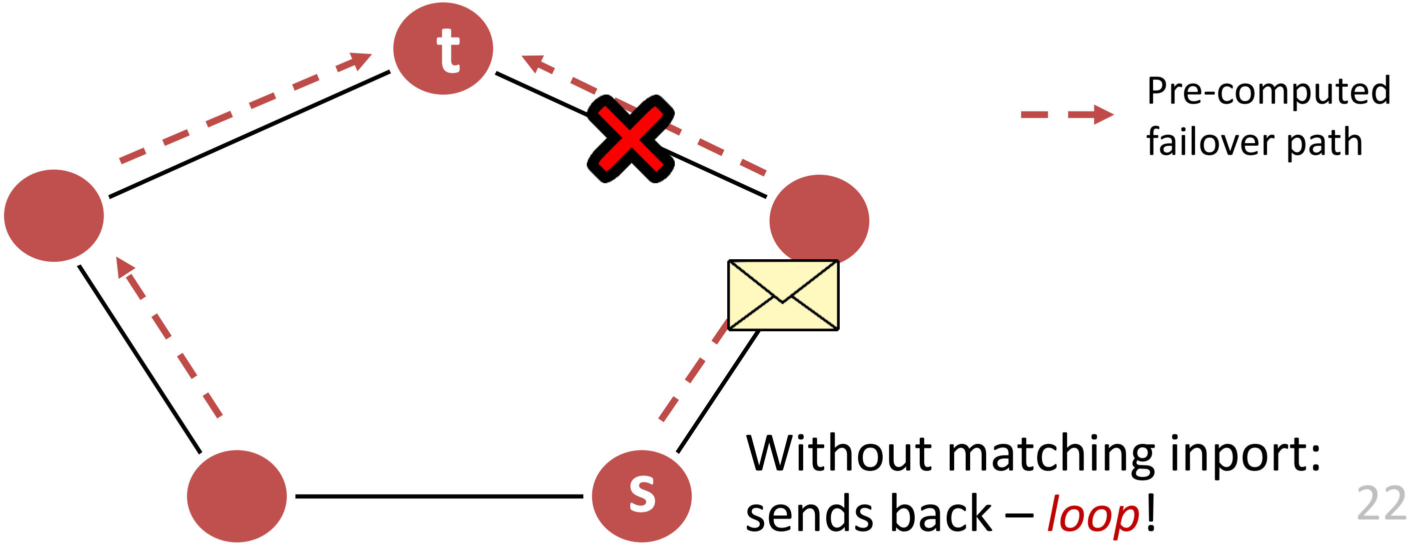
Achievable resilience depnds on

Per-destination	Per source	Incoming port	Probabilistic forwarding	Packet header rewriting

Credits: Marco Chiesa

Per-destination routing *cannot cope* with *even one* link failure

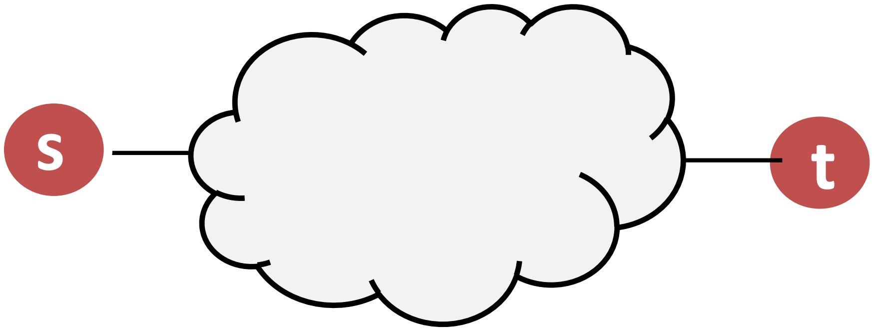
Per-destination	Per source	Incoming port	Probabilistic forwarding	Packet header rewriting	Resiliency
X					0



Without matching inport: sends back – *loop*!

Can we achieve k – 1 resiliency in k-connected graph here?

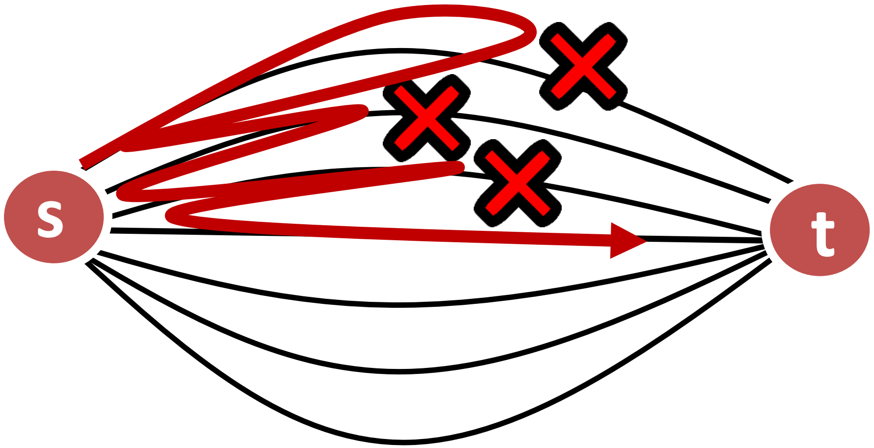
Per-destination	Per source	Incoming port	Probabilistic forwarding	Packet header rewriting	Resiliency
X	X	X			?



Credits: Marco Chiesa

Can we achieve k – 1 resiliency in k-connected graph here?

Per-destination	Per source	Incoming port	Probabilistic forwarding	Packet header rewriting	Resiliency
X	X	X			Yes



k disjoint paths: try one after the other, routing *back to source* each time.

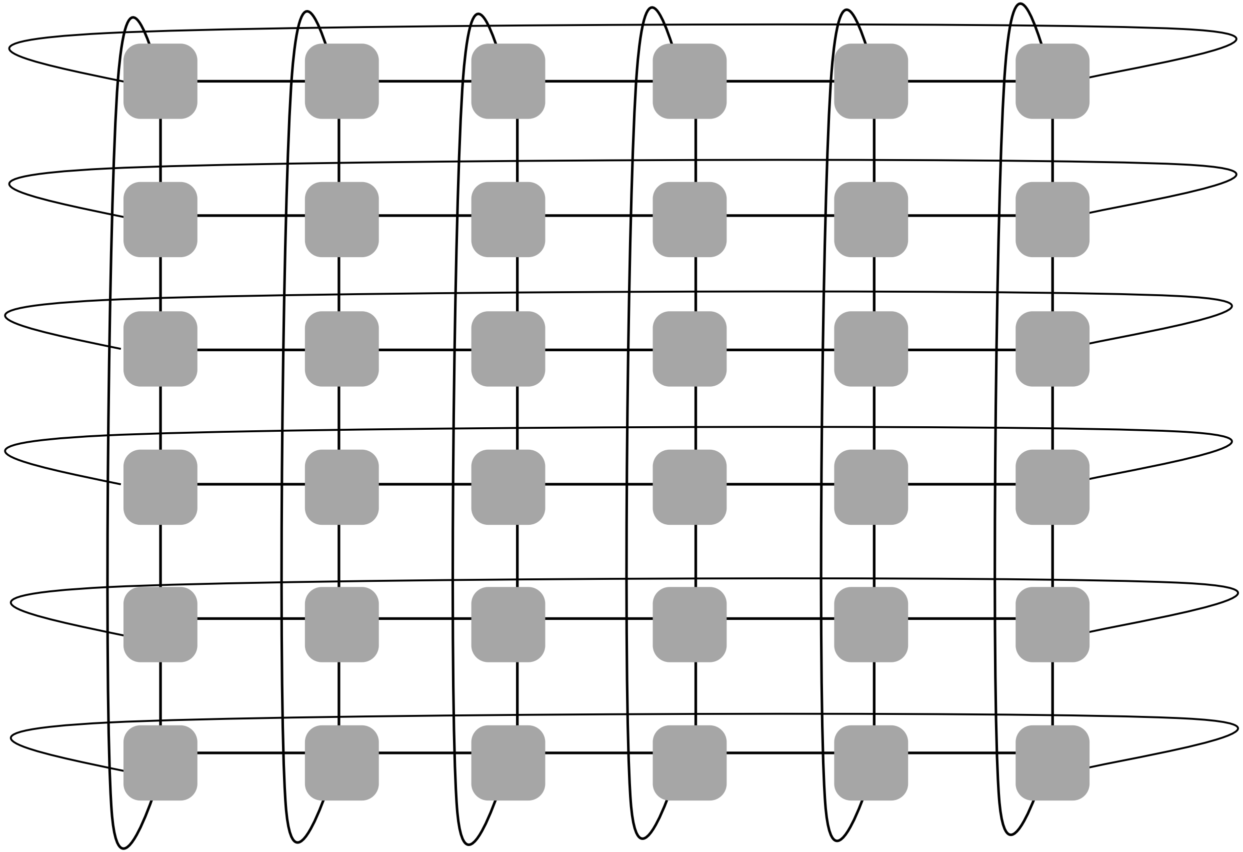
Credits: Marco Chiesa

Can we achieve k – 1 resiliency in k-connected graph here?

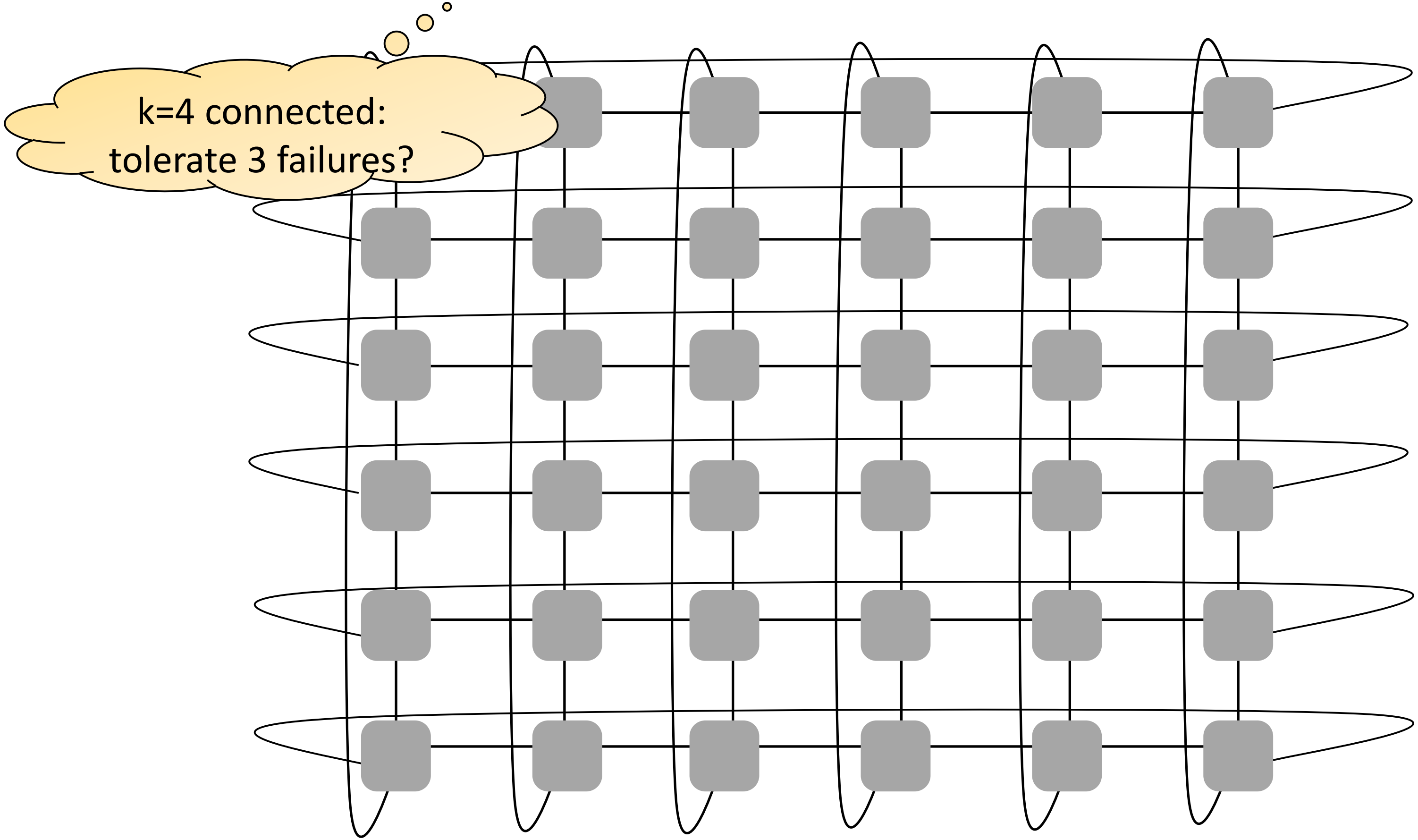
Per-destination	Per source	Incoming port	Probabilistic forwarding	Packet header rewriting	Resiliency
X		X			?

What about this scenario? Practically important. From now on called "ideal resilience".

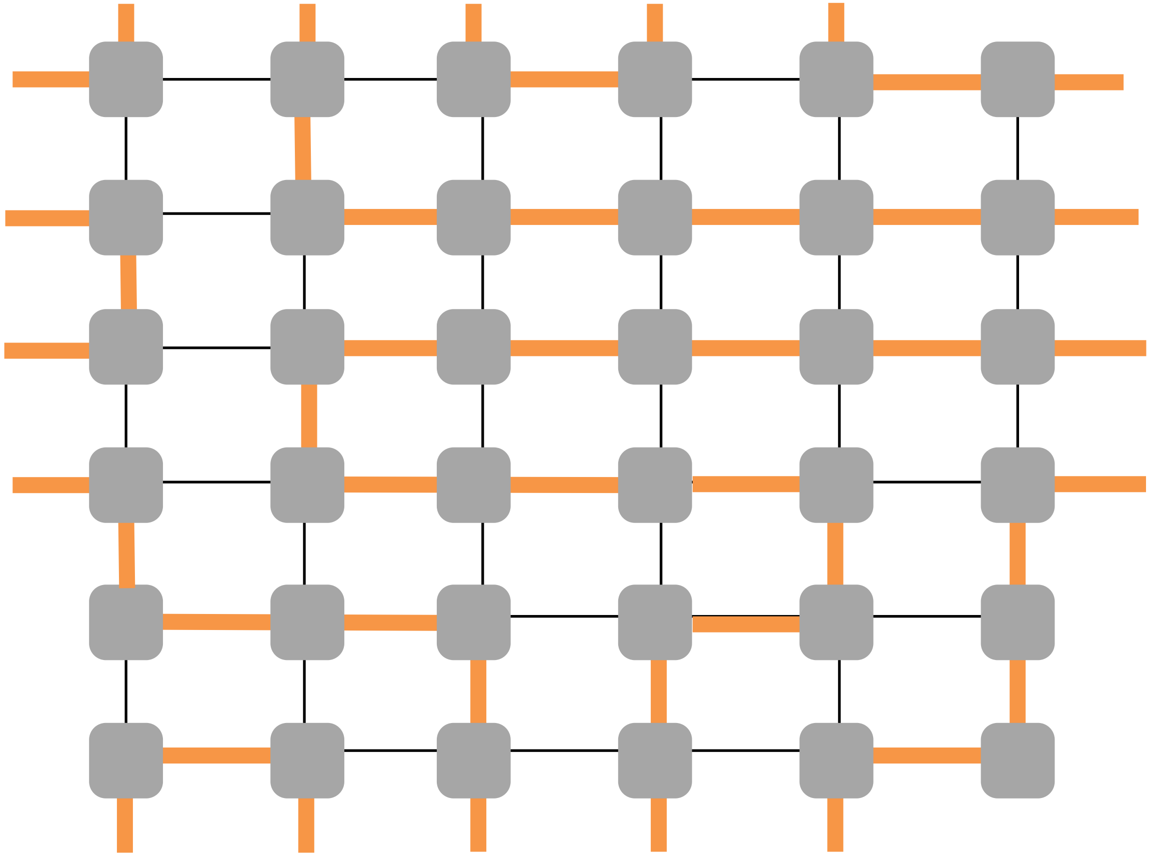
Ideal Resilience: Example 2-dim Torus?



Ideal Resilience: Example 2-dim Torus?



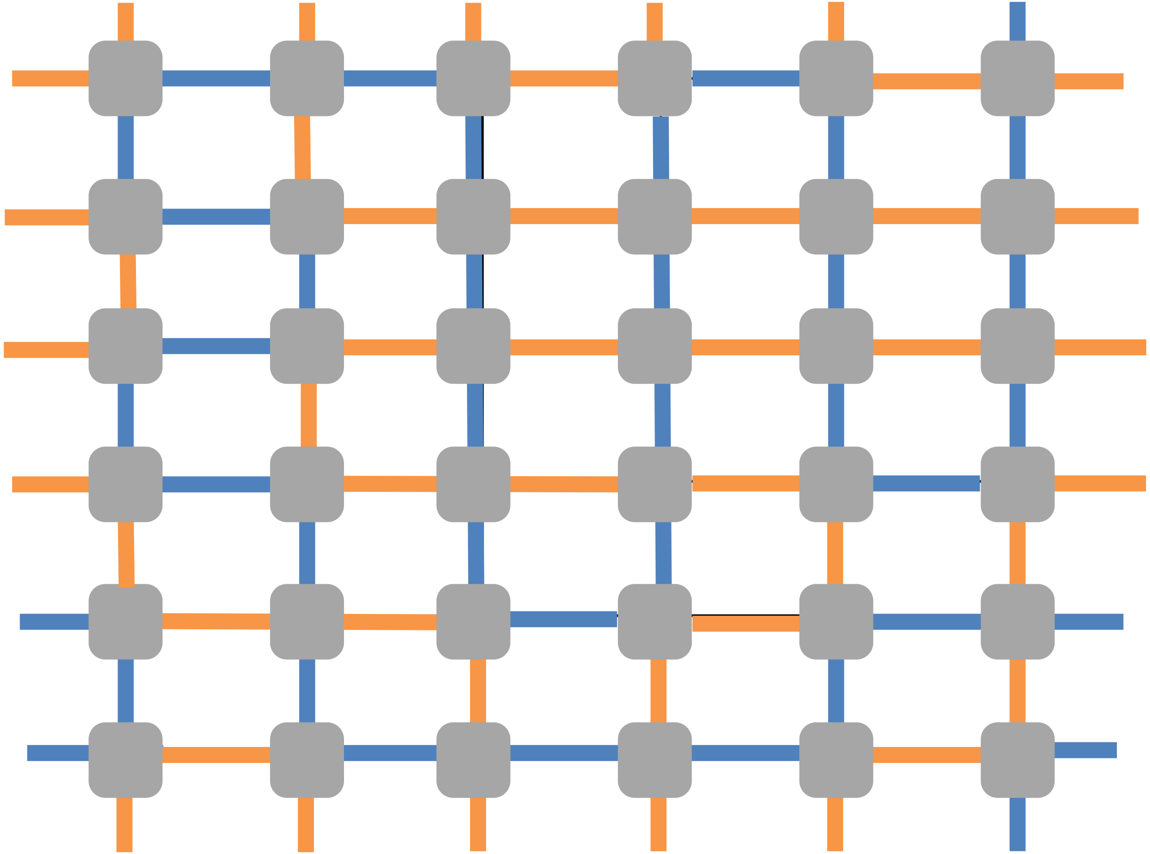
Idea: Decomposition into Hamilton Cycles



- Decompose torus into 2-edge-disjoint Hamilton Cycles (HC)

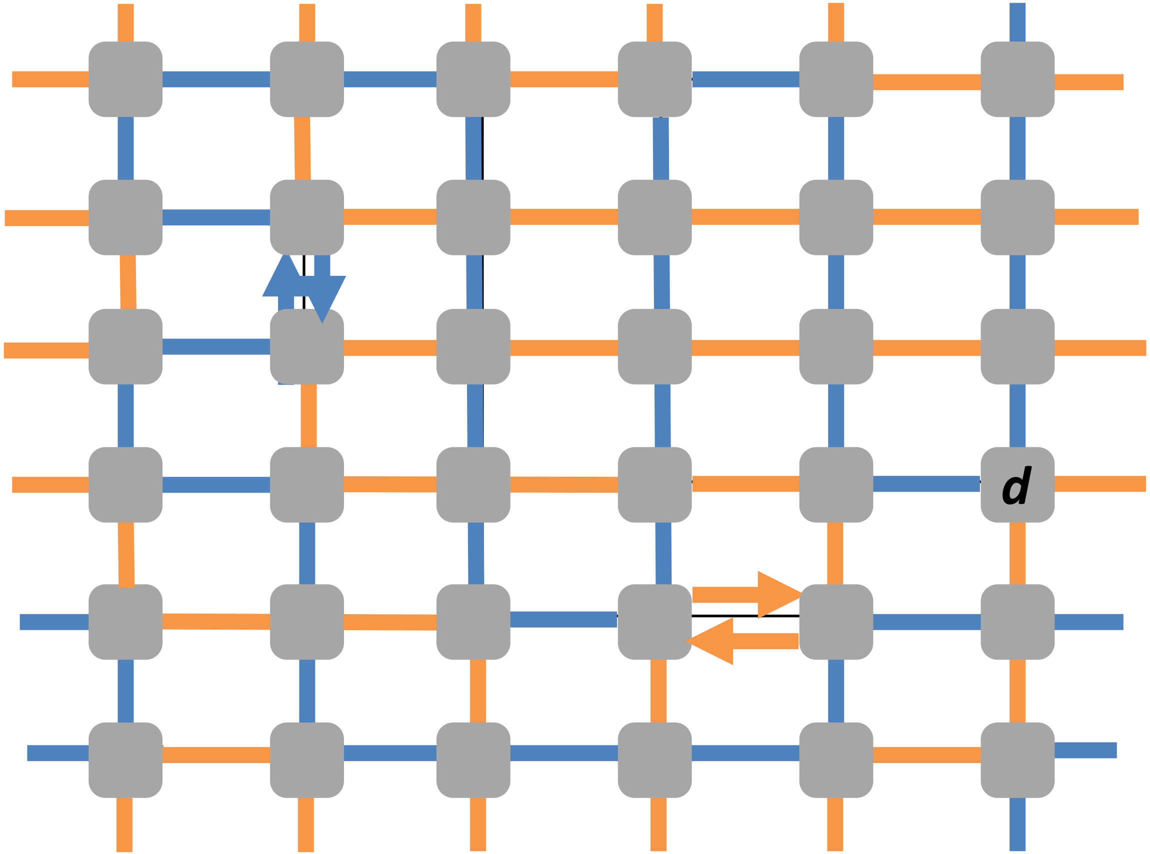
1st Hamilton cycle

Idea: Decomposition into Hamilton Cycles



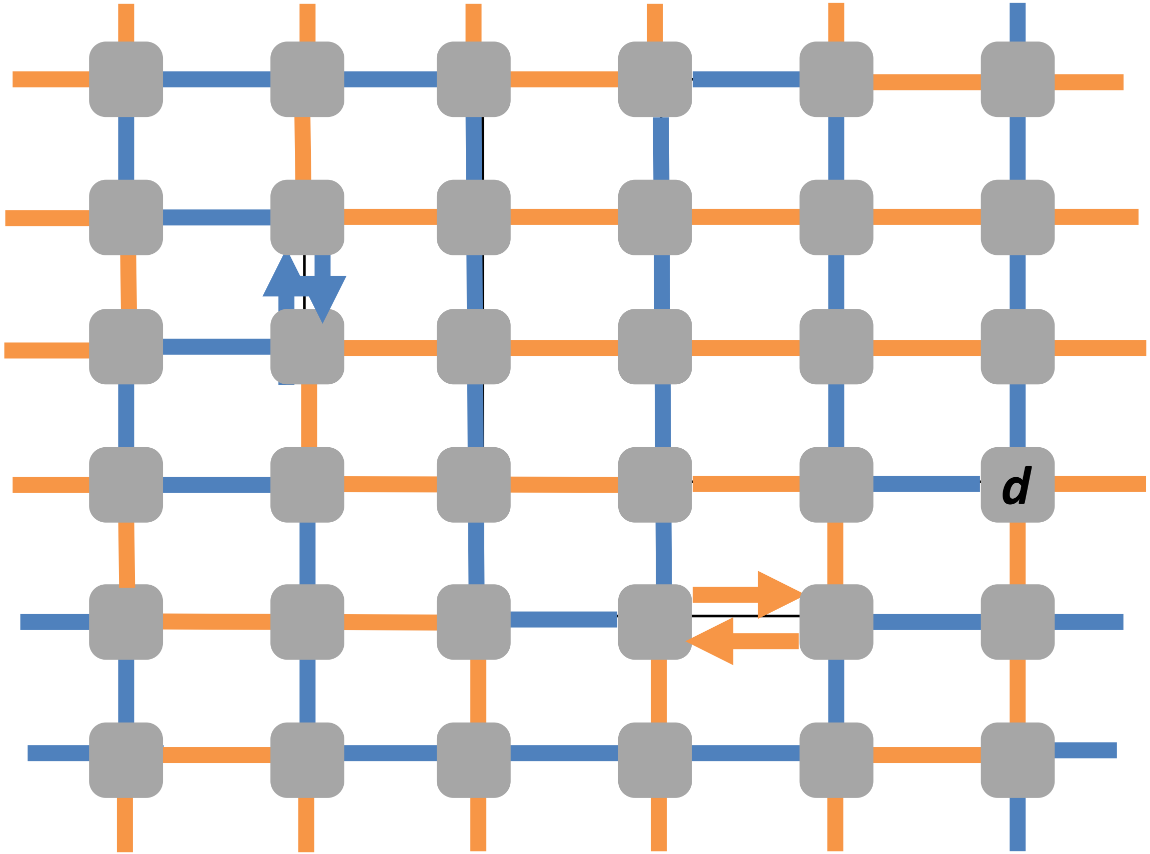
- Decompose torus into 2-edge-disjoint Hamilton Cycles (HC)

Idea: Decomposition into Hamilton Cycles



- Decompose torus into 2-edge-disjoint Hamilton Cycles (HC)
- Can route in both directions: *4-arc-disjoint* HCs

Idea: Decomposition into Hamilton Cycles



- Decompose torus into 2-edge-disjoint Hamilton Cycles (HC)
- Can route in both directions: *4-arc-disjoint* HCs

3-resilient routing to destination d:

- go along *1st directed HC*, if hit failure, *reverse* direction
- if again failure switch to *2nd HC*, if again failure *reverse direction*
- No more failures possible!

Ideal Resilience with Hamilton Cycles

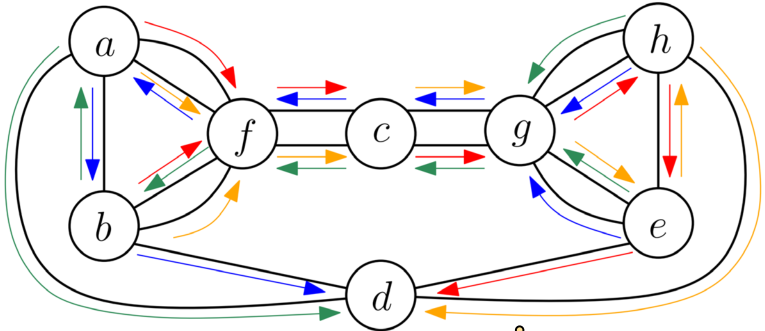
Chiesa et al.: if k-connected graph has k arc disjoint Hamilton Cycles, k-1 resilient routing can be constructed!

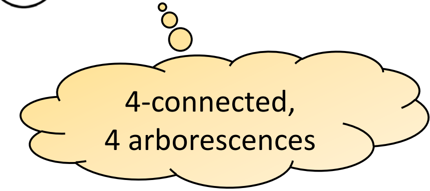
What about graphs which cannot be decomposed into Hamilton cycles?

Chiesa et al. **On the Resiliency of Static Forwarding Tables.** IEEE/ACM Transactions on Networking (ToN), 2017.

Ideal Resilience in General k-Connected Graphs

- Use directed trees (i.e. *arborescences*) instead of Hamilton cycles
  - *Arc-disjoint*, spanning, and *rooted* at destination
- Classic result: k-connectivity guarantees k-arborescence decomposition



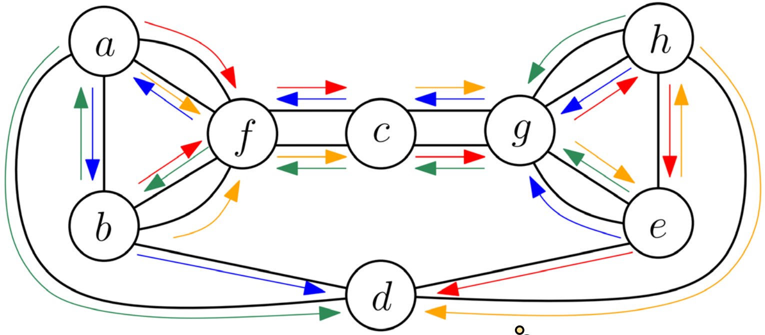


Basic idea:
- Idea: route towards root on one arborescence
- After failure: change arborescence (e.g. in circular fashion)
- Incoming port defines current arborescence
- After k-1 failures: At least one arborescence intact

J. Edmonds, **Edge-disjoint branchings**. Combinatorial Algorithms, 1972.

Ideal Resilience in General k-Connected Graphs

- Use directed trees (i.e. *arborescences*) instead of Hamilton cycles
  - *Arc-disjoint*, spanning, and *rooted* at destination
- Classic result: k-connectivity guarantees k-arborescence decomposition



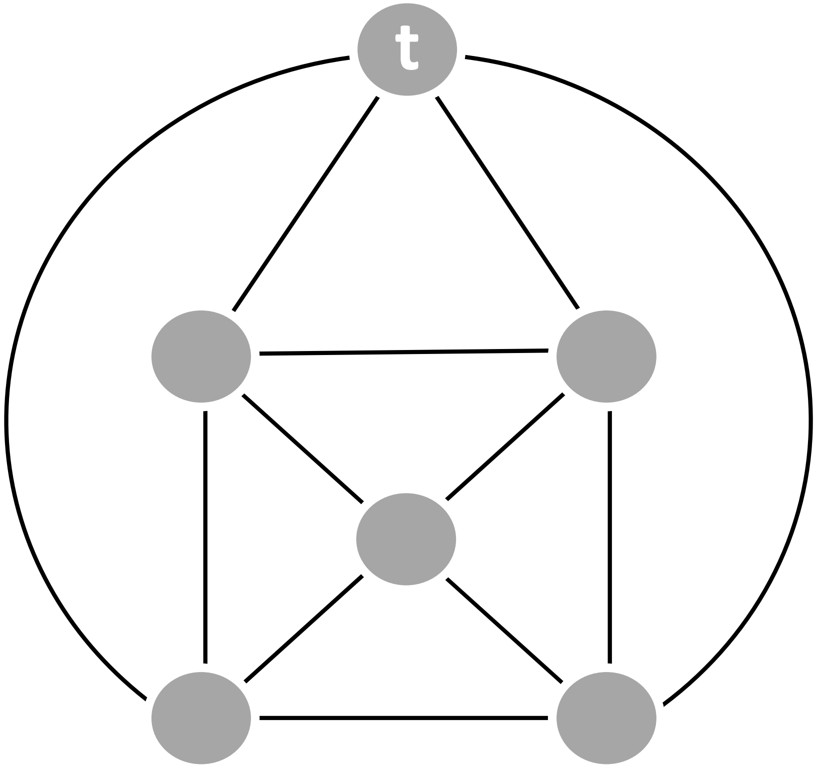
4-connected,
4 arborescences

Basic idea:
- Idea: route towards root on one arborescence
- After failure: change arborescence (e.g. in circular fashion)
- Incoming port defines current arborescence
- After k-1 failures: At least one arborescence intact

The challenge: how
to avoid earlier tree?

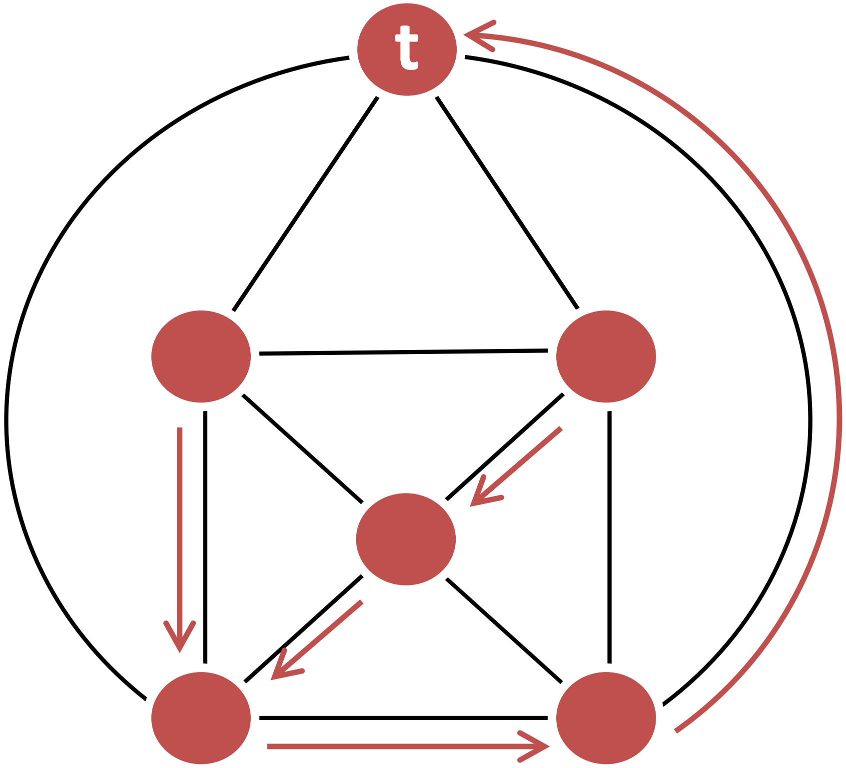
J. Edmonds, **Edge-disjoint branchings**. Combinatorial Algorithms, 1972.

A k-connected network contains k arc-disjoint spanning arborescences [Edmonds, 1972]



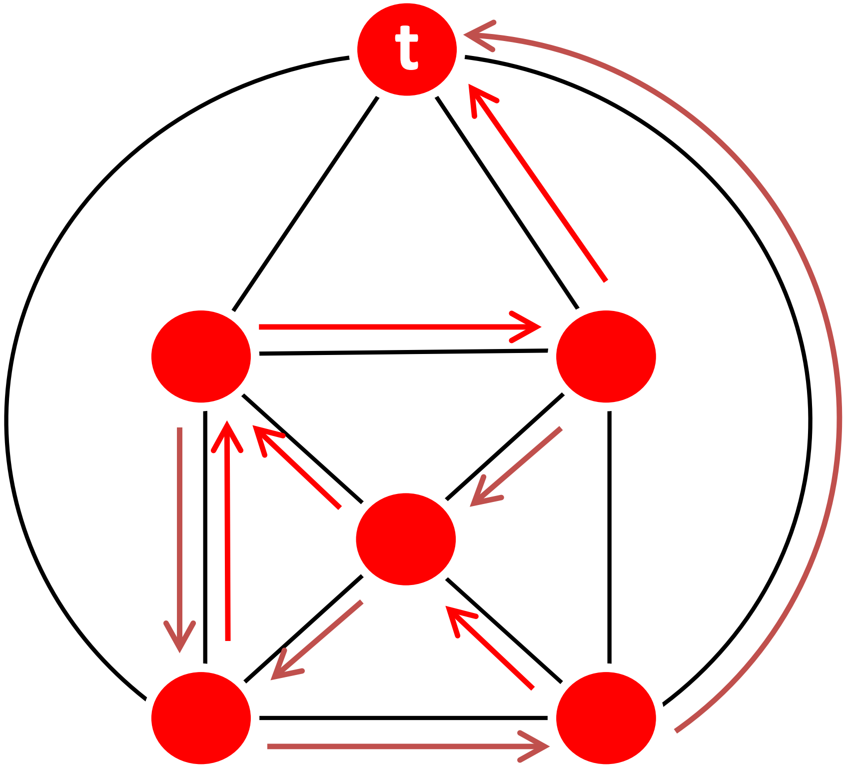
Credits: Marco Chiesa

A k-connected network contains k arc-disjoint spanning arborescences [Edmonds, 1972]



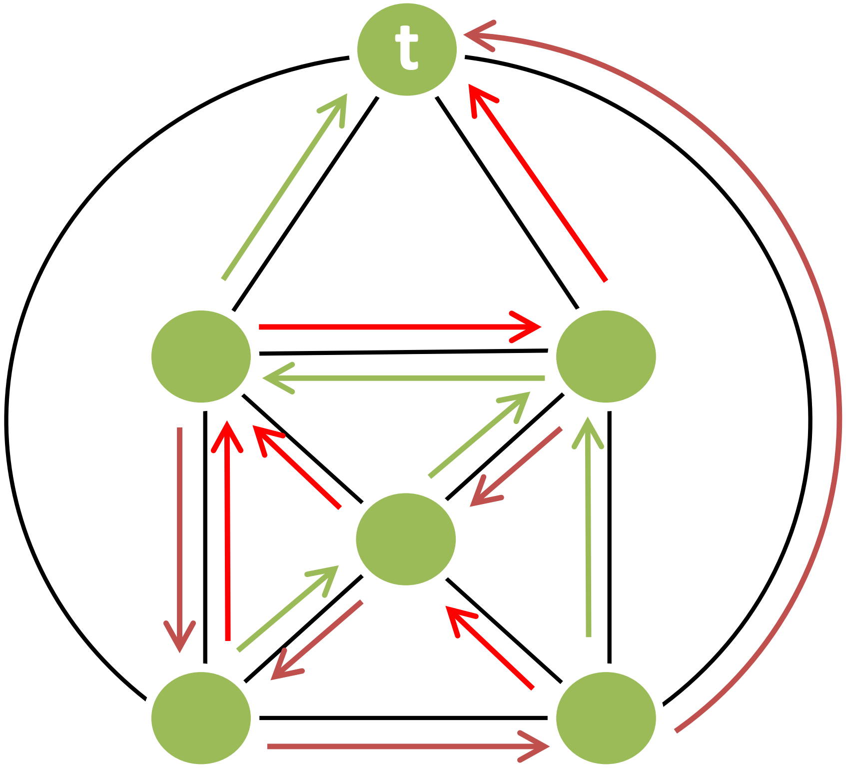
Credits: Marco Chiesa

A k-connected network contains k arc-disjoint spanning arborescences [Edmonds, 1972]



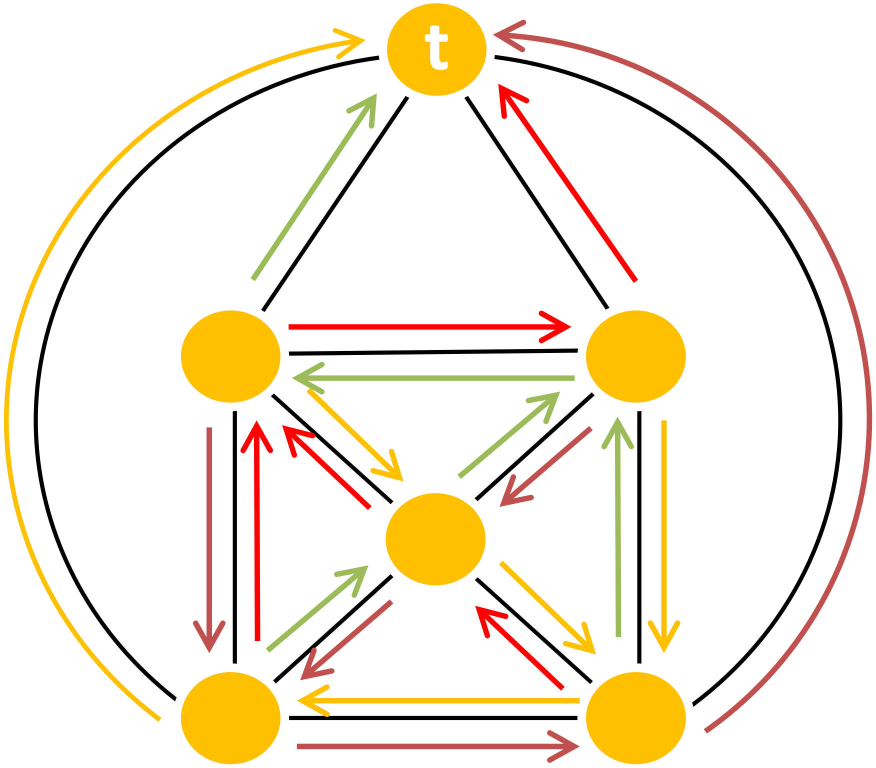
Credits: Marco Chiesa

A k-connected network contains k arc-disjoint spanning arborescences [Edmonds, 1972]



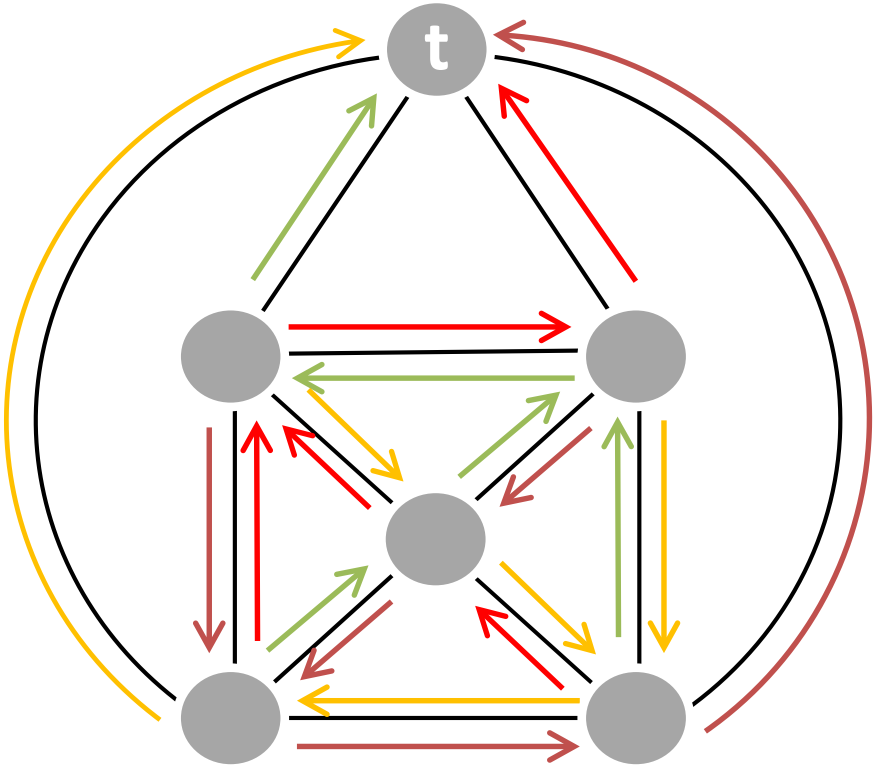
Credits: Marco Chiesa

A k-connected network contains k arc-disjoint spanning arborescences [Edmonds, 1972]



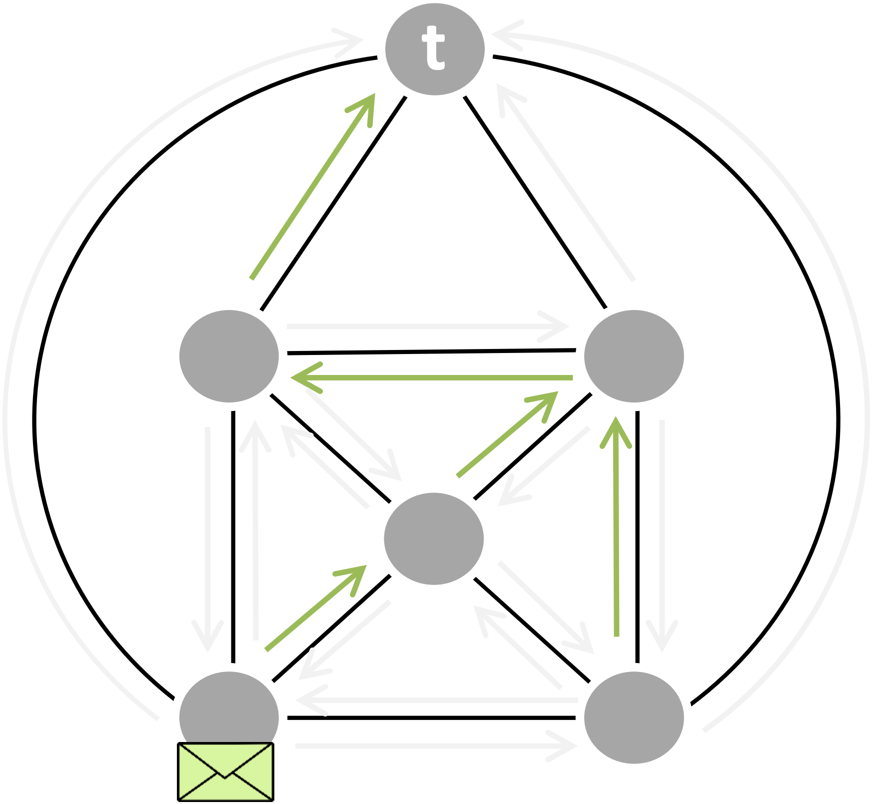
Credits: Marco Chiesa

A k-connected network contains k arc-disjoint spanning arborescences [Edmonds, 1972]



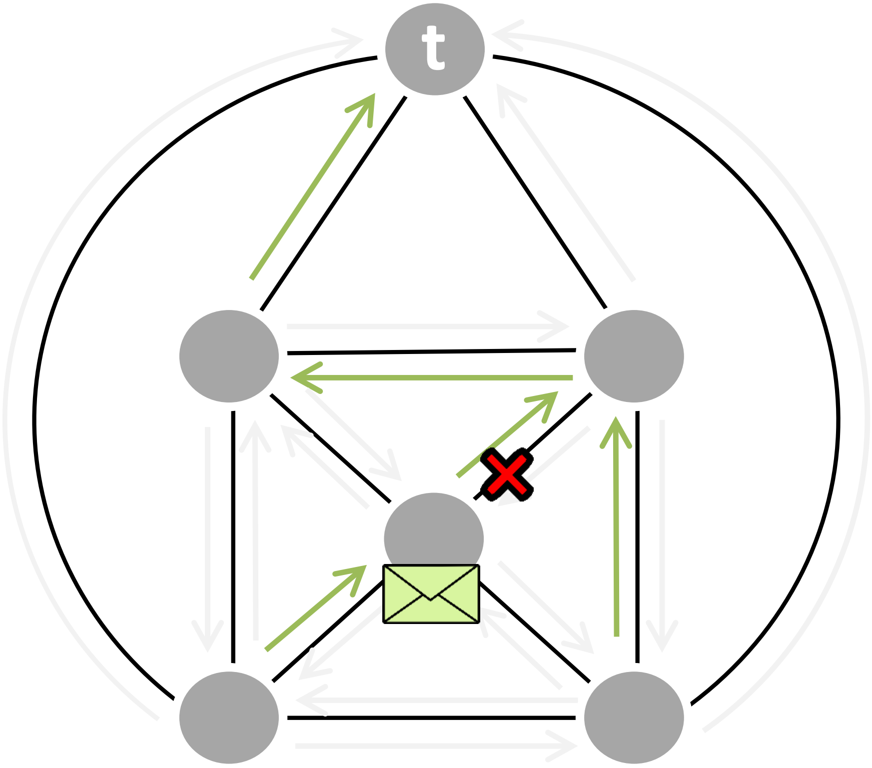
Credits: Marco Chiesa

General technique: routing along the same tree



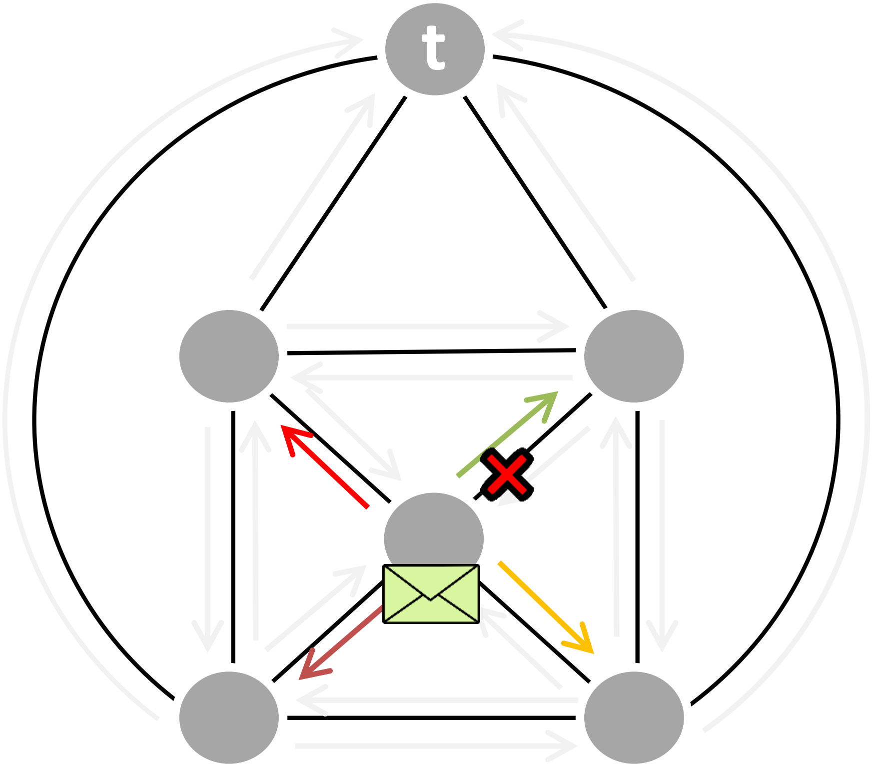
Credits: Marco Chiesa

When a failed link is hit…



Credits: Marco Chiesa

… how do we choose the next arborescence?



Credits: Marco Chiesa

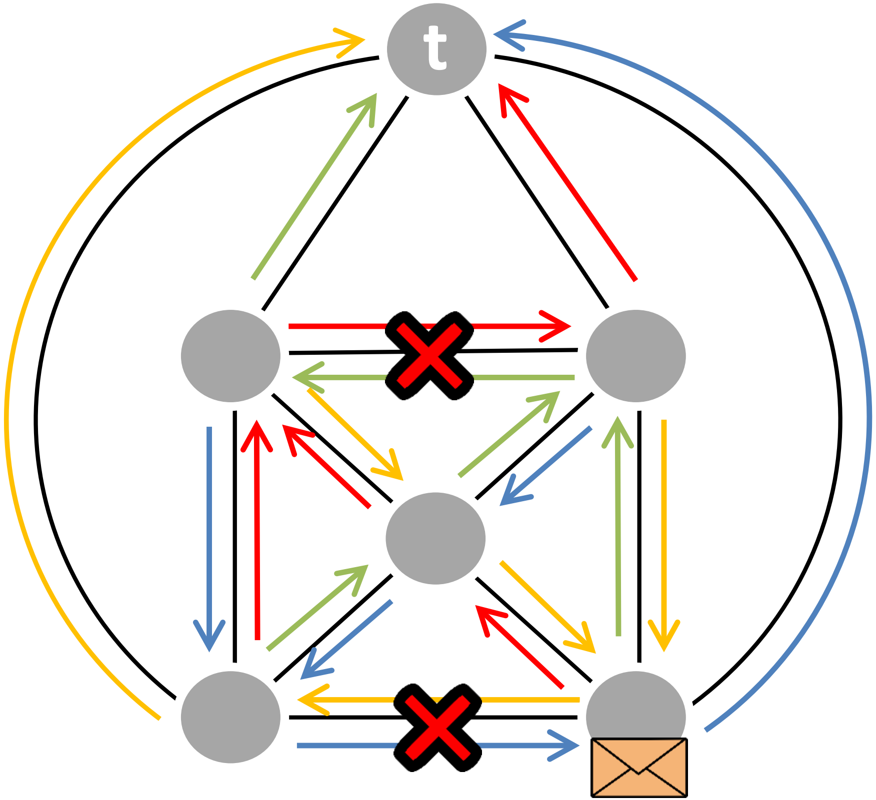
But how do we choose the next arborescence?

Circular-arborescence routing:

- compute an order of the arborescences
- switch to the next arborescence when hitting a failed link

Circular arborescence-routing is (k/2-1)-resilient

Arborescence order



Intuition: each single failure may affect two arborescences

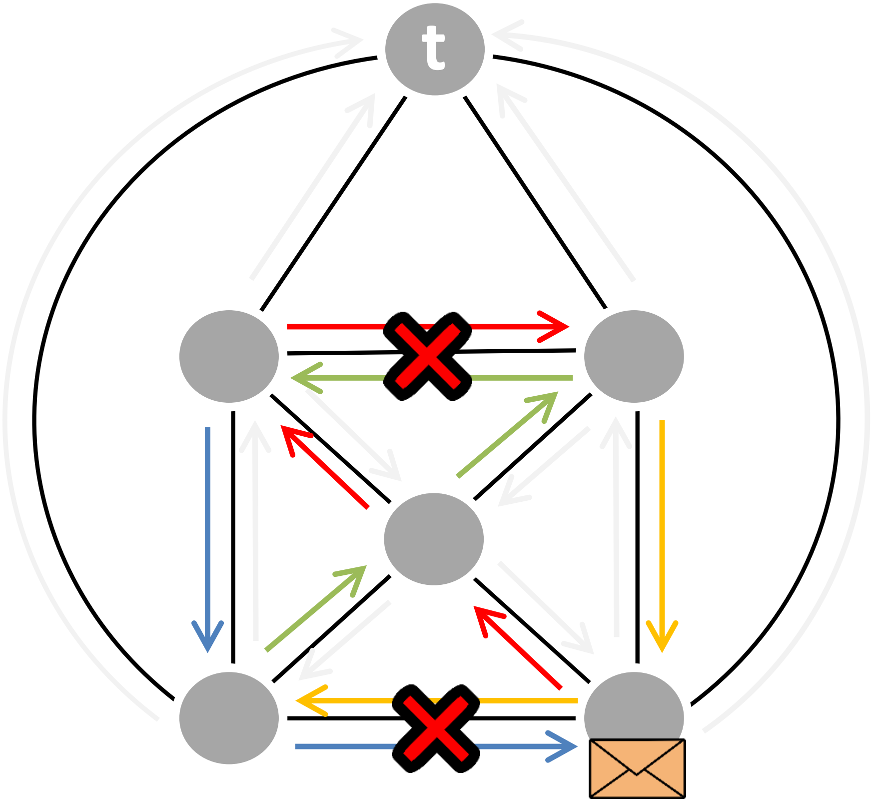
Credits: Marco Chiesa

Circular arborescence-routing is (k/2-1)-resilient

Arborescence order

1	2	3	4

Go along arborescence 1 to destination...



Intuition: each single failure may affect two arborescences

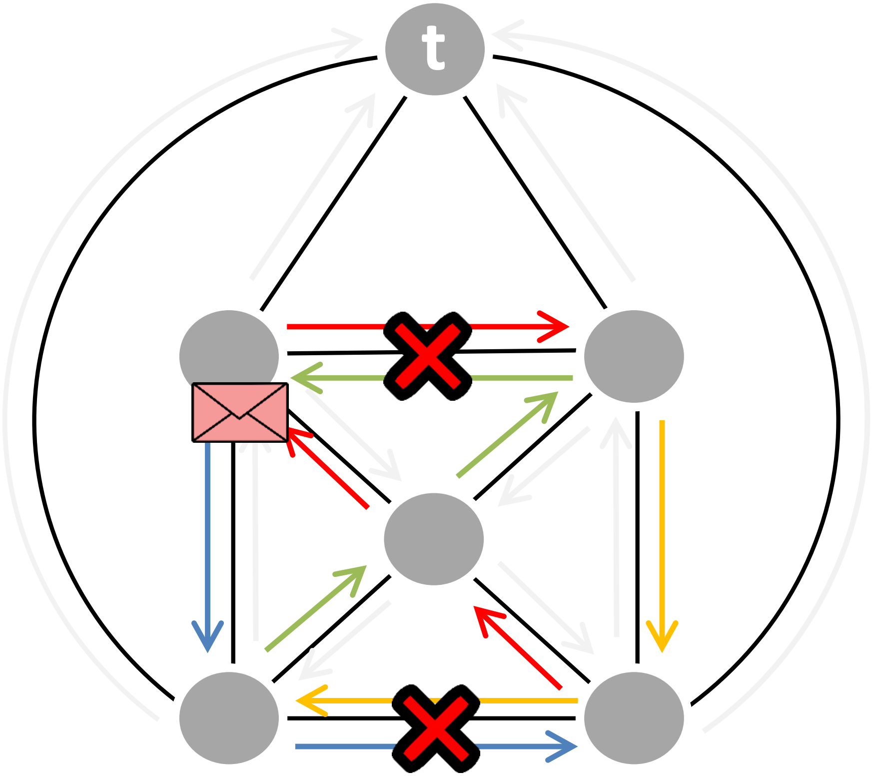
Credits: Marco Chiesa

Circular arborescence-routing is (k/2-1)-resilient

Arborescence order

1	2	3	4

Go along arborescence 2 to destination...



Intuition: each single failure may affect two arborescences

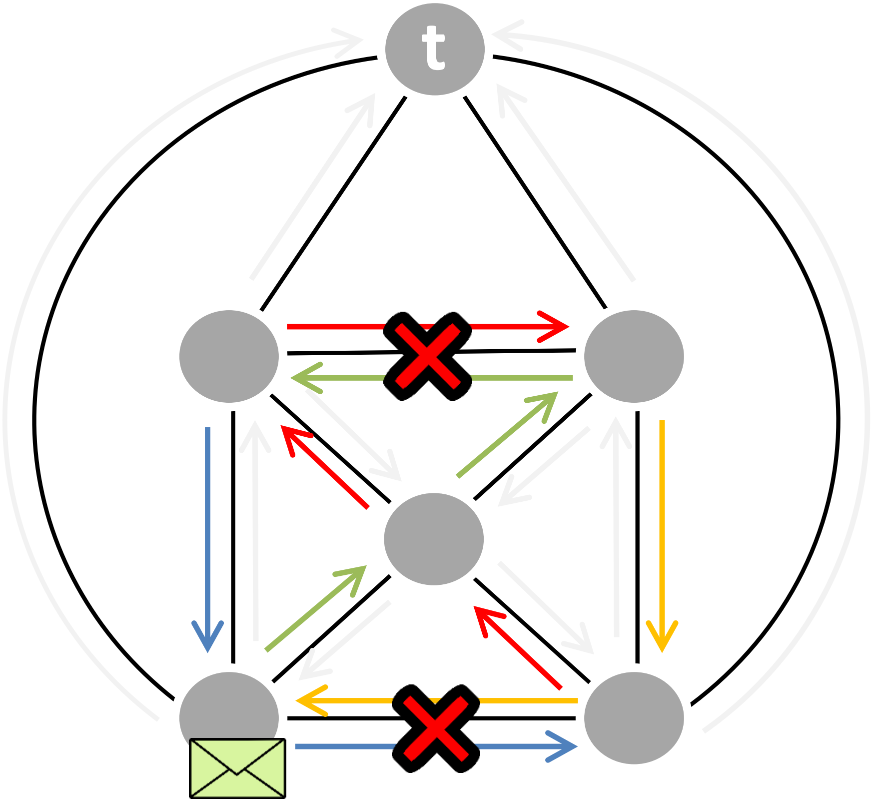
Credits: Marco Chiesa

Circular arborescence-routing is (k/2-1)-resilient

Arborescence order

1	2	3	4

Go along arborescence 3 to destination...



Intuition: each single failure may affect two arborescences

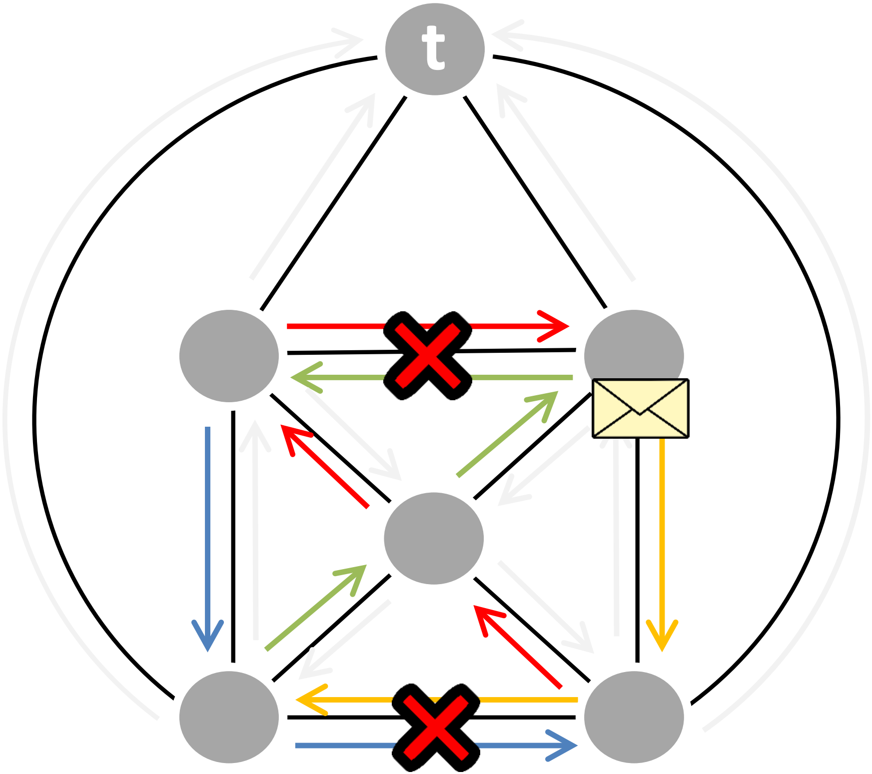
Credits: Marco Chiesa

Circular arborescence-routing is (k/2-1)-resilient

Arborescence order

1	2	3	4

Go along arborescence 4 to destination...

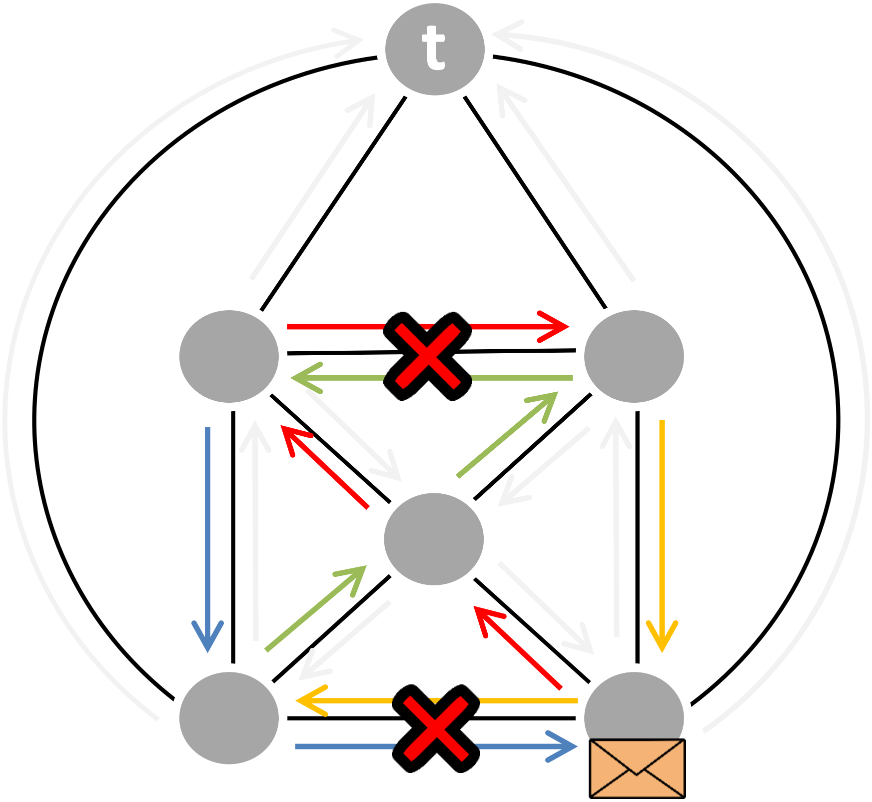


Intuition: each single failure may affect two arborescences

Credits: Marco Chiesa

Circular arborescence-routing is (k/2-1)-resilient

Arborescence order



Intuition: each single failure may affect two arborescences

All k=4 arborescences used (2 failures disconnected affected all four): LOOP!

Credits: Marco Chiesa

An Alternative Algorithm: Bouncing Arborescence

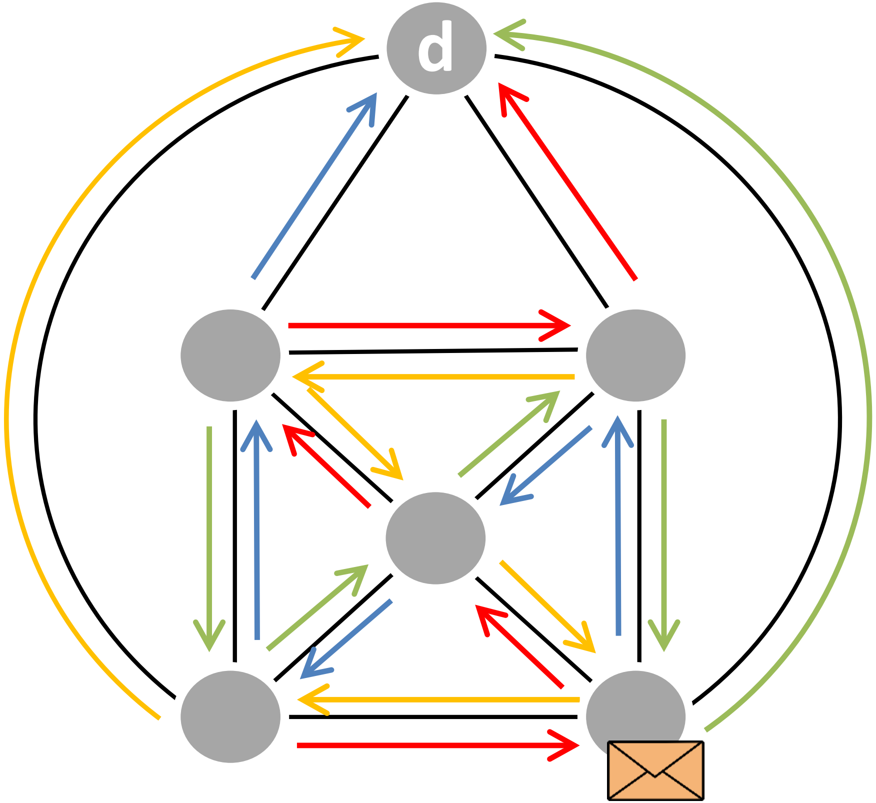
Bouncing-arborescence algorithm:

- Reroute on the tree that shares the failed link

This algorithm is ***1-resilient***.

Bouncing-Arborescence is 1-Resilient

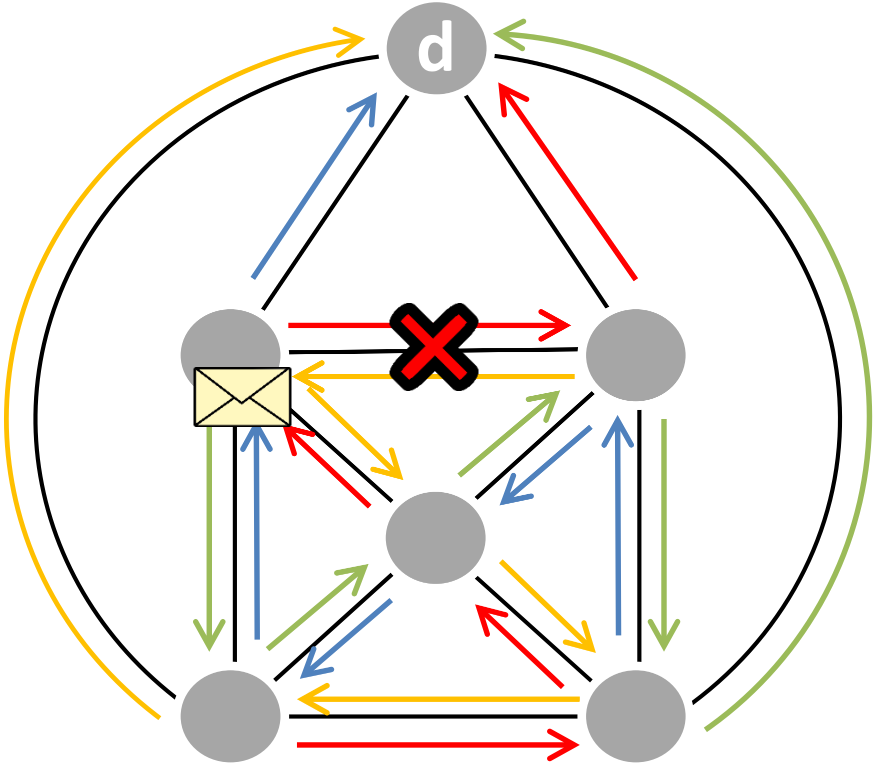
Start with red…



Credits: Marco Chiesa

Bouncing-Arborescence is 1-Resilient

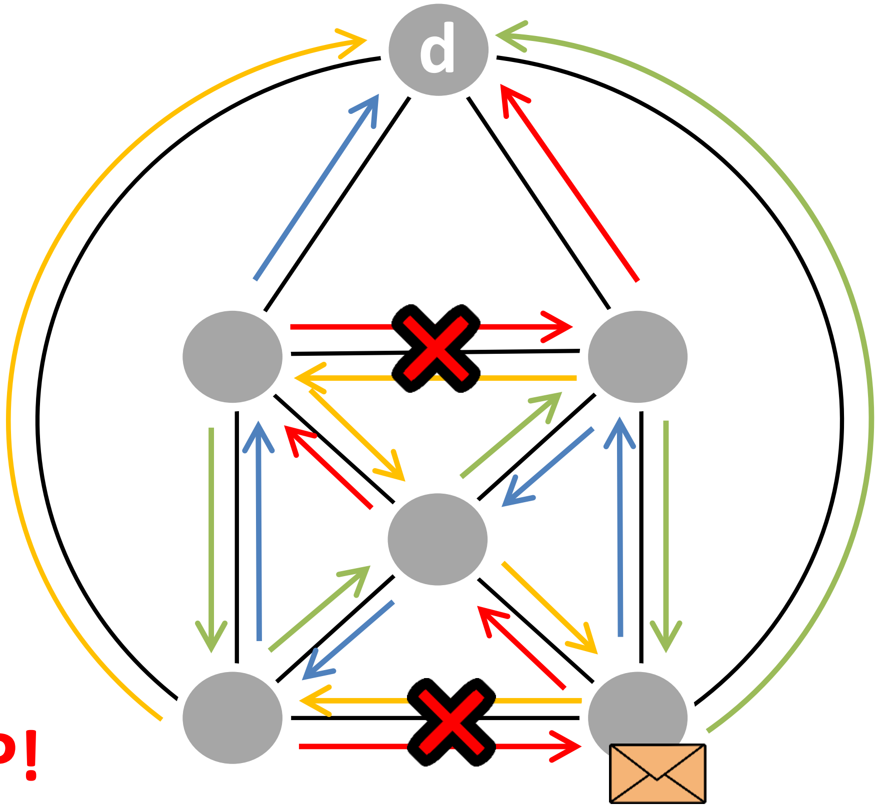
… bounce to yellow…



Credits: Marco Chiesa

Bouncing-Arborescence is 1-Resilient

… bounce to red (again!)…

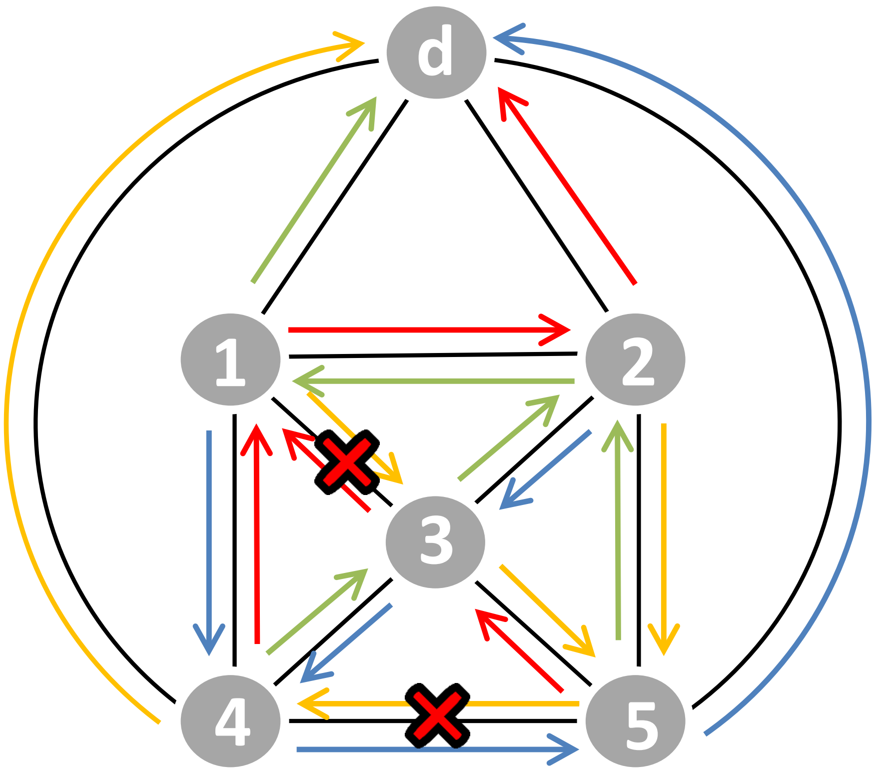


LOOP!

Credits: Marco Chiesa

Idea: Bounce on „Good Arborescences“

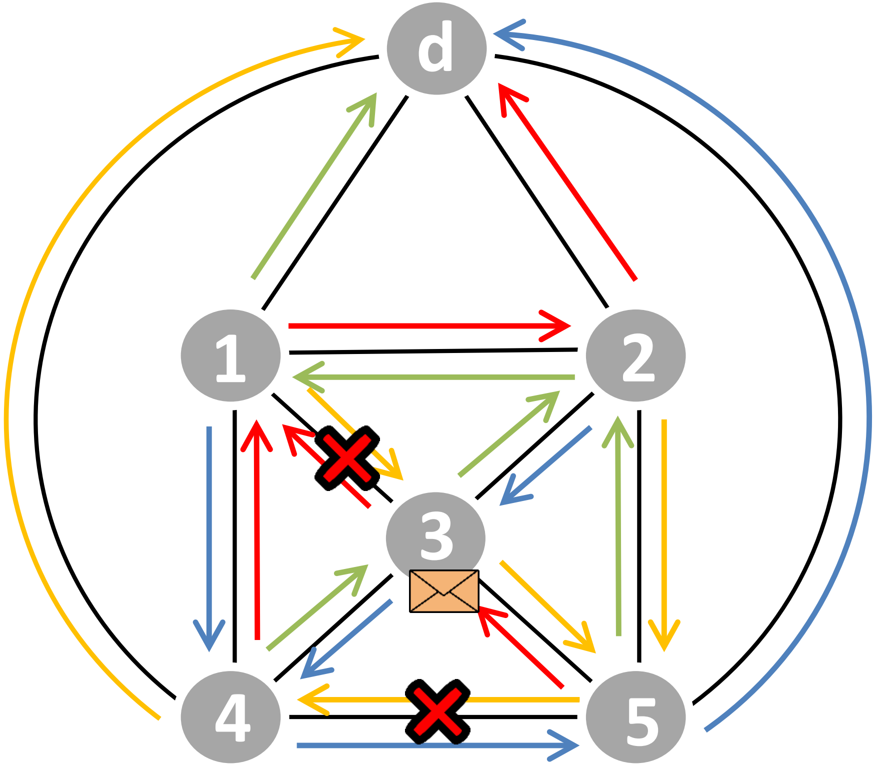
- Define **well-bouncing arc**:
  - When bounce get to the destination
  - Without hitting any other failures



Credits: Marco Chiesa

Idea: Bounce on „Good Arborescences“

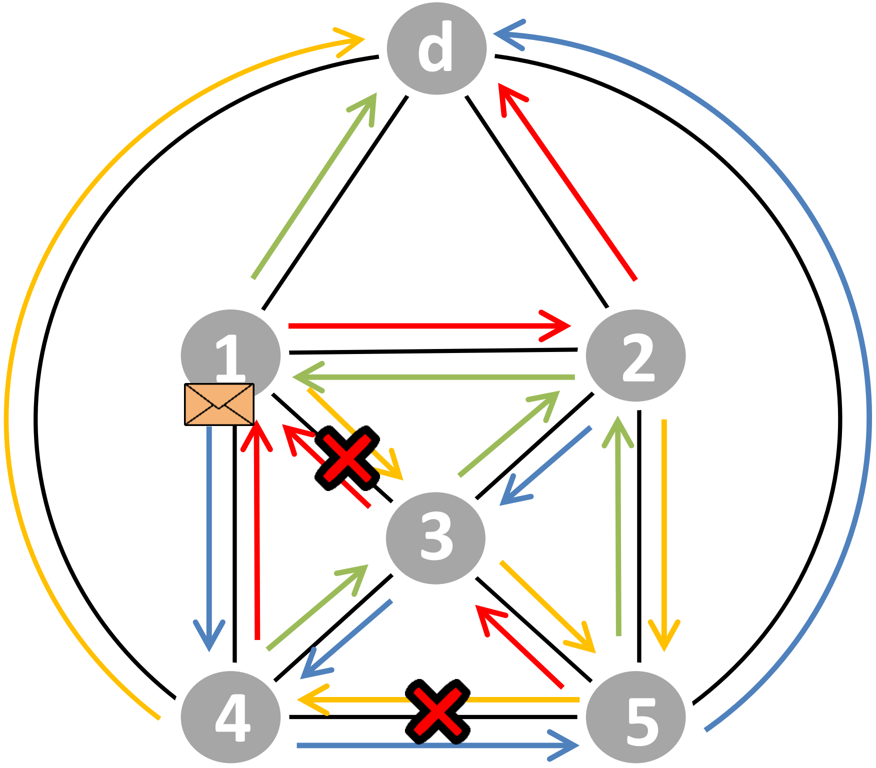
- Define **well-bouncing arc**:
  - When bounce get to the destination
  - Without hitting any other failures
  - (3,1) is not well-bouncing



Credits: Marco Chiesa

Idea: Bounce on „Good Arborescences“

- Define **well-bouncing arc**:
  - When bounce get to the destination
  - Without hitting any other failures
  - (3,1) is not well-bouncing
  - (1,3) is well-bouncing

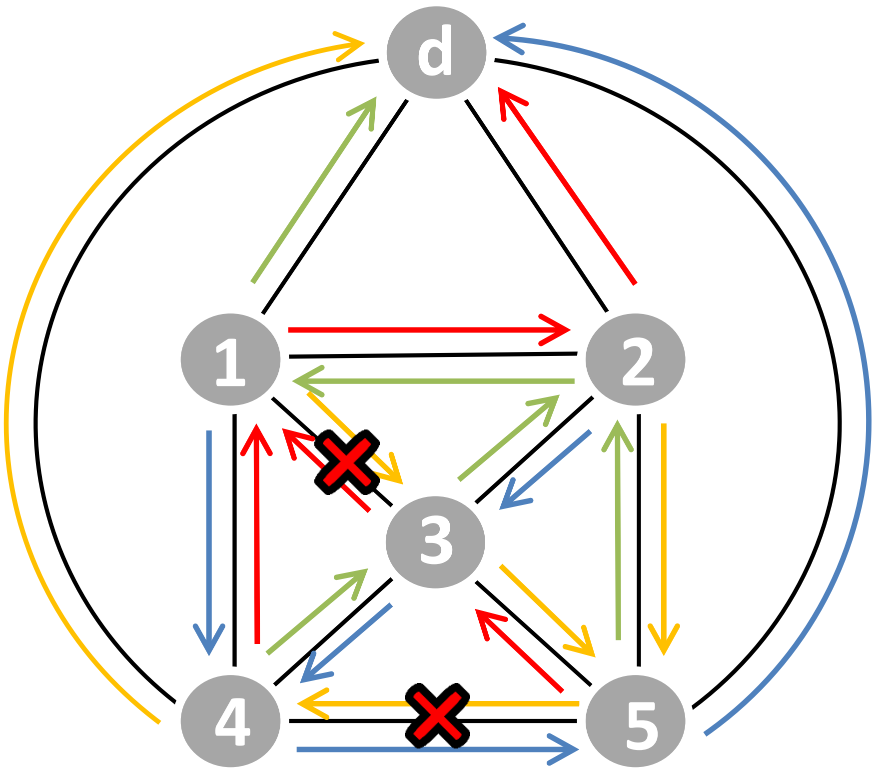


Credits: Marco Chiesa

Idea: Bounce on „Good Arborescences“

- Define **well-bouncing arc**:
  - When bounce get to the destination
  - Without hitting any other failures
  - (3,1) is not well-bouncing
  - (1,3) is well-bouncing

- Define **good arborescence**:
  - every failed arc is well-bouncing

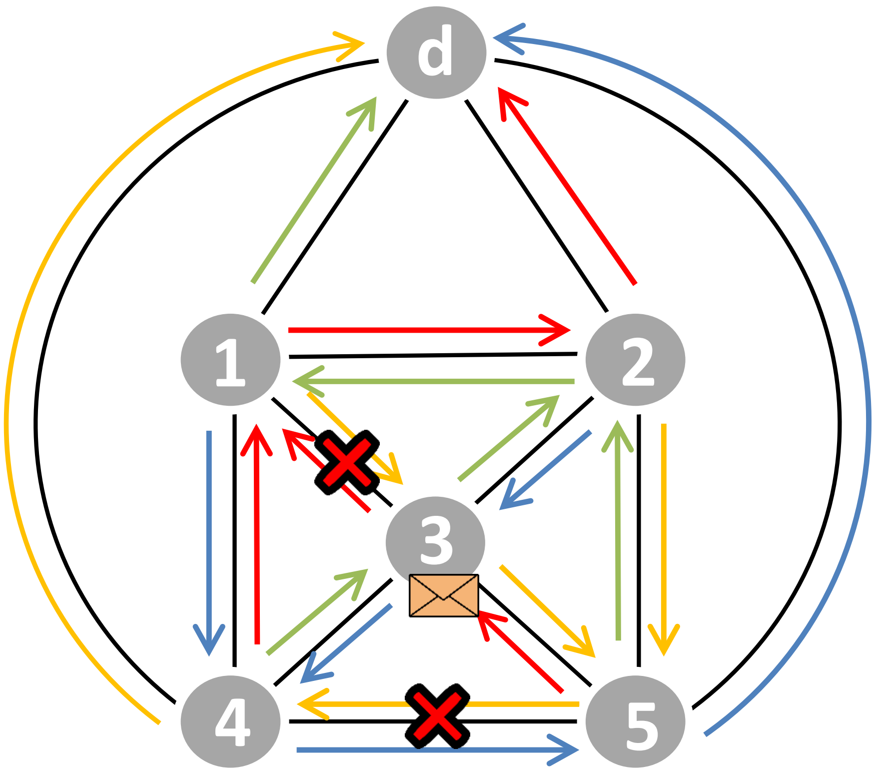


Credits: Marco Chiesa

Idea: Bounce on „Good Arborescences“

- Define **well-bouncing arc**:
  - When bounce get to the destination
  - Without hitting any other failures
  - (3,1) is not well-bouncing
  - (1,3) is well-bouncing

- Define **good arborescence**:
  - every failed arc is well-bouncing
  - Red is not a good arborescence

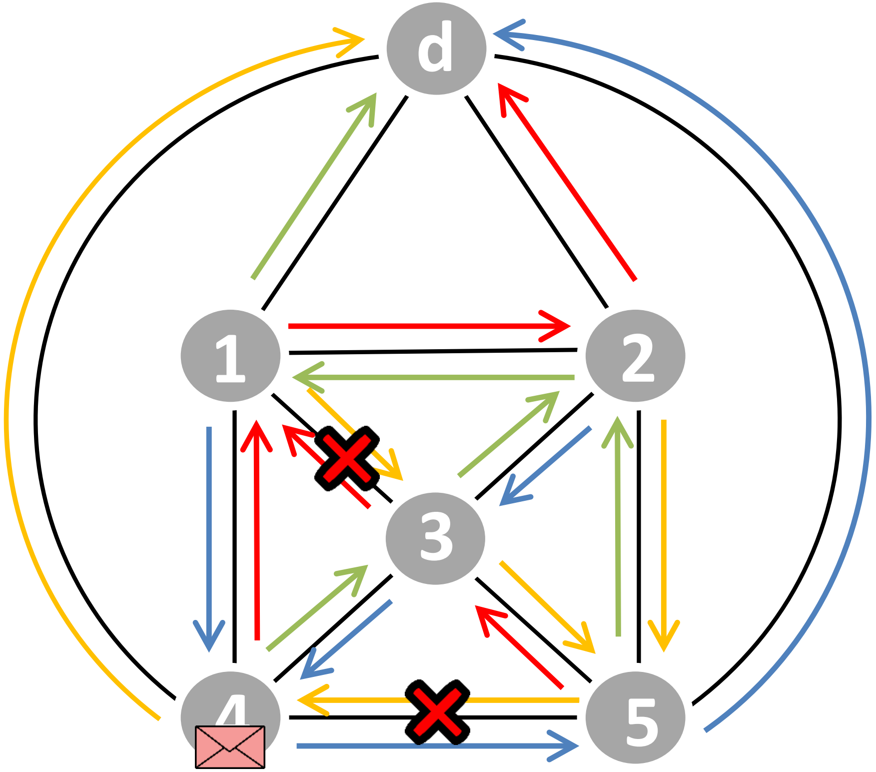


Credits: Marco Chiesa

Idea: Bounce on „Good Arborescences“

- Define **well-bouncing arc**:
  - When bounce get to the destination
  - Without hitting any other failures
  - (3,1) is not well-bouncing
  - (1,3) is well-bouncing

- Define **good arborescence**:
  - every failed arc is well-bouncing
  - Red is not a good arborescence
  - Blue is a good arboresence



Credits: Marco Chiesa

Ideas

- One can show that there is always a good arborescence

- An tempting idea:
  - route on an arborescence X until a failed link is hit:
    - if X is a good arborescence, bounce!
    - otherwise, route circular

- Too good to be true:
  - The “goodness” of an arborescence depends on the actual set of failed links!
  - How do we know a arborescence is good?

Credits: Marco Chiesa

Resilience Criteria

Ideal resilience

Given a *k*-connected graphs, we can tolerate *any k-1 link failures.*

Perfect resilience

Any source *s* can always reach any destination *t* as long as the unterlying network is *physically connected*.

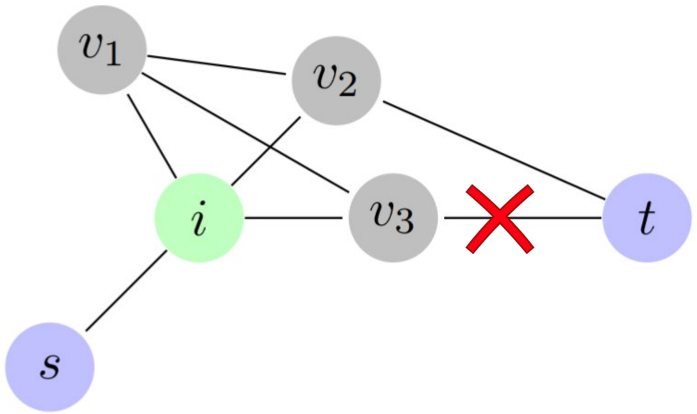
Can this be achieved? Assume undirected link failures.

Resilience Criteria

Perfect resilience is impossible to achieve in general.

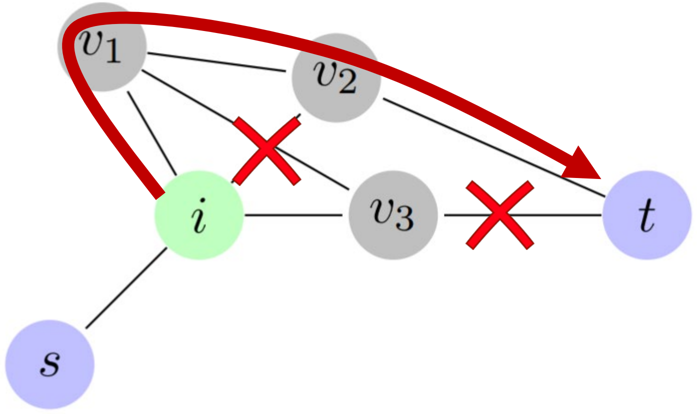
Relevant Neighbors

- Routing table of node i: matches in-ports of i to out-ports of i
  - ... depending on the incident failures
- But not all neighbors are **relevant**: only if potentially required to reach destination!
  - *Without local failures*: just v_2, v_3 for *i*, since v_1 does not give extra connectivity



Relevant Neighbors

- Routing table of node i: matches in-ports of i to out-ports of i
  - ... depending on the incident failures
- But not all neighbors are **relevant**: only if potentially required to reach destination!
  - *Without local failures*: just v_2, v_3 for *i*, since v_1 does not give extra connectivity
  - *With additional failures* v_1 becomes relevant, since v_1 might be only choice to reach destination t
    - Note: v_1 is unaware of these non-incident failures!

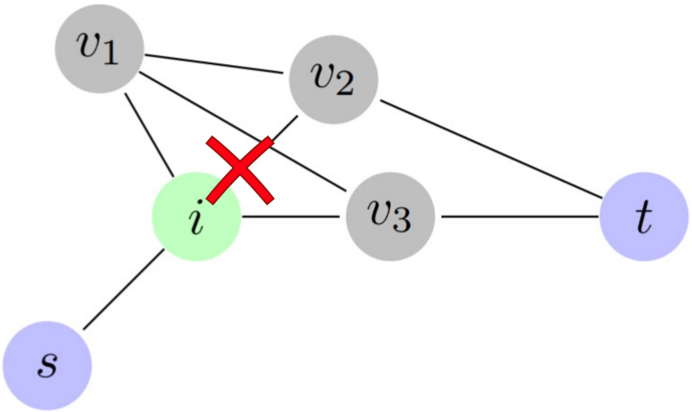


High-level definition of ***relevant***: From the local view-point of the node i, a relevant neighbor might be only neighbor to reach destination (without taking a detour over a current neighbor).

How to Achieve Perfect Resilience?

- Necessary: need to *try all relevant* neighbors
  - Here, if local link to v_2 broken: v_1 and v_3

- That is, if packet
  - comes from v_3: eventually try v_1
  - comes from v_1: eventually try v_3



Impossibility: On Planar Graphs

Some observations:

- Additional failures only ***add relevant neighbors*** to nodes
- Any node of ***degree 2*** of G after failures must forward packets with incoming port p to port p'
- If all neighbors are relevant, the forwarding function of a node must be a ***cyclic permutation***

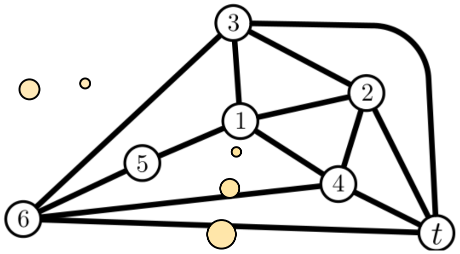
Impossibility: On Planar Graphs

Some observations:

- Additional failures only ***add relevant neighbors*** to nodes
- Any node of ***degree 2*** of G after failures must forward packets with incoming port p to port p'
- If all neighbors are relevant, the forwarding function of a node must be a ***cyclic permutation***

Idea of the counter example:

All neighbors of all nodes are relevant (even without failures).



Considered node 1 will not see any local failures.

So we must fix a permutation for node 1.

Impossibility: On Planar Graphs

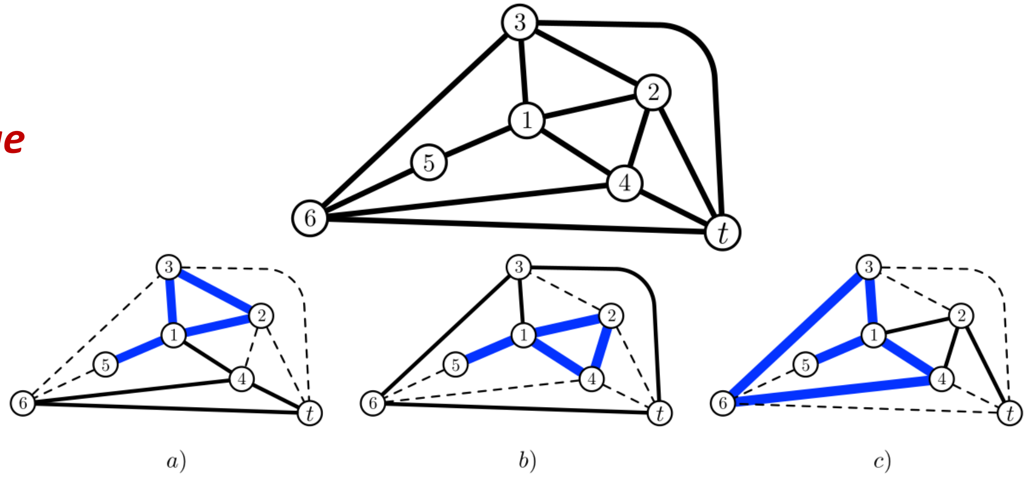
Some observations:

- Additional failures only ***add relevant neighbors*** to nodes
- Any node of ***degree 2*** of G after failures must forward packets with incoming port p to port p'
- If all neighbors are relevant, the forwarding function of a node must be a ***cyclic permutation***

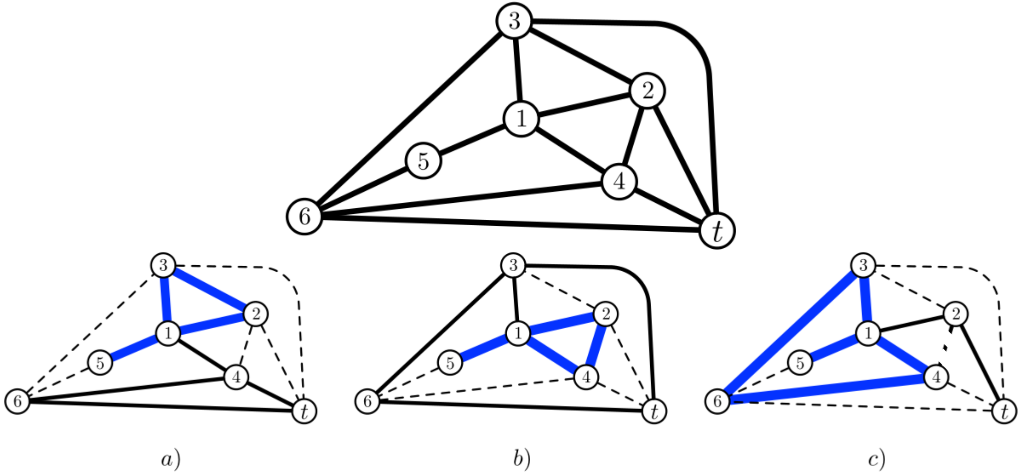
Proof idea, with three cases:

- If the ***dashed*** links fail (***non-local*** to node 1), in any forwarding pattern, packets will be stuck in one of the ***blue loops***…
- … even though there is at least one ***remaining path*** to the target

Go through all possible permutations @1 and give counter example.



Impossibility: On Planar Graphs

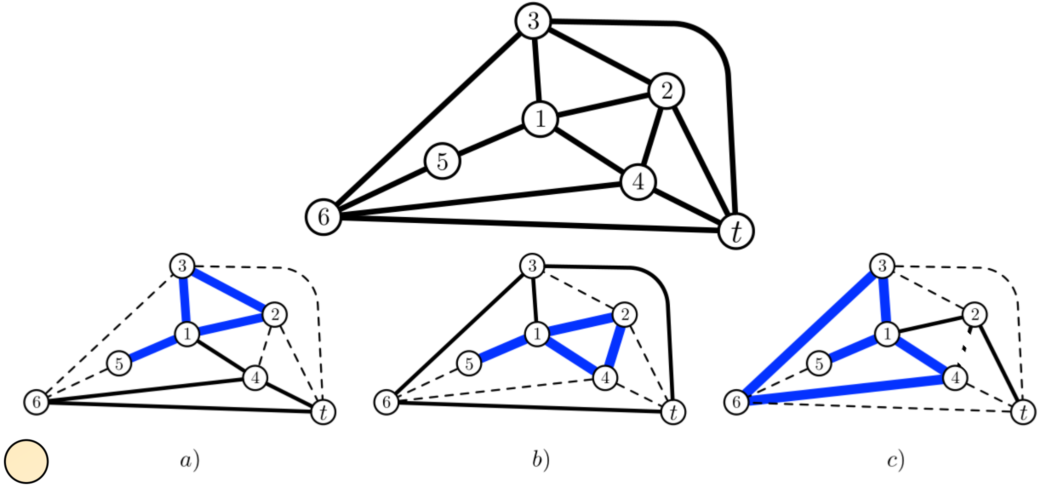


Arriving on inport 5, forwarded to 2.

For node 1:
5->2 implies
(5,2,3,4) (b)
(5,2,4,3) (a)

Possible cyclic permutations: when a packet arrives from 2, due to cyclic permutation, it can only be forwarded to either 3 or 4. Leads to *loops* in scenarios (b) (4 goes to 5, 2 can only go to 4) and (a) (3 goes to 5, 2 can only go to 3), respectively.

Impossibility: On Planar Graphs



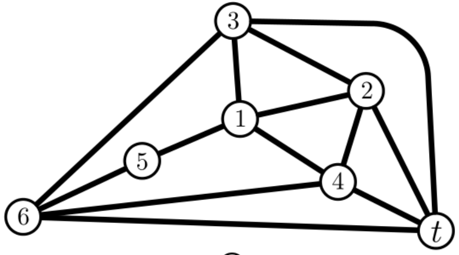
Arriving on inport 5, forwarded to 3.

For node 1:
5->2 implies
(5,2,3,4) (b)
(5,2,4,3) (a)

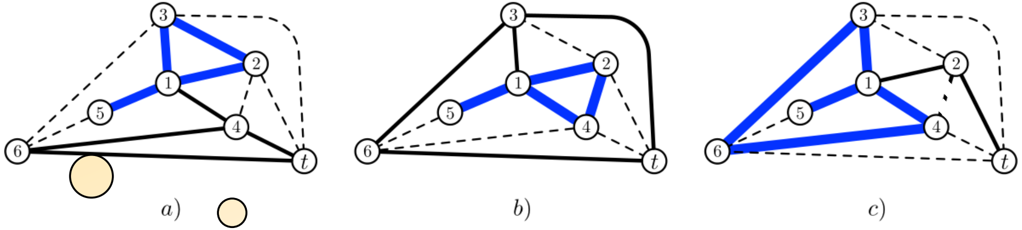
For node 1:
5->3 implies
(5,3,4,2) (a)
(5,3,2,4) (c)

Possible cyclic permutations: when a packet then arrives on port 4, it can only be forwarded to either 2 or 5. Leads to *loops* in scenarios (a) (2 will go to 5, 5 can only go to 1 and 3 only to 2) and (c) (5 goes to 3, 4 goes to 5, rest degree-2), respectively.

Impossibility: On Planar Graphs



Arriving on inport 5, forwarded to 4.

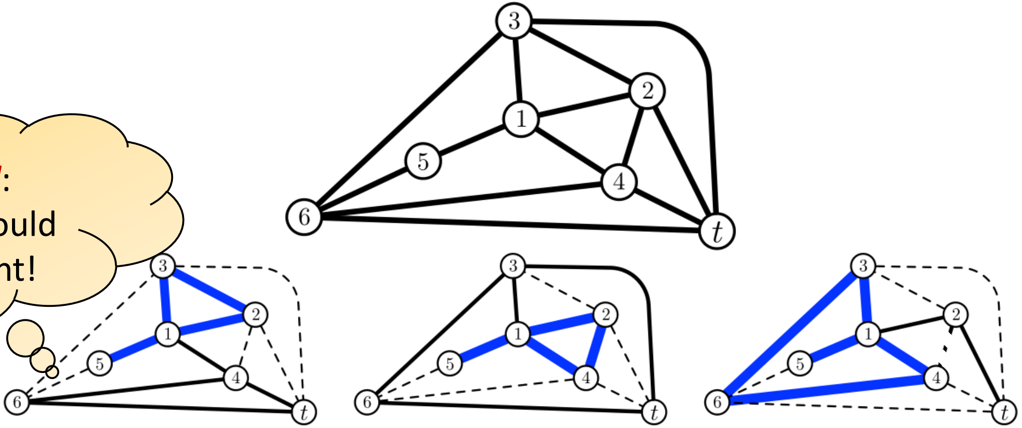


For node 1: 5->2 implies (5,2,3,4) (b) (5,2,4,3) (a)	For node 1: 5->3 implies (5,3,4,2) (a) (5,3,2,4) (c)	For node 1: 5->4 implies (5,4,2,3) (c) (5,4,3,2) (b)

Possible cyclic permutations: packet arriving on port 3 can only be forwarded to either 5 or 2. Leads to *loops* in scenarios (c) and (b), respectively.

Impossibility: On Planar Graphs

Link needed: otherwise 5 would not be relevant!



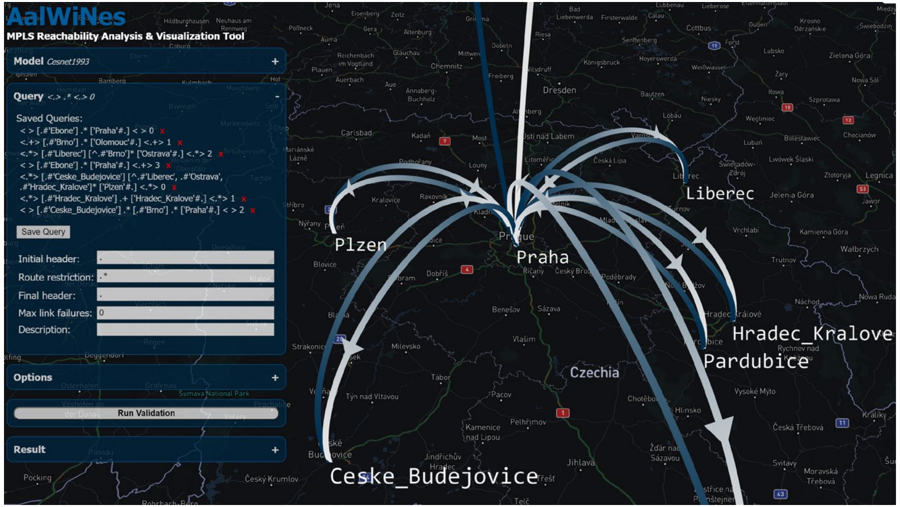
a) b) c)

For node 1: 5->2 implies (5,2,3,4) (b) (5,2,4,3) (a)	For node 1: 5->3 implies (5,3,4,2) (a) (5,3,2,4) (c)	For node 1: 5->4 implies (5,4,2,3) (c) (5,4,3,2) (b)

Possible cyclic permutations: packet arriving on port 3 can only be forwarded to either 5 or 2. Leads to *loops* in scenarios (c) and (b), respectively.

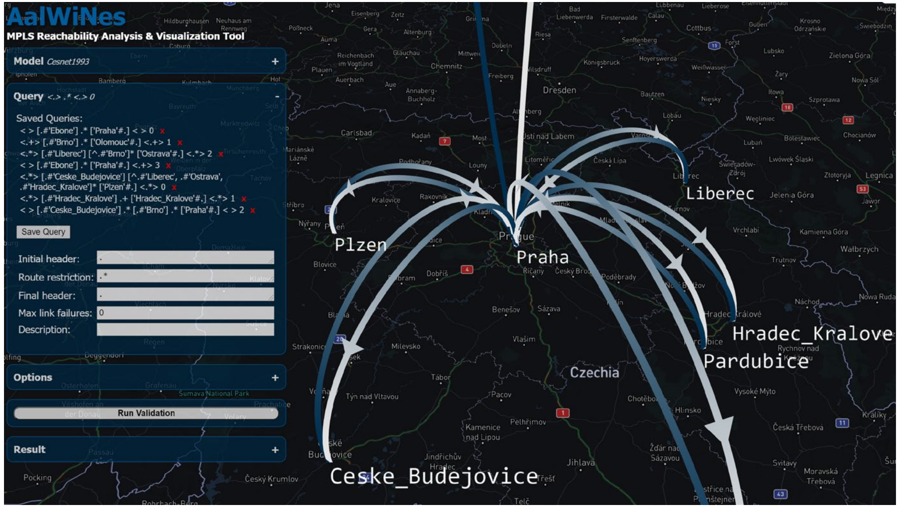
A Pity: Planar Graphs Are Important

- Internet Topology Zoo and Rocketfuel topologies
  - 88% of the graphs are ***planar***



A Pity: Planar Graphs Are Important

- Internet Topology Zoo and Rocketfuel topologies
  - 88% of the graphs are ***planar***
  - However:
    - Almost a third (32%) belong to the family of ***cactus*** graphs
    - Roughly half of the graphs (49%) are ***outerplanar***
    - … and they work ☺



Where Can Perfect Resilience Be Achieved?

For example on **outerplanar graphs**:

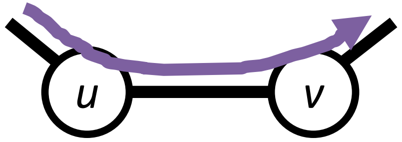
- Via *geometric routing*, well studied in sensor networks etc.
- Embed graph in the plane s.t. all nodes are on the outer face
  - Note: If a link l belongs to the outer face of a planar graph G, it also belongs to the outer face for all subgraphs of G
- Apply *right-hand rule* to forwarding (skipping failures)
  - Ensures packets use only the links of the outer face and do not change the direction despite failures
- Strategy traverses all nodes on the outer face

- Also works for any graph which is *outerplanar without the source* (e.g., K4)

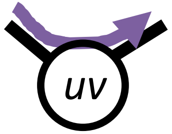
Some Observations

- K_5, K_3,3: *no perfect resilience*

- Perfect resiliency on graph G -> any *subgraph* G' of G also allows for perfect resiliency
  - Idea: Take routing on G, fail edges to create G', routing must still work



- Contraction works as well, by a simulation argument
  - A bit technical

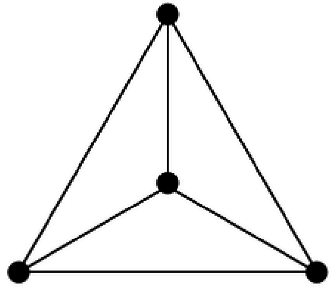


- Combined: Perfect resilience on graph G -> any minor G' of G as well
  - But since K_5, K_3,3 not: *non-planar graphs not perfectly resilient*

What we know about perfect resilience

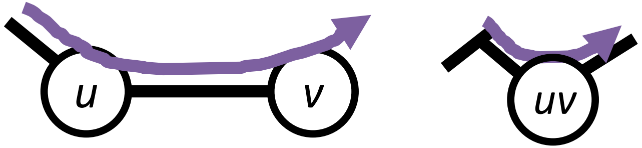
Possible:

- On all outerplanar graphs [right-hand rule]
- On every graph that is outerplanar without the destination (e.g. non-outerplanar planar K_4)



Impossible:

- On some planar graphs
- Every non-planar graph
- Perfect resilience must hold on minors



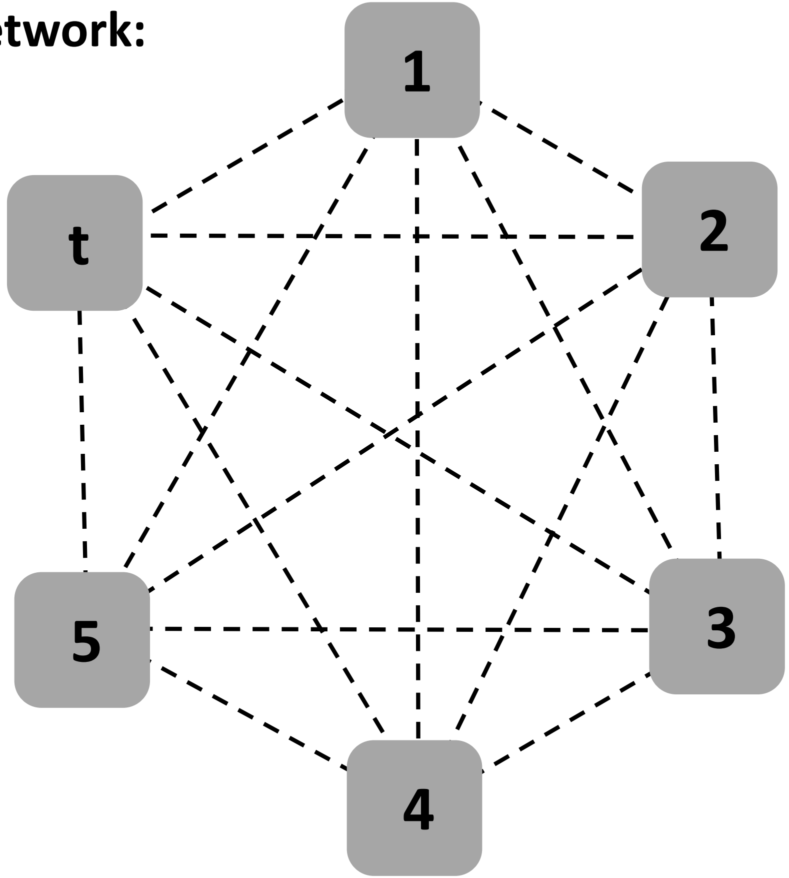
Foerster et al. **On the Feasibility of Perfect Resilience with Local Fast Failover.** SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS), 2021.

Roadmap

- A Brief History of Resilient Networking
- Algorithms for Local Fast Re-Routing (FRR)
- **Accounting for Congestion**
- Accounting for Network Policy

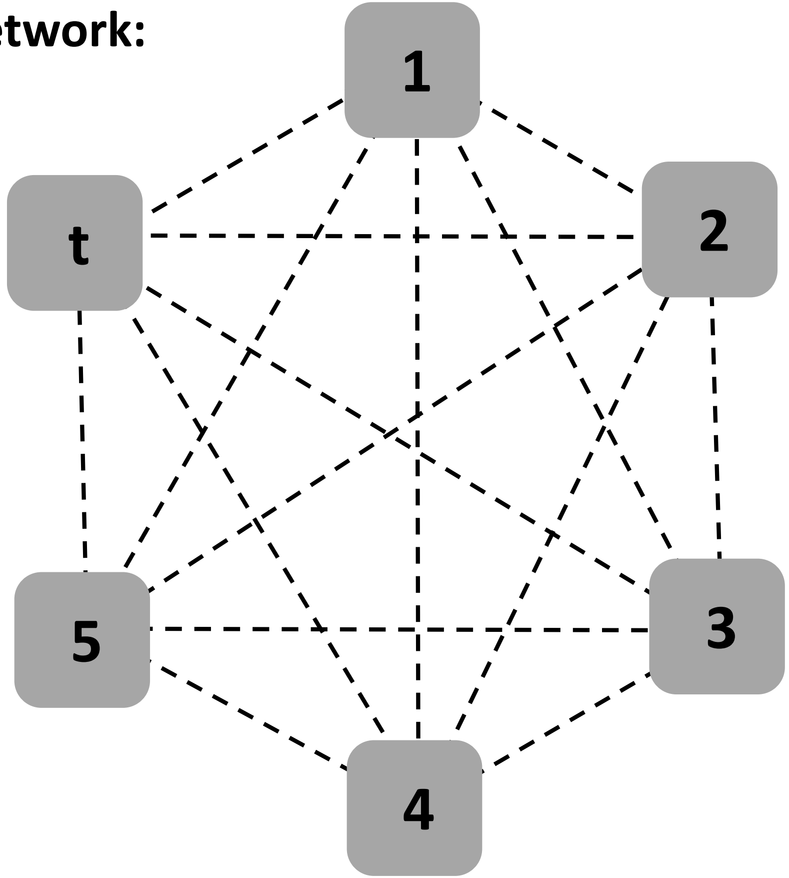
Congestion-Aware FRR

A most simple network: the clique



Congestion-Aware FRR

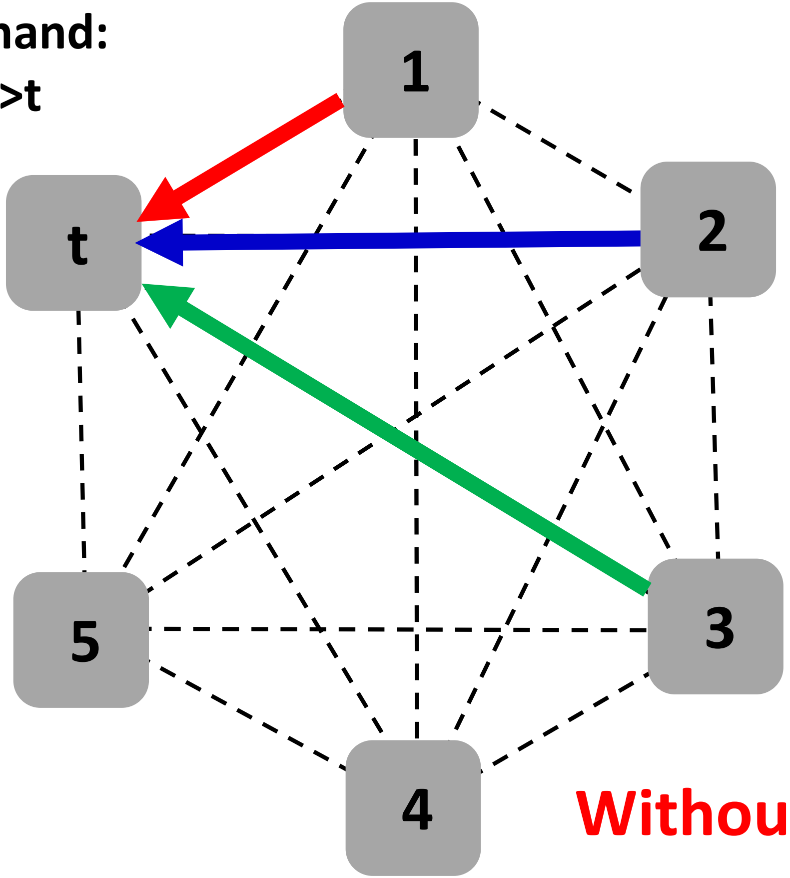
A most simple network: the clique



Assume we can match source.

Congestion-Aware FRR

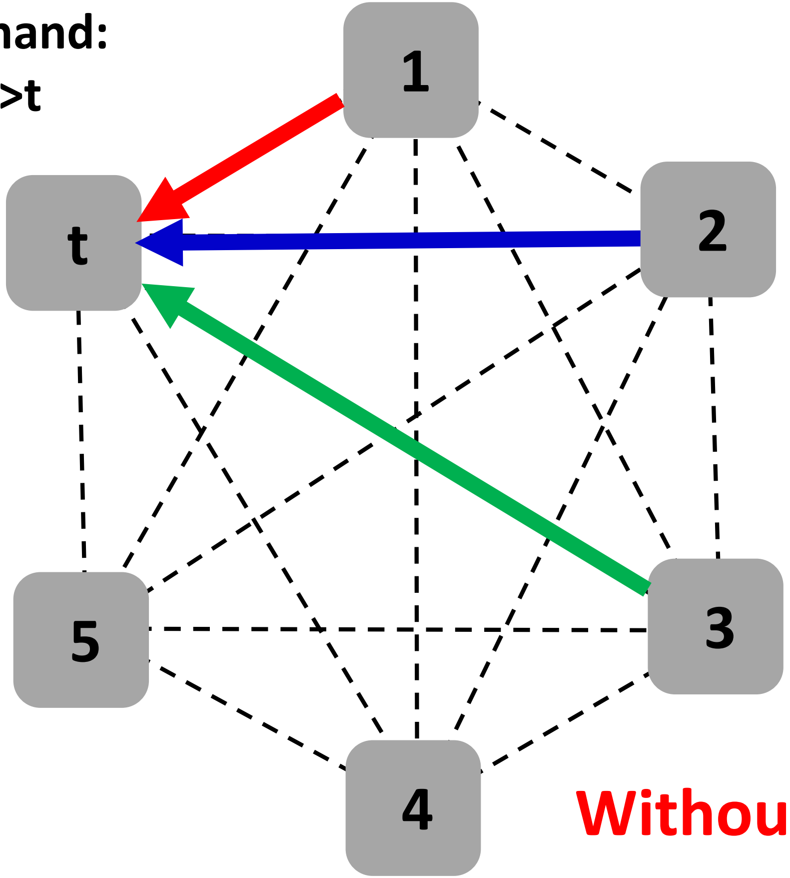
Traffic demand:
{1,2,3}->t



Without failures!

Congestion-Aware FRR

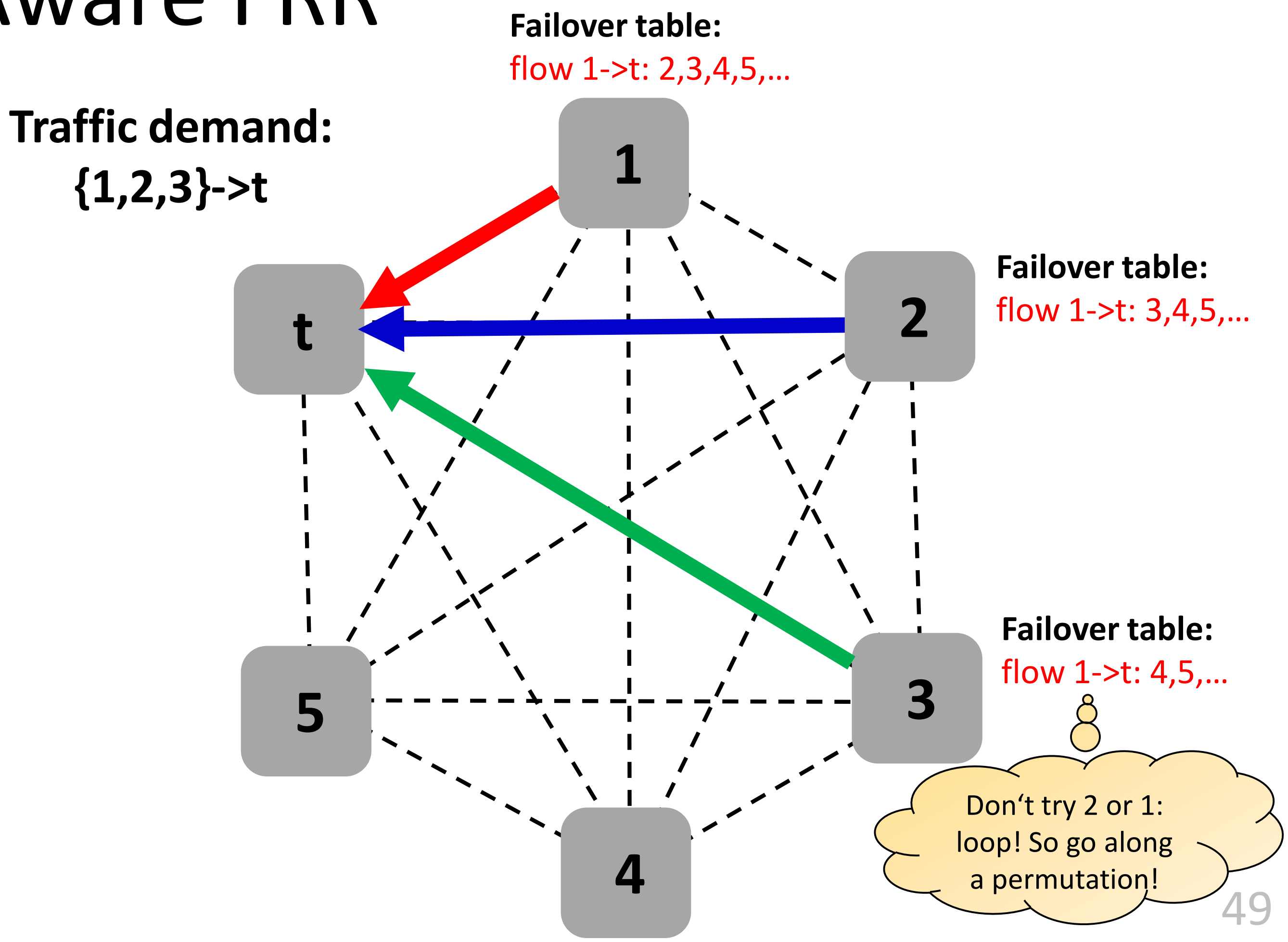
Traffic demand:
{1,2,3}->t



Assume single destination (incast scenario).

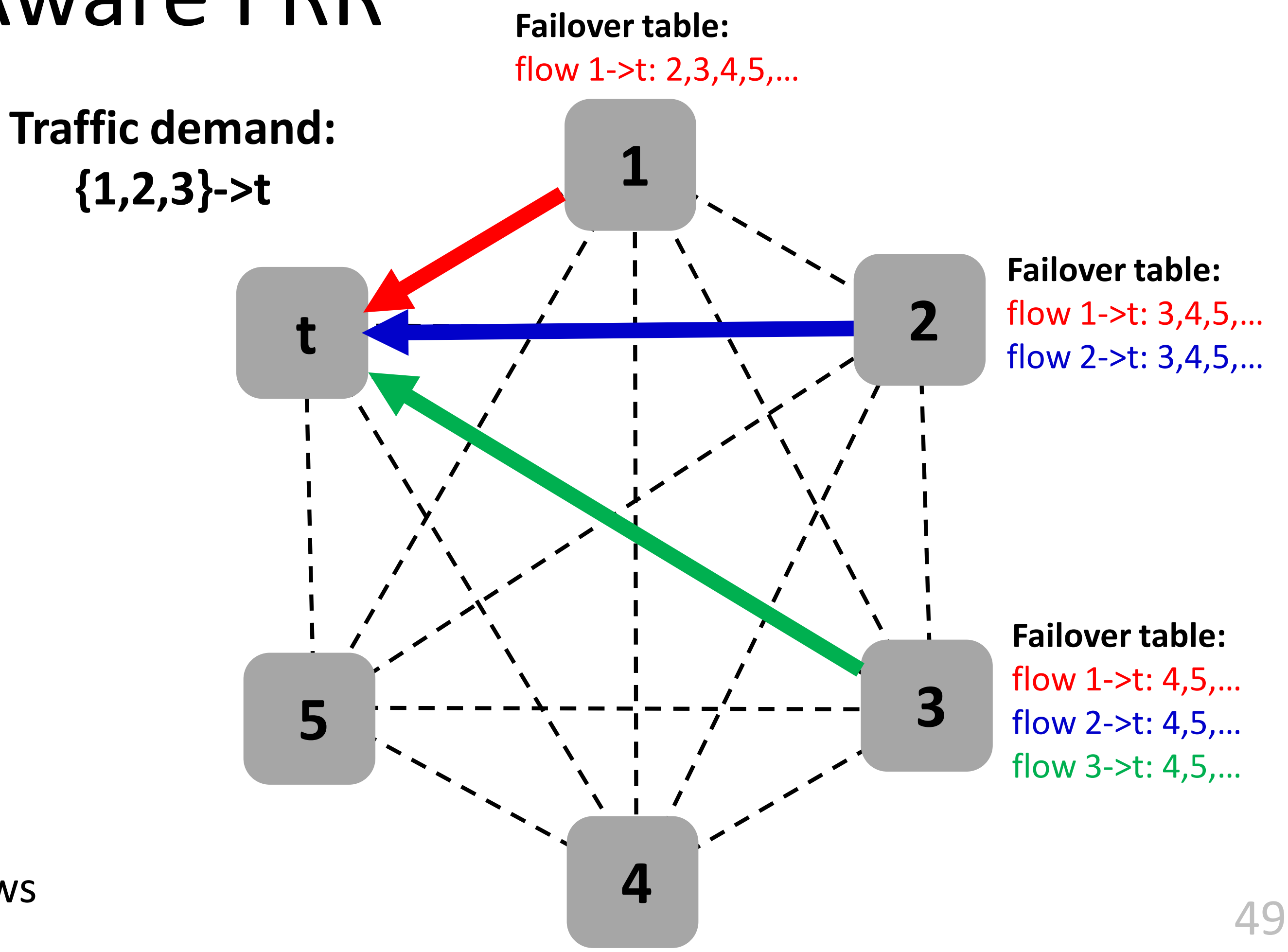
Without failures!

Congestion-Aware FRR



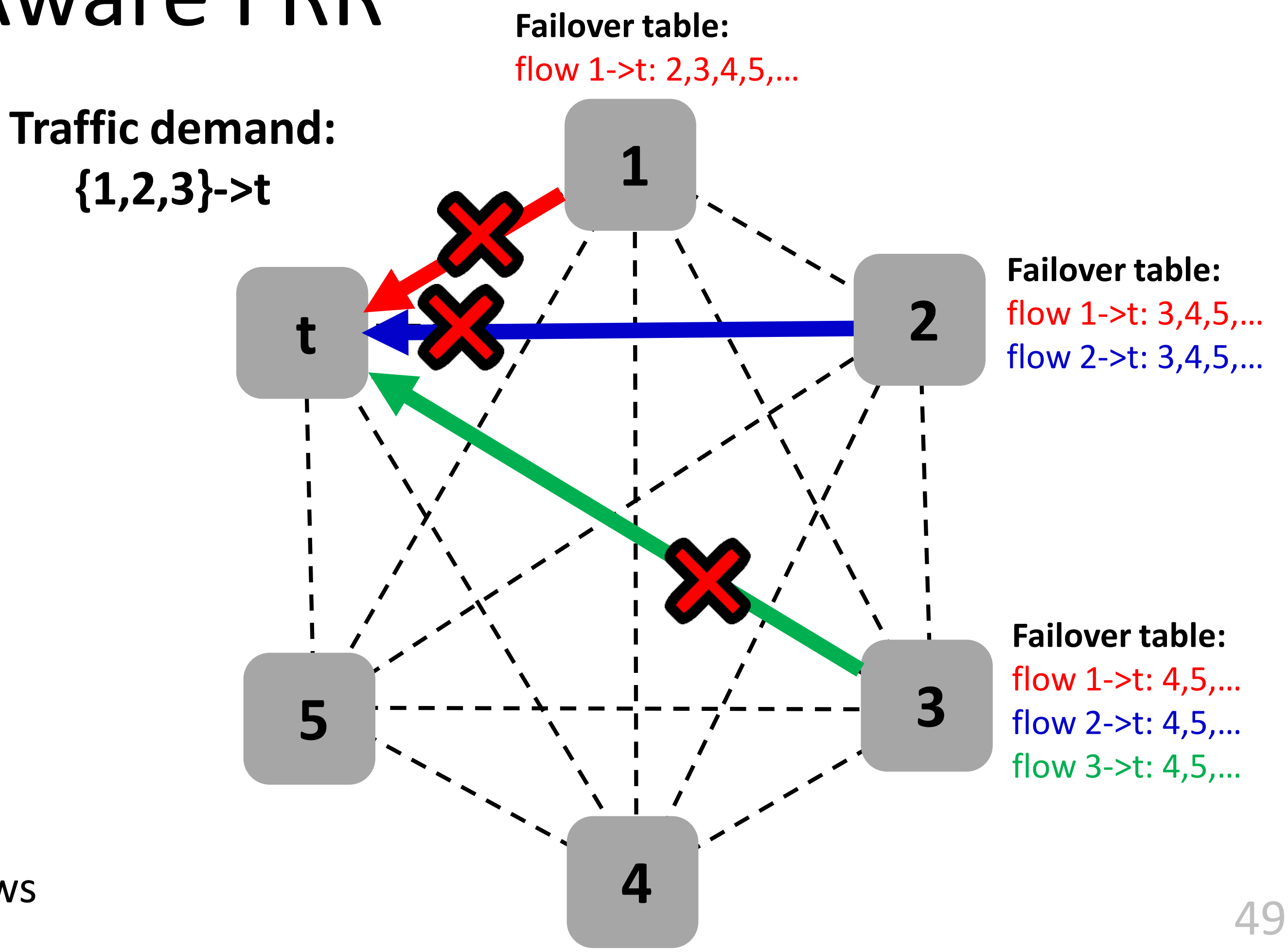
Preinstalled failover rules for red flow

Congestion-Aware FRR



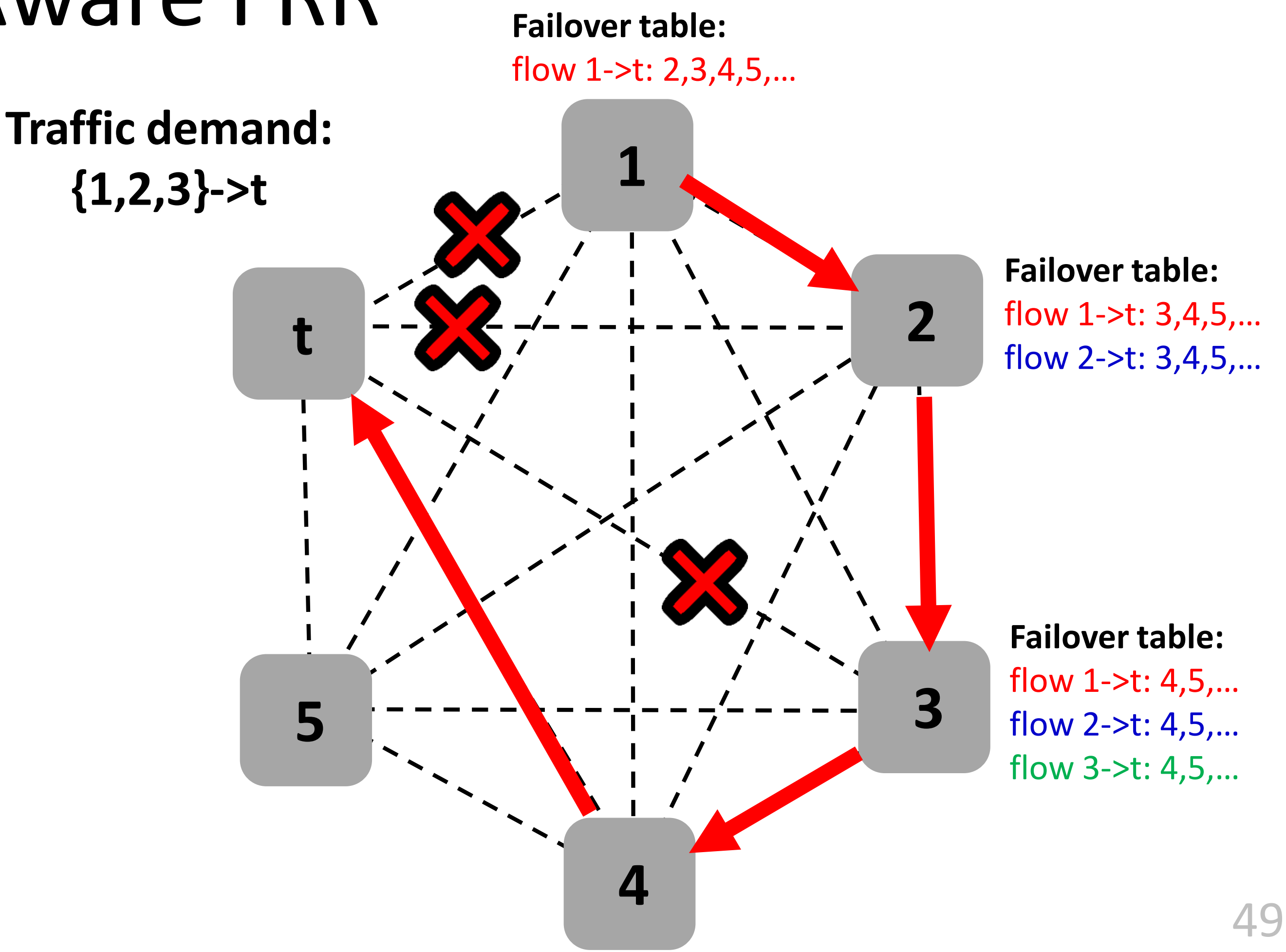
Preinstalled failover rules
for red, blue and green flows

Congestion-Aware FRR



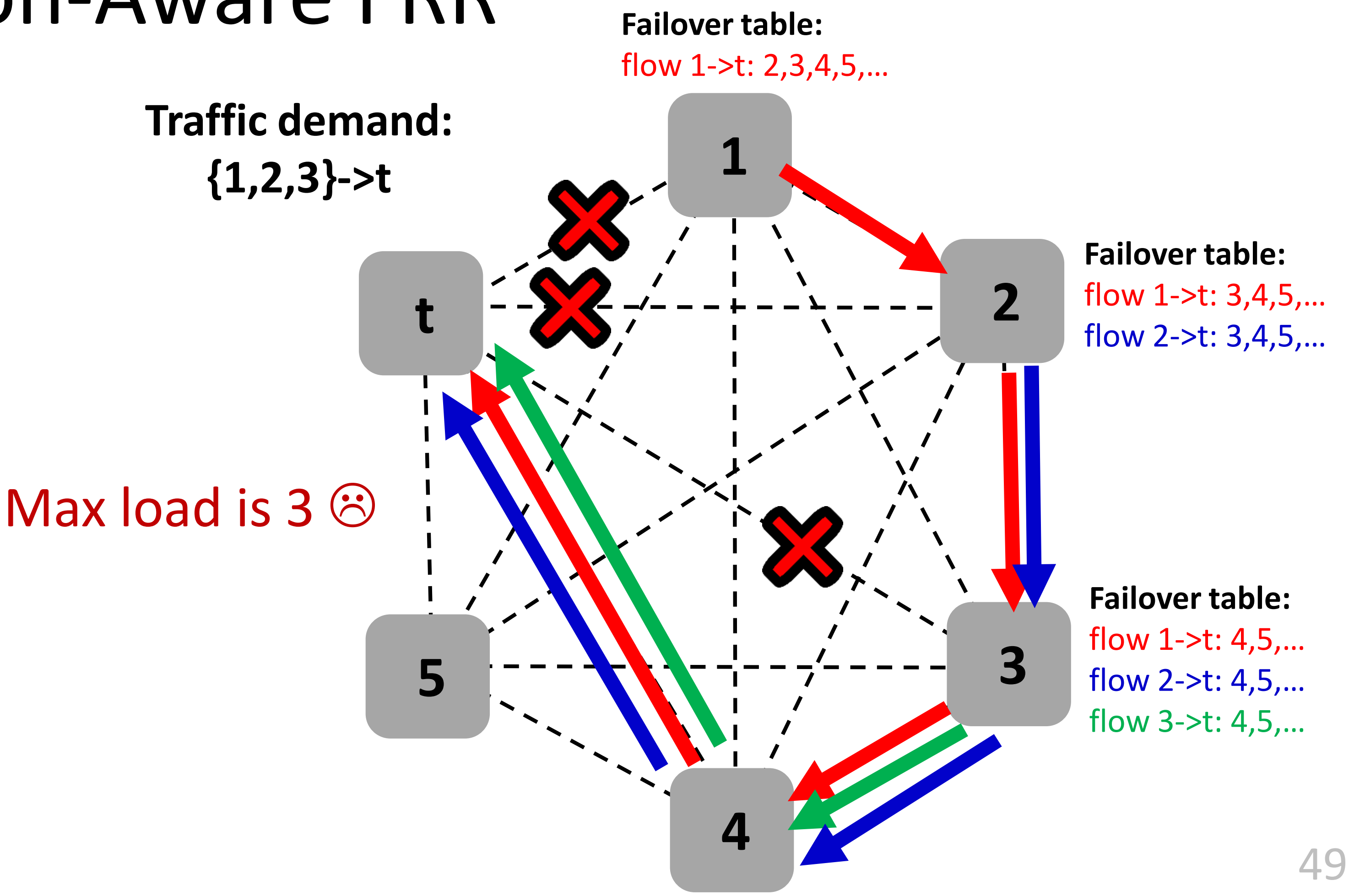
Preinstalled failover rules
for red, blue and green flows

Congestion-Aware FRR



Finally, *t* is reached!

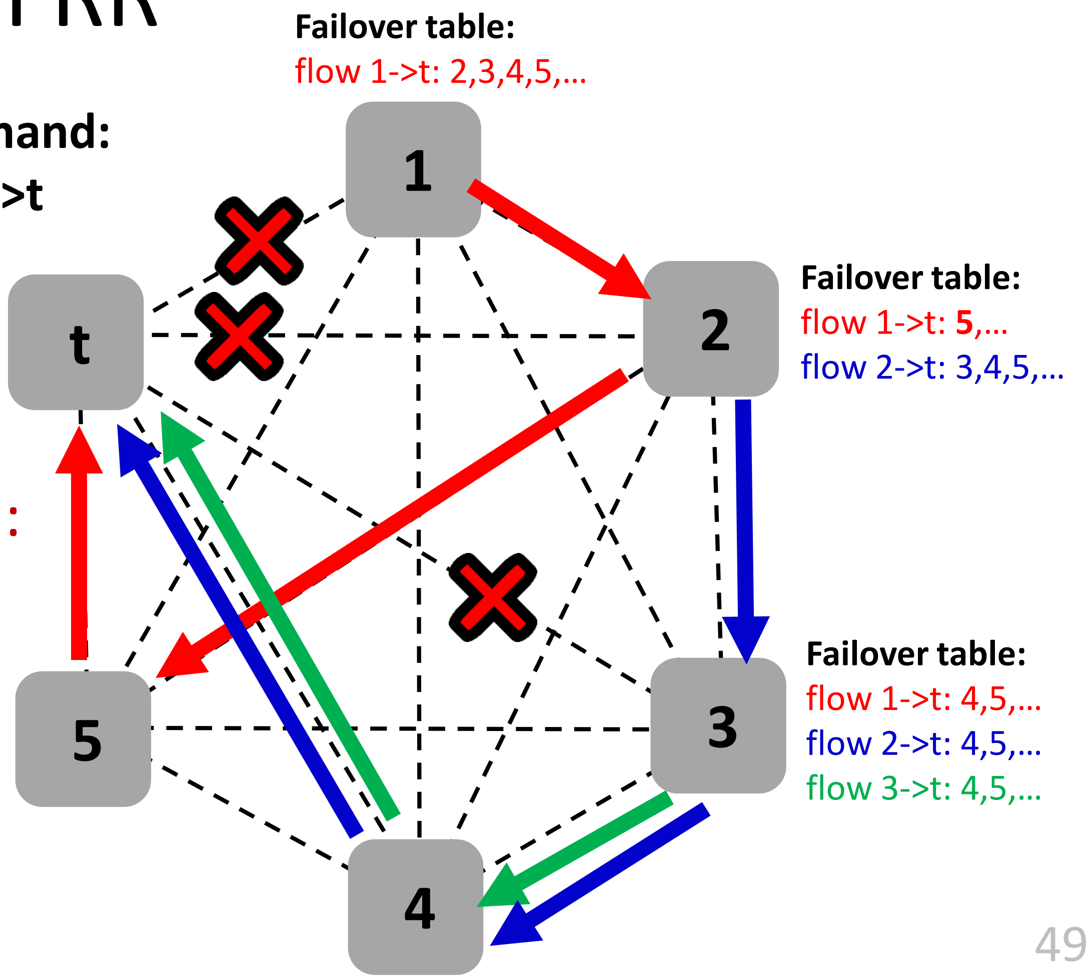
Congestion-Aware FRR



Congestion-Aware FRR

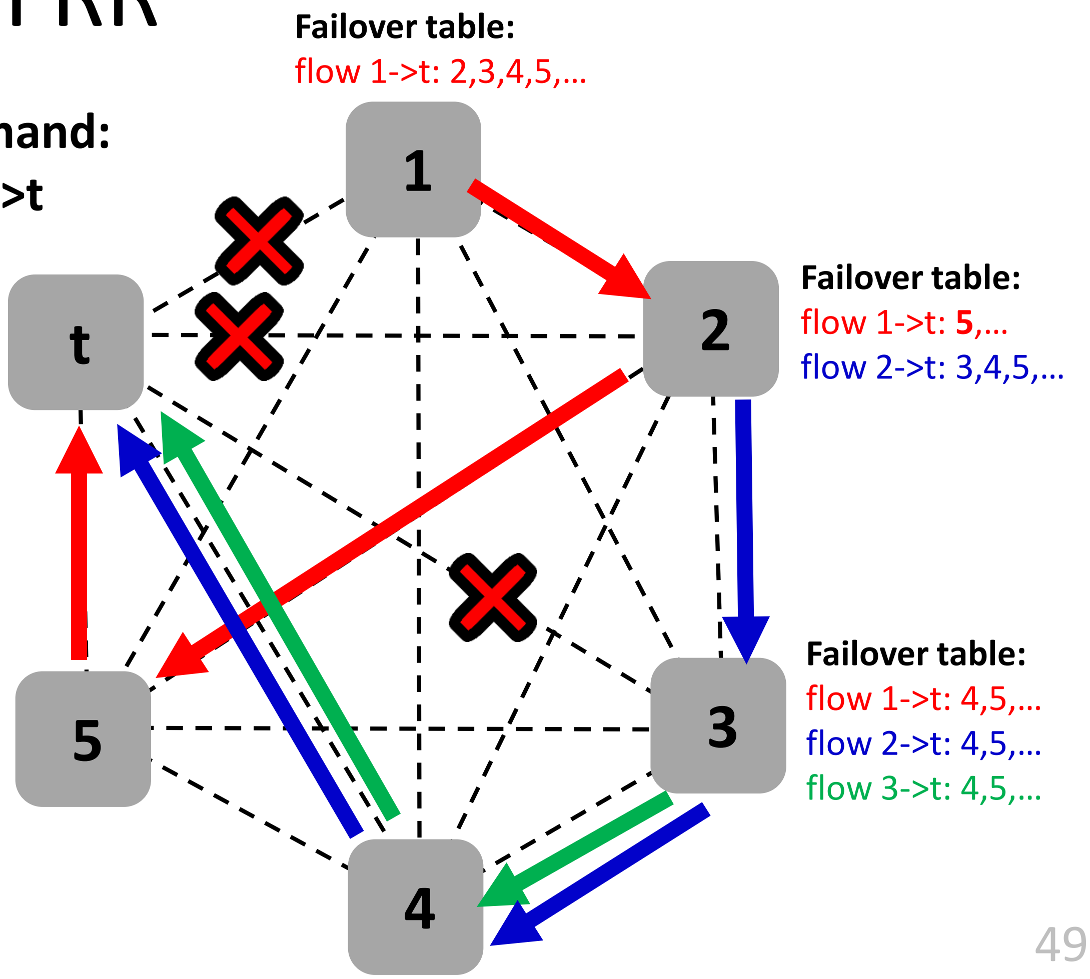
Traffic demand:
{1,2,3}->t

A better solution:
load 2 ☺



Congestion-Aware FRR

Traffic demand:
{1,2,3}->t

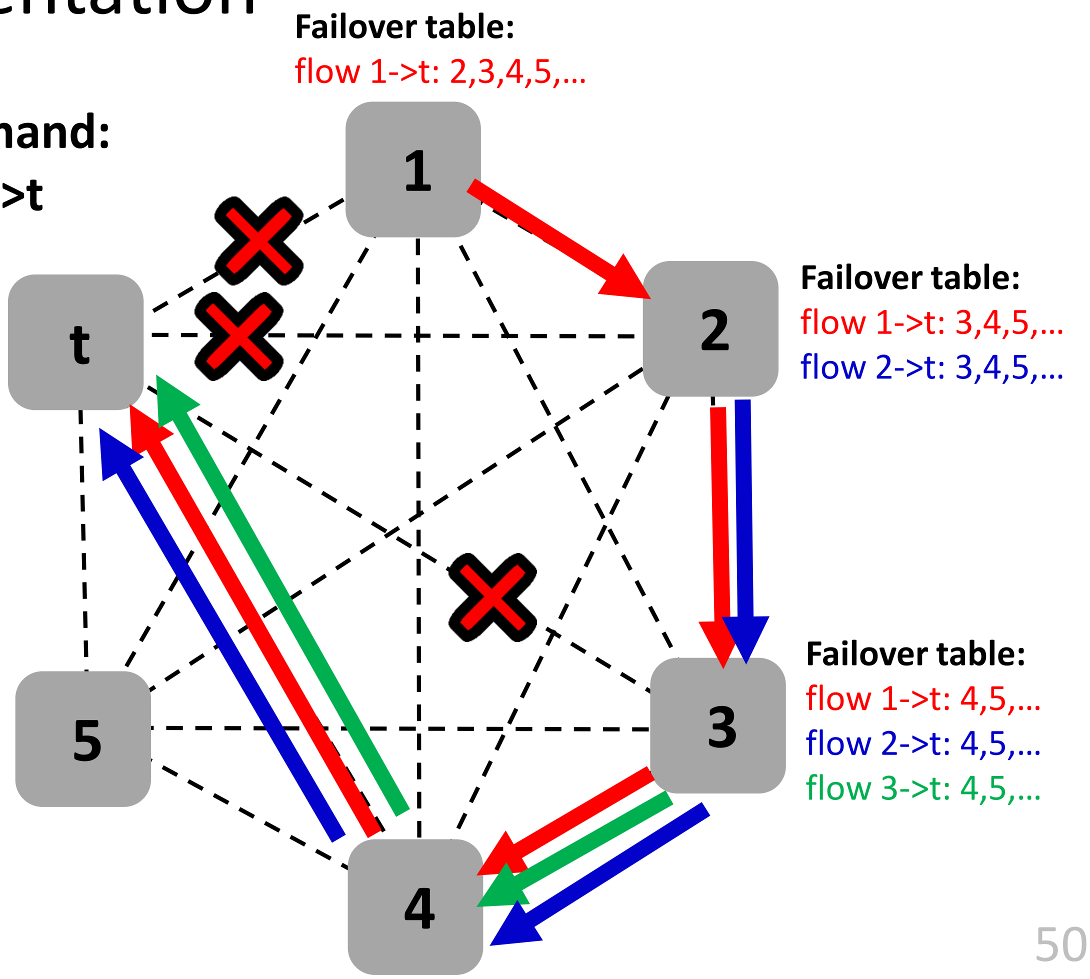


Observation: we can represent failover tables as a matrix. To load balance: prefixes of rows should be different!

Failover Matrix Representation

Traffic demand:
{1,2,3}->t

Matrix:
source 1: 2,3,4,5
source 2: 3,4,5,1
source 3: 4,5,1,2

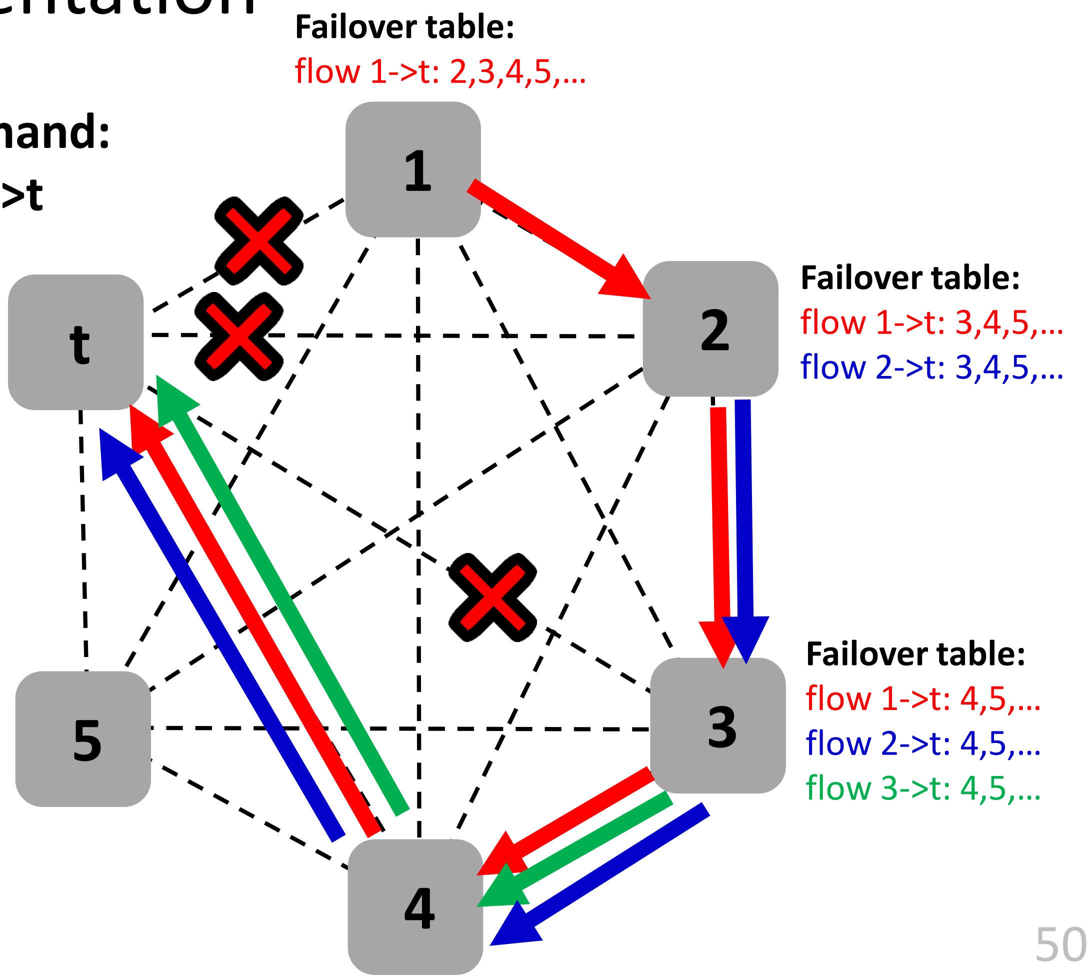


Failover Matrix Representation

Traffic demand:
{1,2,3}->t

Matrix:
source 1: 2,3,4,5
source 2: 3,4,5,1
source 3: 4,5,1,2

Problem: failing link (3,t) will affect all three rerouted flows… In general: easy to create high load on node 4, as failures can be „reused".



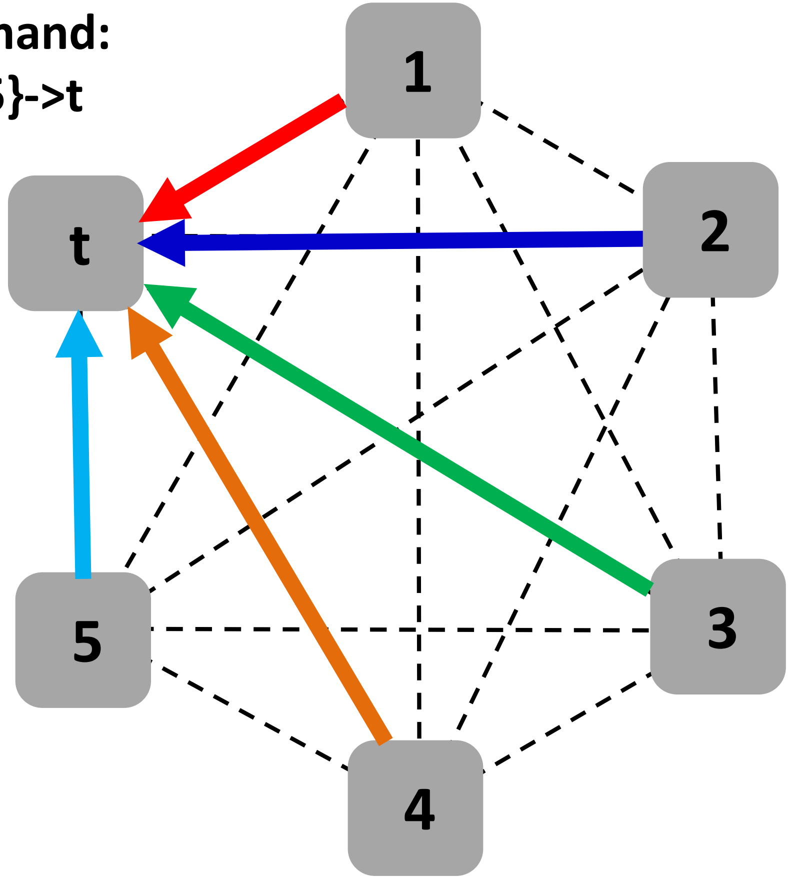
What Are Good Failover Matrices?

- The matrices should be **Latin squares**: each node appears exactly once on each row and each column. No repetitions implies loop-freedom.

- Latin squares property gives high resilience, but is not sufficient for minimizing load.

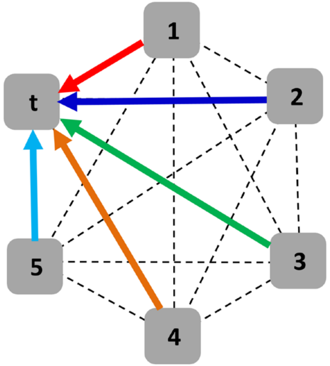
Challenging Example: Incast

Traffic demand:
{1,2,3,4,5}->t



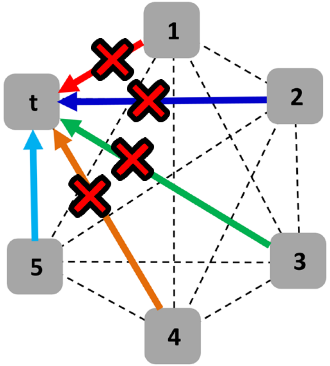
In the following, consider *all-to-one* demand pattern.

A Bad Matrix for Load



Src 1:	2	3	4	5
Src 2:	3	4	5	1
Src 3:	4	5	1	2
Src 4:	5	1	2	3
Src 5:	1	2	3	4

A Bad Matrix for Load

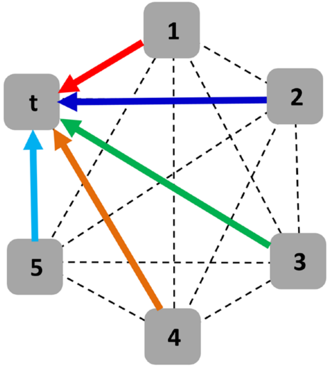


Src 1:	2	3	4	5
Src 2:	3	4	5	1
Src 3:	4	5	1	2
Src 4:	5	1	2	3
Src 5:	1	2	3	4

Failing (1,t), (2,t), (3,t), (4,t), gives load 4 on node 5 / link (5,t).

If the adversary fails the *l* first links to destination *d* (that is, $\{(v_i,t), i = 1, \ldots, l\}$), then *l* sources will route through (v_{i+1},t). Load *l* for *l* failures. Can we do better?

Good Failover Matrices?

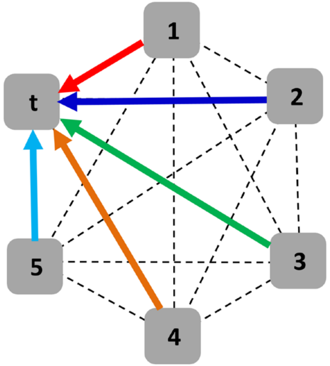


- To bring the flow from a source i to a node X, need to fail *all links* in corresponding *row*
  - Worst case: all *to destination*

- The same for each other flow/row which should reach X

					X
		X			
i				X	
			X		
					X
			X		

Good Failover Matrices?



- To bring the flow from a source i to a node X, need to fail *all links* in corresponding *row*
  - Worst case: all *to destination*

- The same for each other flow/row which should reach X

- Adversary will try to *reuse link* failures: **good matrices** have *prefixes with little overlap* (resp. large number of unique nodes)

					X
		X			
i				X	
			X		
					X
			X		

Connection to Block Designs

- A closely related problem: generating **block designs**
  - and its geometric counterpart, generating **projective planes** of high order

- Using *symmetric balanced incomplete block designs (BIBDs)*

- Gives a latin failover matrix M with intersection properties representing a failover scheme that is ***optimal up to a constant factor***

- Also used in the context **disconnected cooperation**, e.g.:
  - G. Malewicz, A. Russell, and A. A. Shvartsman. Distributed Scheduling for Disconnected Cooperation. Distributed Computing, 18(6), 2005.

Overview of Results

Good news: Theory of local algorithms without communication: symmetric block design theory.

Bad news (counting argument): High load unavoidable even in well-connected residual networks: a price of locality. Given L failures, load at least √L, although network still highly connected (n-L connected). E.g., L=n/2, load could be 2 still, but due to locality at least √n.

Borokhovich et al. **Load-Optimal Local Fast Rerouting for Dense Networks.** IEEE/ACM Transactions on Networking (TON), 2018.

Randomized Failover

- Recall: deterministic lower bound of *√L* for *L* failures, although load could be O(1) for L<L/2. A large ***price of locality***.

- So what about ***randomized*** approaches?

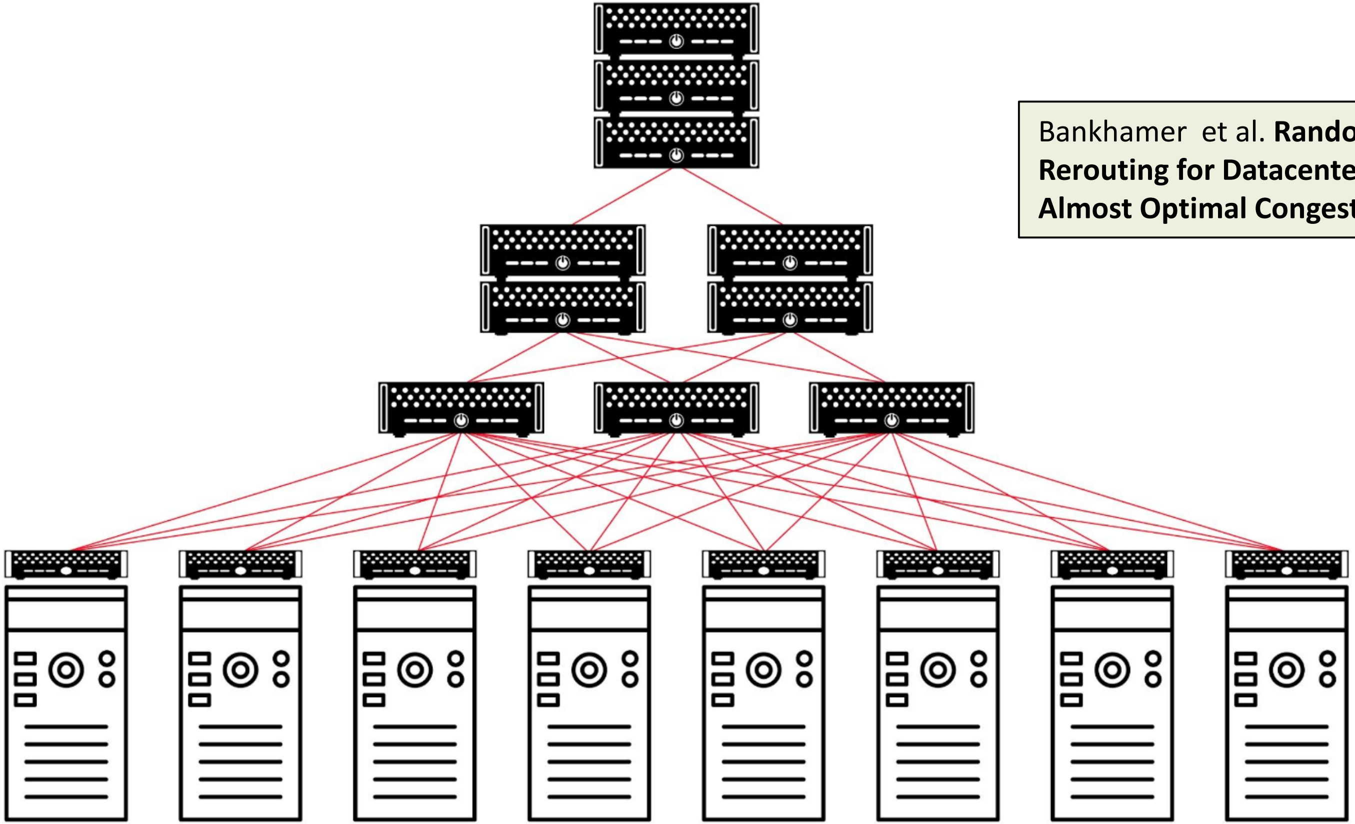
The Power of Randomization

	3-Permutations	*Intervals*	*Shared-Permutations*
Rule Set	Destination + Hop	Destination	Destination + Hop
Resilience	$\Theta(n)$	$\Theta(n/\log n)$	$\Theta(n)$
Congestion	$\mathcal{O}(\log^2 n \cdot \log\log n)$	$\mathcal{O}(\log n \cdot \log\log n)$	$\mathcal{O}(\sqrt{\log n})$

- While deterministic algorithms can at best achieve a ***polynomial*** load, randomized algorithms can achieve a ***polylogarithmic load***.
- Even when just matching the destination.
  - Losing a log n factor in resilience.
  - Matching also the hop count can overcome this.

Bankhamer et al. **Local Fast Rerouting with Low Congestion: A Randomized Approach**. 27th IEEE International Conference on Network Protocols (ICNP), 2019.

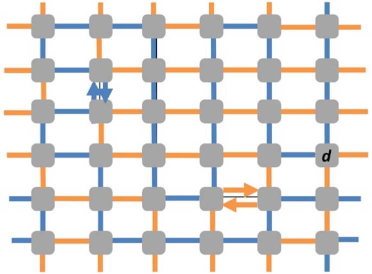
Benefits in Datacenter Networks

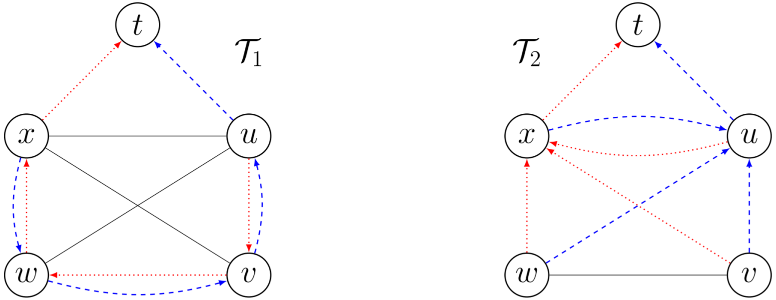


Bankhamer et al. **Randomized Local Fast Rerouting for Datacenter Networks with Almost Optimal Congestion**. DISC, 2021.

What About Path Length and Stretch?

- So far: ignored the length of the failover routes
  - Hamilton cycles are particularly bad
  - The heights of general arborescences may be lower



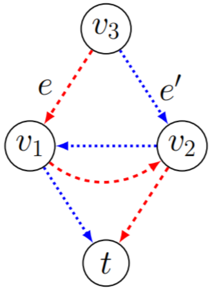


- Idea (so far heuristic):
  - Postprocess the arborescences to lower their heights
  - Two different t-rooted arc-disjoint spanning arborescence decompositions, T1 and T2
  - The mean path length of T1 is higher than that of T2

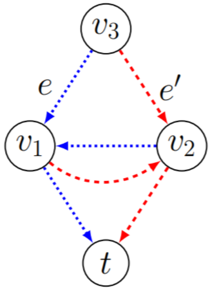
Foerster et al. **Improved Fast Rerouting Using Postprocessing** (Best Paper Award). 38th International Symposium on Reliable Distributed Systems (SRDS), 2019.

Swapping Operations Which Maintain Decomposition

①

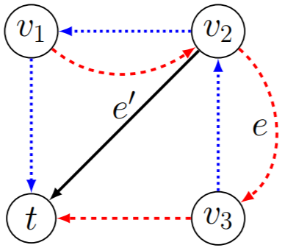


Before swapping

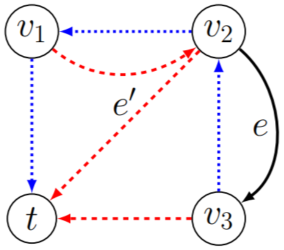


After swapping

②



Before swapping



After swapping

Roadmap

- A Brief History of Resilient Networking
- Algorithms for Local Fast Re-Routing (FRR)
- Accounting for Congestion
- **Accounting for Network Policy**

Roadmap

- A Brief History of Resilient Networking
- Algorithms for Local Fast Re-Routing (FRR)
- Accounting for Congestion
- **Accounting for Network Policy**

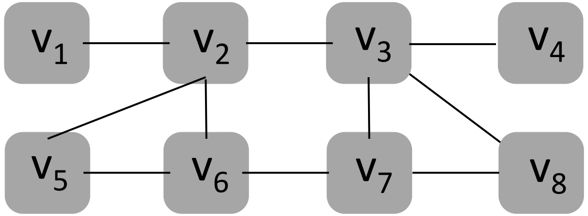
An example with header rewriting.

Case Study: MPLS Networks

- Widely deployed networks by Internet Service Providers (**ISPs**)
- Often used for **traffic engineering**
  - Avoid congestion by going non-shortest paths
- Allows for ***header re-writing*** upon failures
  - Header based on **stack of labels**

How (MPLS) Networks Work

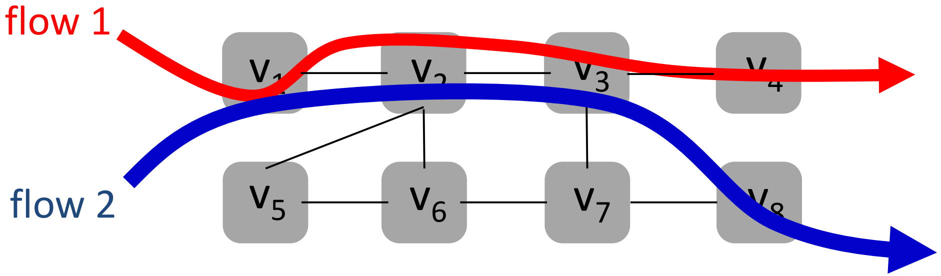
- Forwarding based on **top label** of label **stack**



Default routing of
two flows

How (MPLS) Networks Work

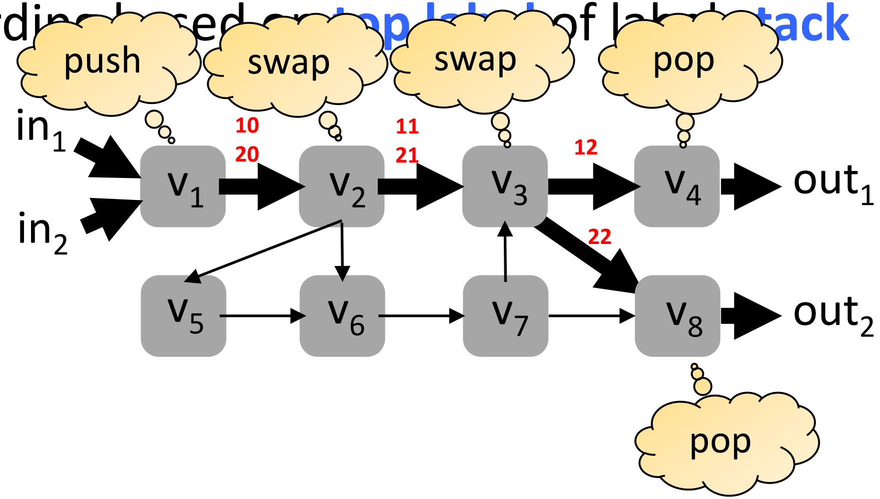
- Forwarding based on **top label** of label **stack**



Default routing of two flows

How (MPLS) Networks Work

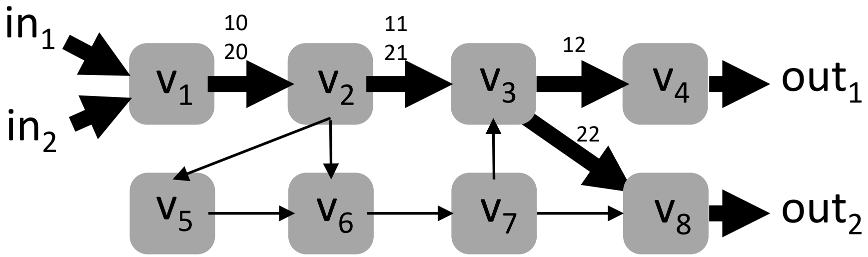
- Forwarding based on **top label** of label **stack**



Default routing of two flows

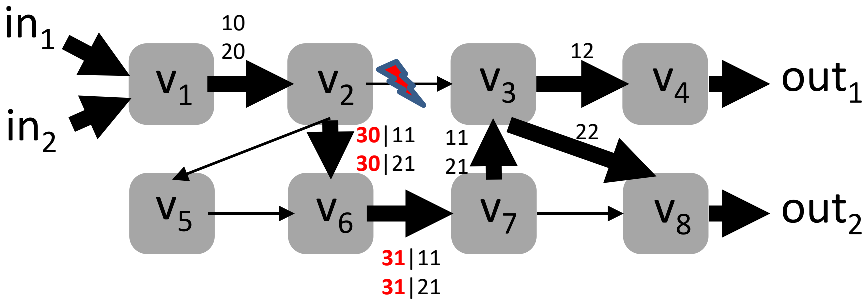
Fast Reroute Around *1 Failure*

- Forwarding based on **top label** of label **stack** (in packet header)



Default routing of two flows

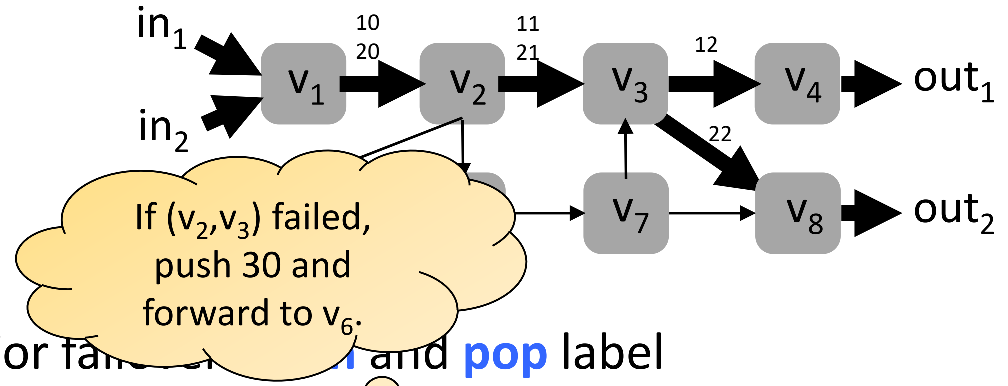
- For failover: **push** and **pop** label



One failure: push 30: route around (v_2,v_3)

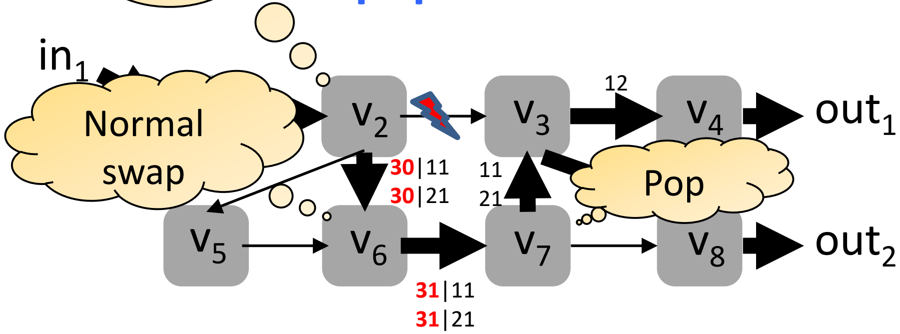
Fast Reroute Around 1 Failure

- Forwarding based on **top label** of label **stack** (in packet header)



Default routing of two flows

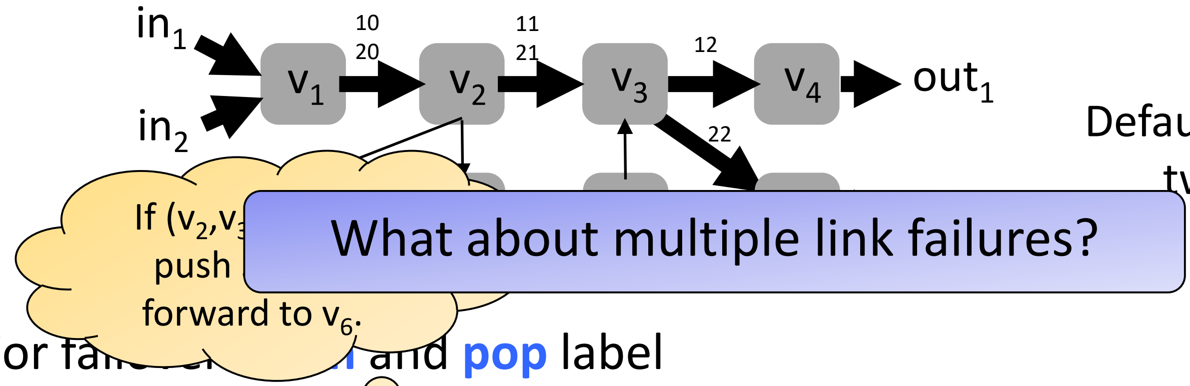
- For failover push and **pop** label



One failure: push 30: route around (v_2,v_3)

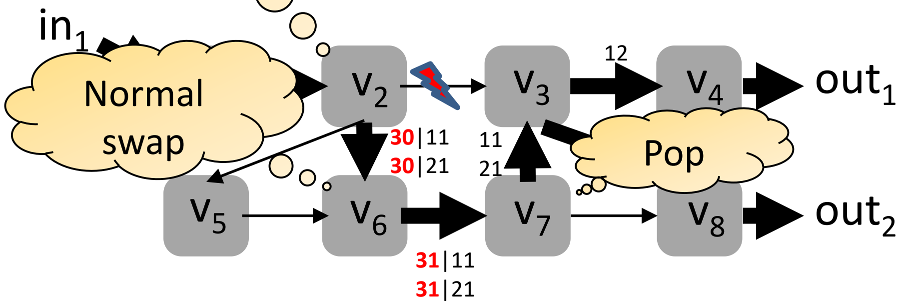
Fast Reroute Around *1 Failure*

- Forwarding based on **top label** of label **stack** (in packet header)



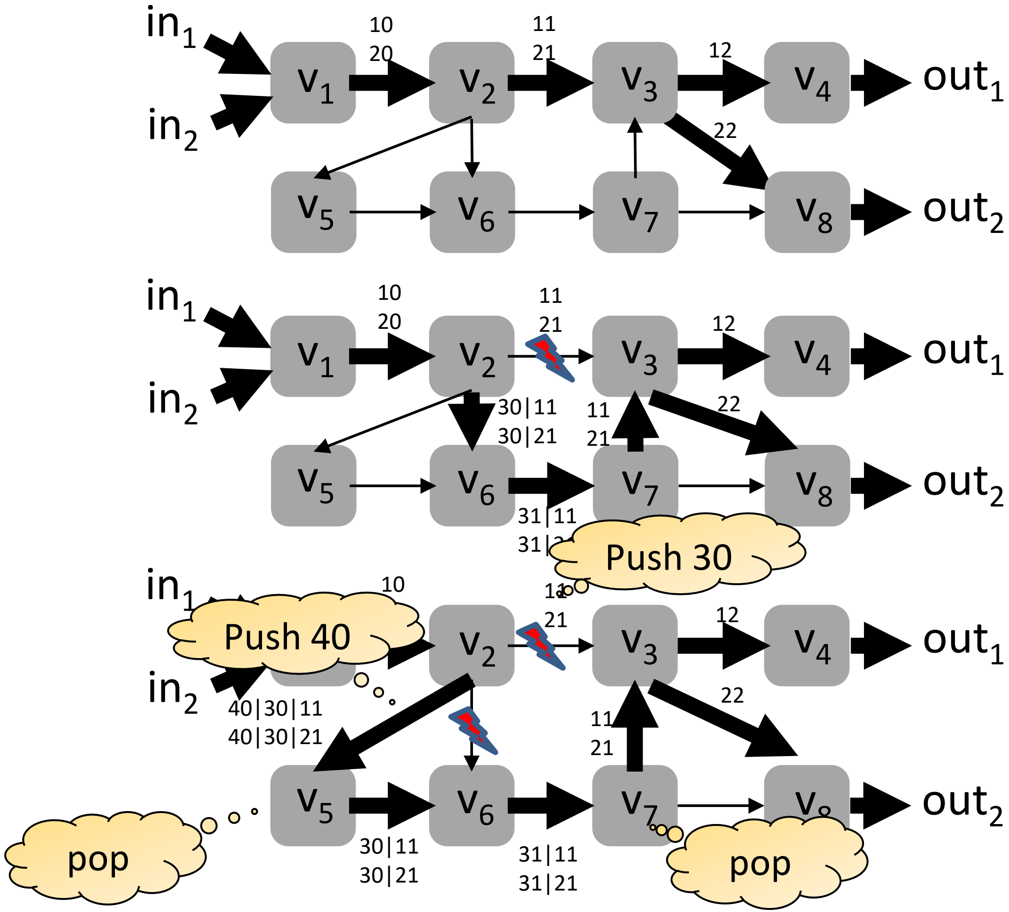
Default routing of two flows

- For fa... and **pop** label



One failure: push 30: route around (v_2,v_3)

2 Failures: Push *Recursively*



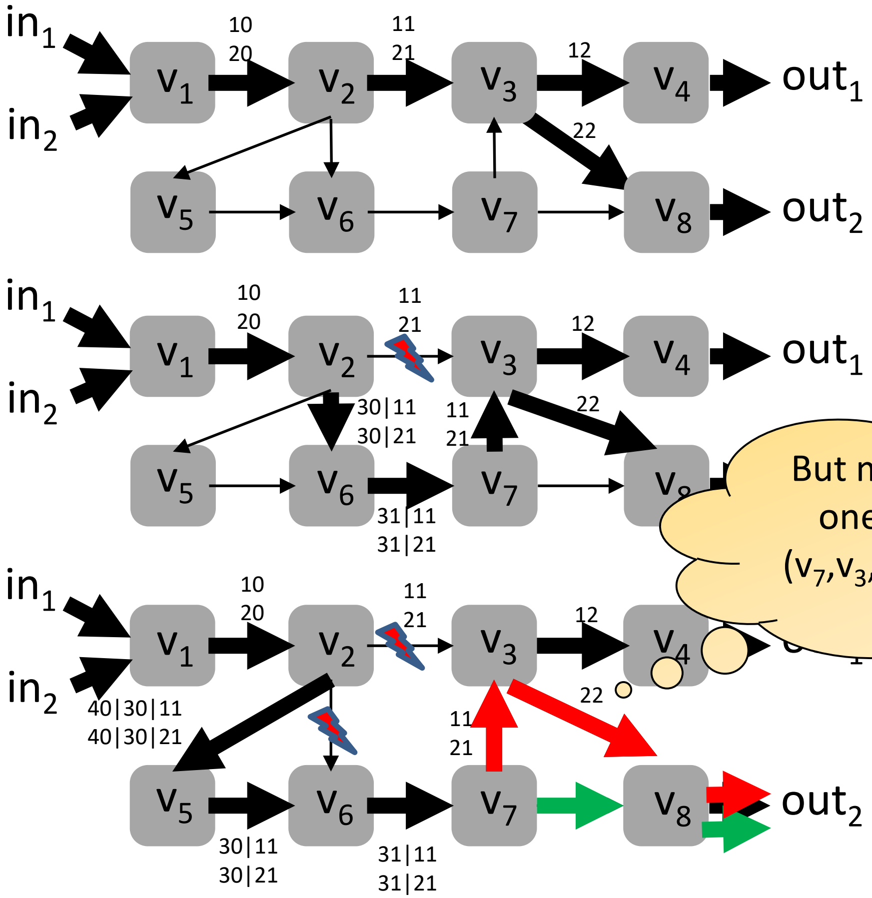
Original Routing

One failure: push 30: route around (v_2,v_3)

Two failures: first push 30: route around (v_2,v_3)

Push recursively 40: route around (v_2,v_6)

2 Failures: Push *Recursively*



Original Routing

One failure: push 30:

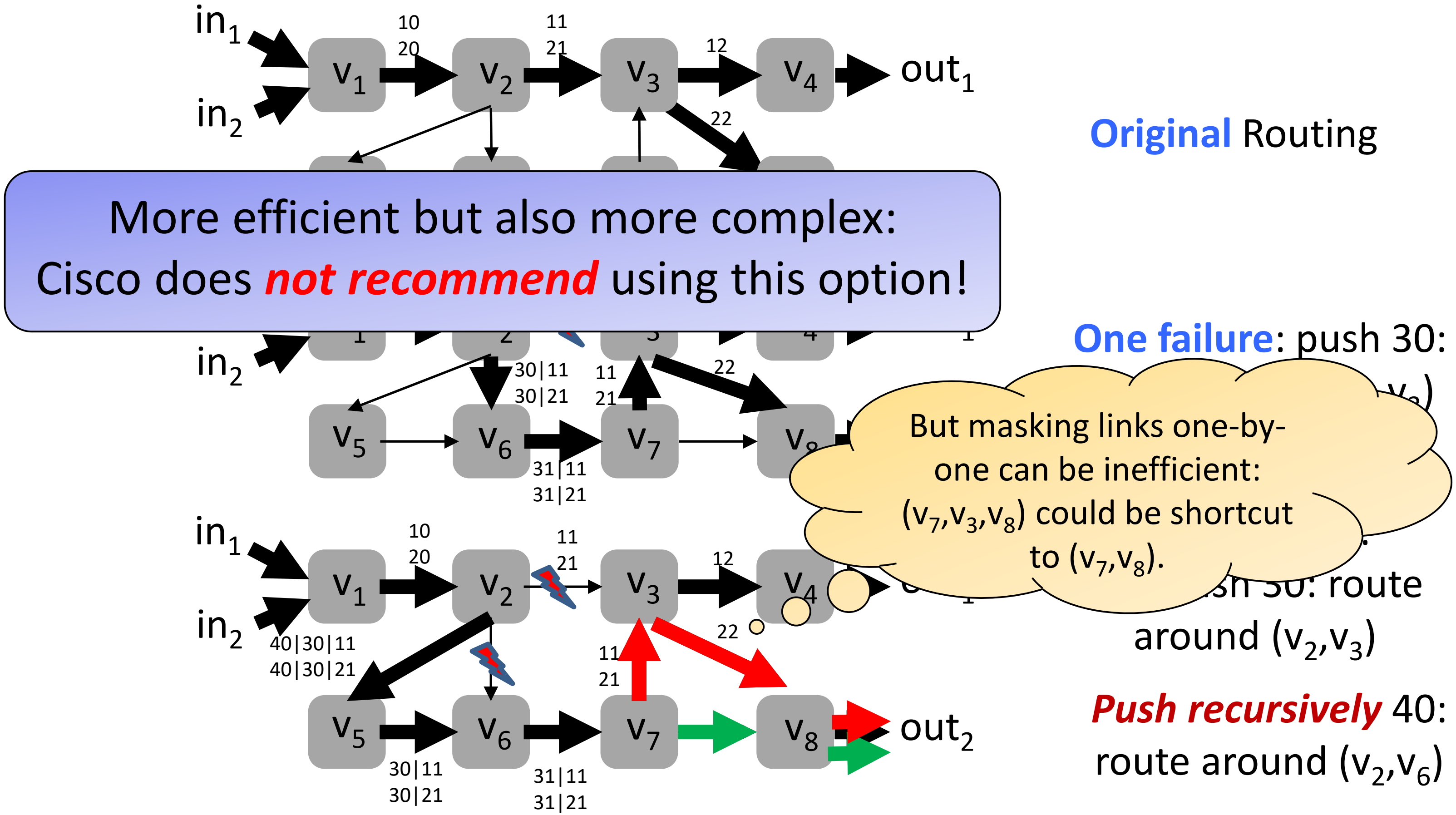
v_3)

But masking links one-by-one can be inefficient: (v_7,v_3,v_8) could be shortcut to (v_7,v_8).

sh 30: route around (v_2,v_3)

Push recursively 40: route around (v_2,v_6)

2 Failures: Push *Recursively*



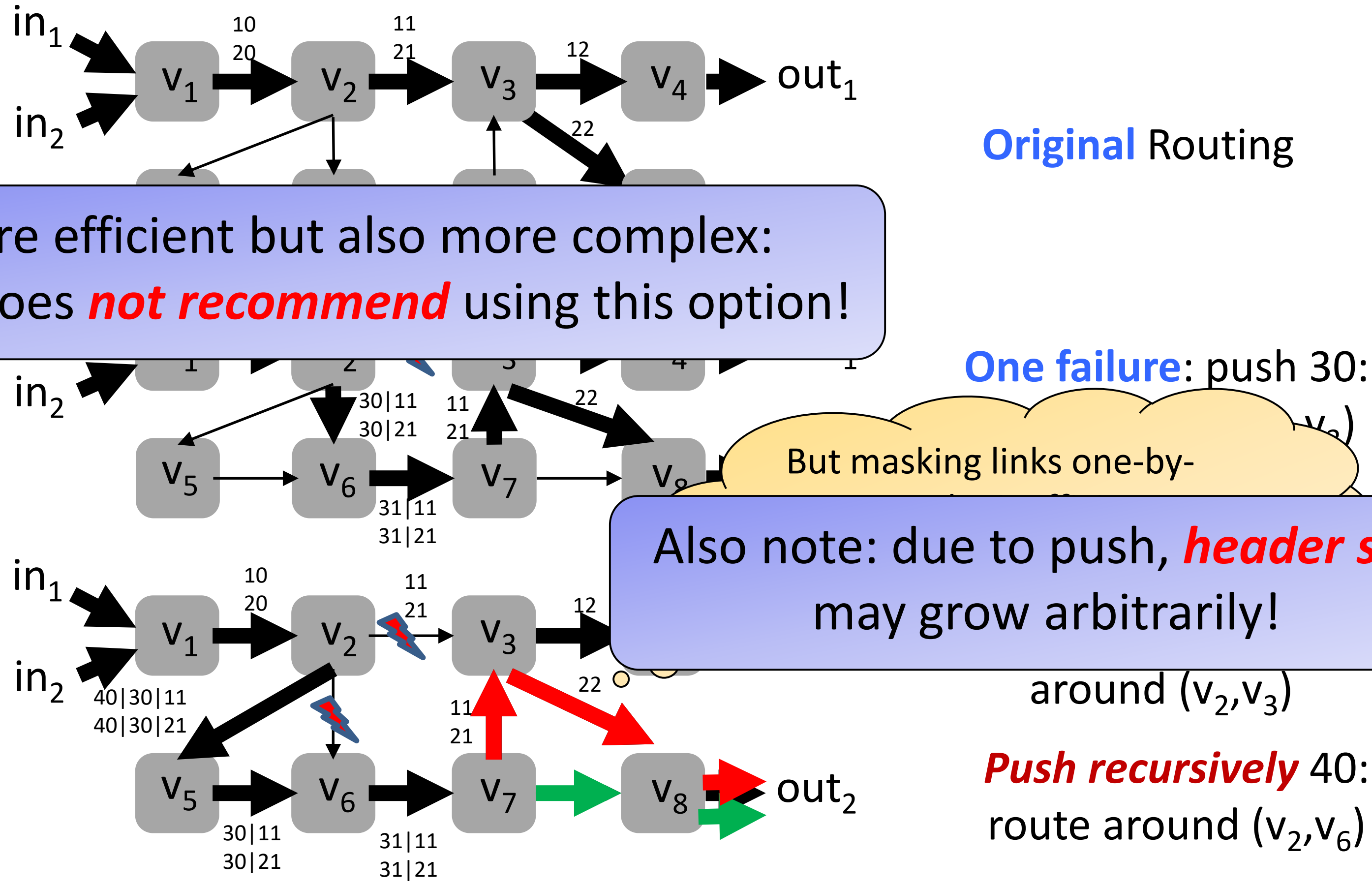
Original Routing

One failure: push 30:

around (v$_2$,v$_3$)

Push recursively 40: route around (v$_2$,v$_6$)

2 Failures: Push *Recursively*



Original Routing

More efficient but also more complex:
Cisco does ***not recommend*** using this option!

One failure: push 30:

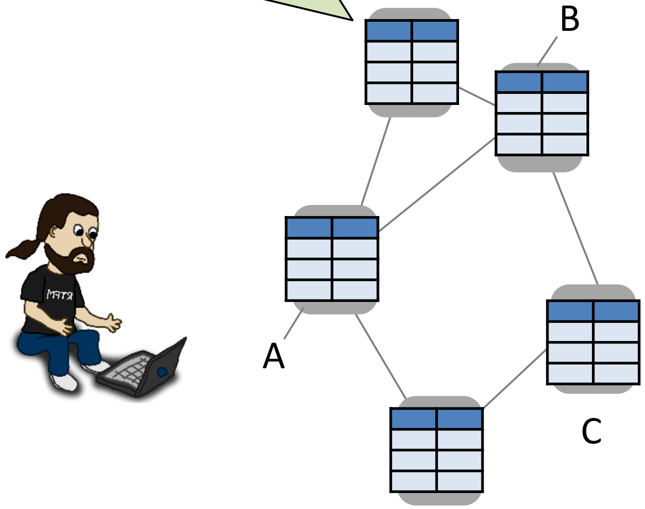
Also note: due to push, ***header size*** may grow arbitrarily!

around (v_2,v_3)

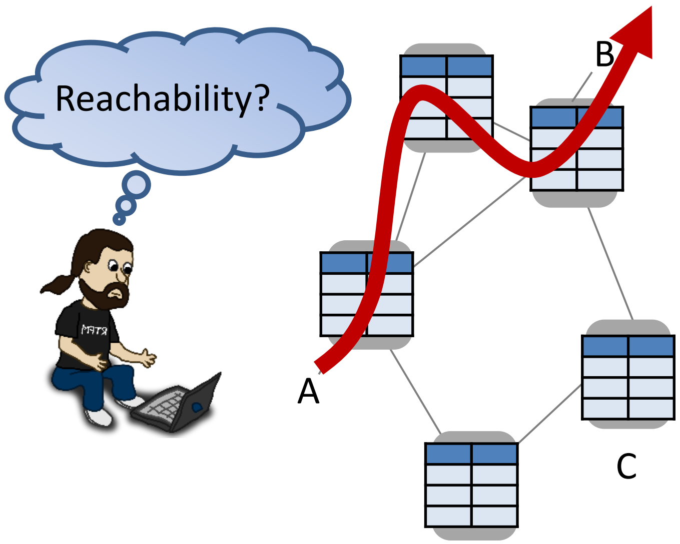
Push recursively 40:
route around (v_2,v_6)

Responsibilities of a Sysadmin

Routers and switches store list of forwarding rules, and conditional failover rules.



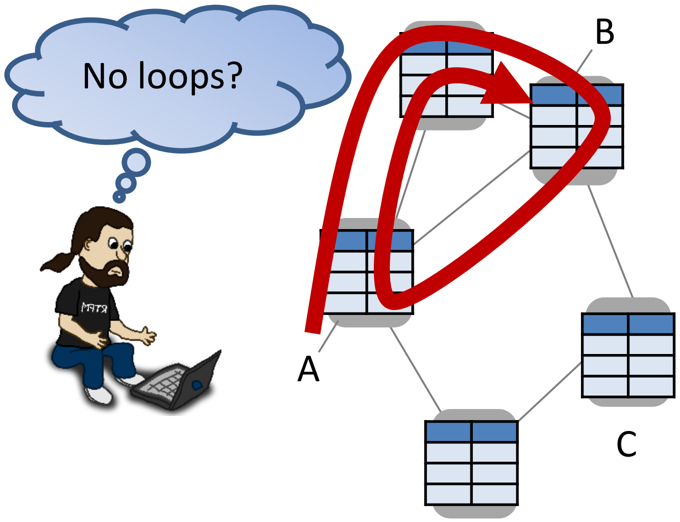
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?

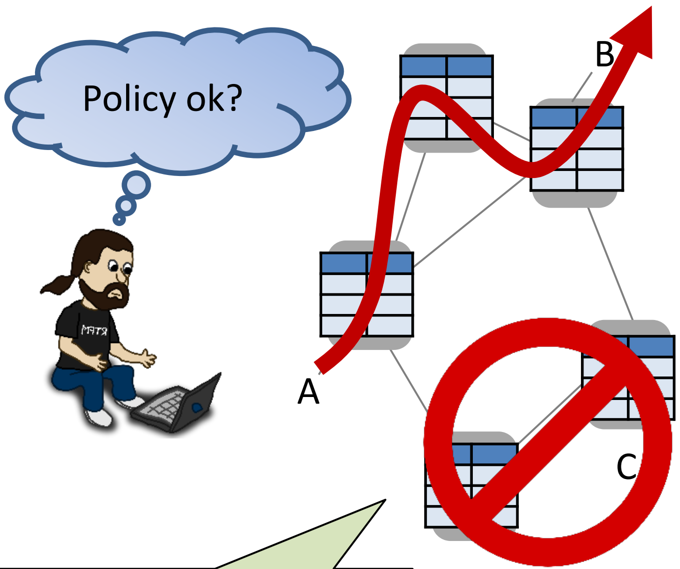
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?

Responsibilities of a Sysadmin

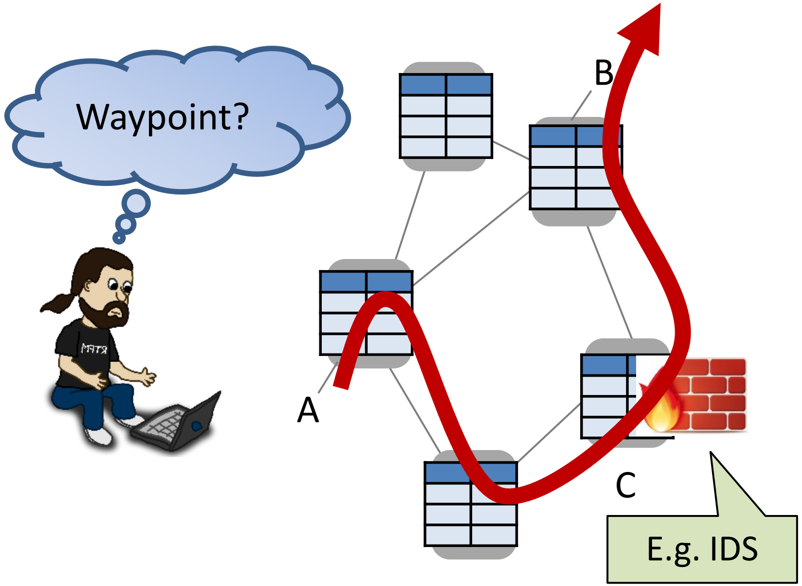


E.g. ***NORDUnet***: no traffic via Iceland (expensive!).

Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?

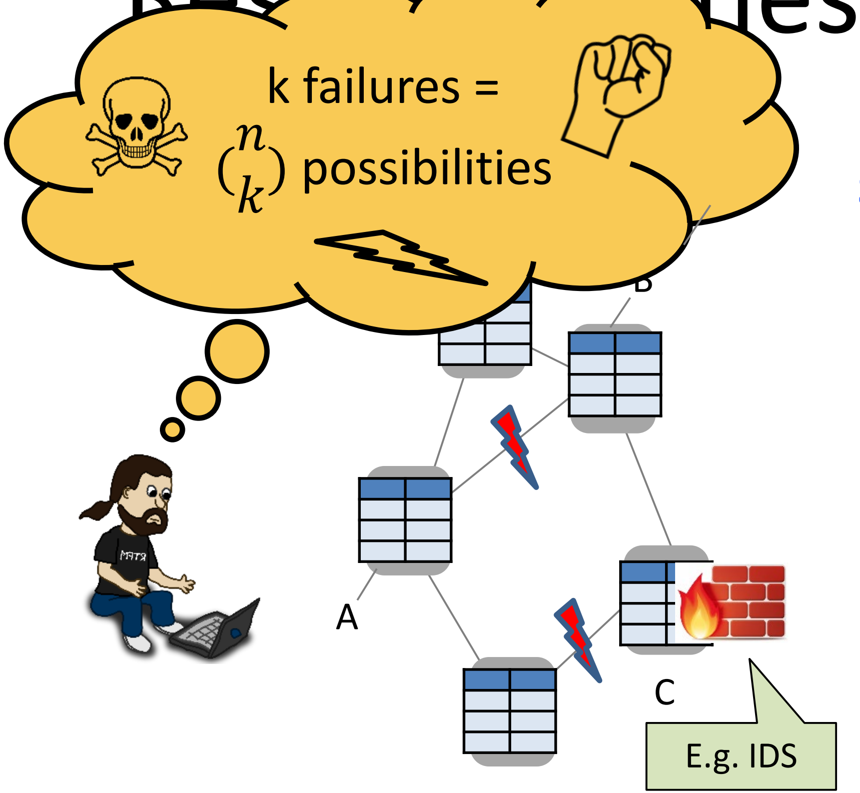
Responsibilities of a Sysadmin



Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?
- **Waypoint enforcement:** Is it ensured that traffic from A to B is always routed via a node C (e.g., intrusion detection system or a firewall)?

Responsibilities of a Sysadmin

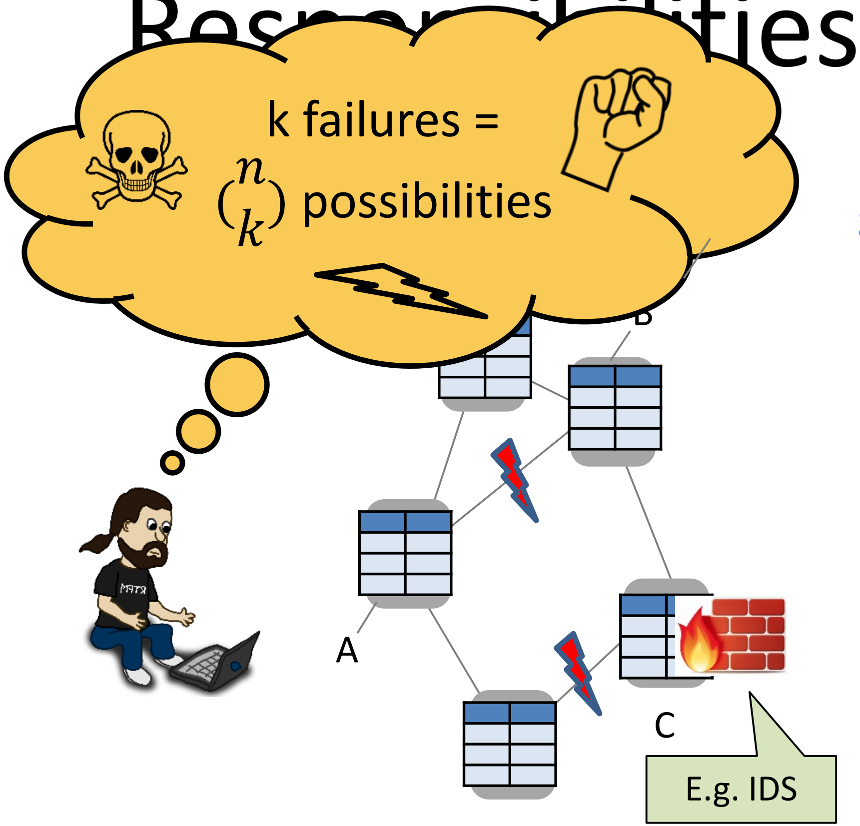


Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?
- **Waypoint enforcement:** Is it ensured that traffic from A to B is always routed via a node C (e.g., intrusion detection system or a firewall)?

… and everything even under multiple failures?!

Responsibilities of a Sysadmin



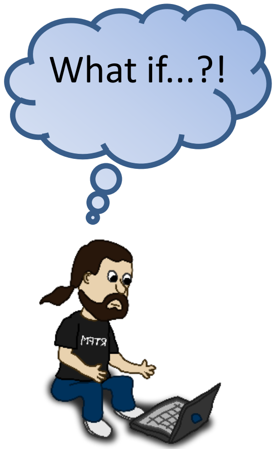
Sysadmin responsible for:

- **Reachability:** Can traffic from ingress port A reach egress port B?
- **Loop-freedom:** Are the routes implied by the forwarding rules loop-free?
- **Policy:** Is it ensured that traffic from A to B never goes via C?
- **Waypoint enforcement:** Is it ensured that traffic from A to B is always routed via a node C (e.g., intrusion detection system or a firewall)?

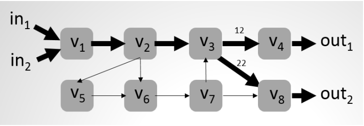
… and everything even under multiple failures?!

Generalization: service chaining!

Approach: Automation and Formal Methods



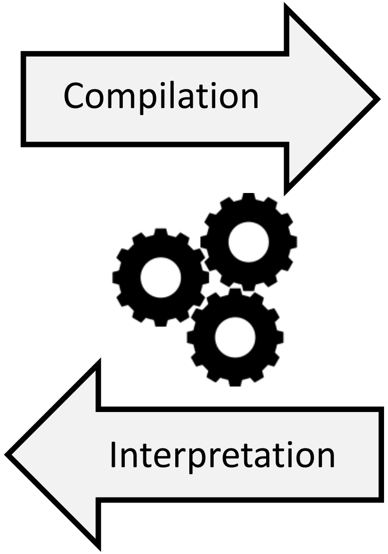
FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\perp$	(v_1, v_2)	$push(10)$
	in_2	$\perp$	(v_1, v_2)	$push(20)$
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	$swap(11)$
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	(v_5, v_6)	pop
τ_{v_6}	(v_2, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	61	(v_6, v_7)	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop



local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$

global FFT	Out-I	In-Label	Out-I	op
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$

Router **configurations**
(Cisco, Juniper, etc.)

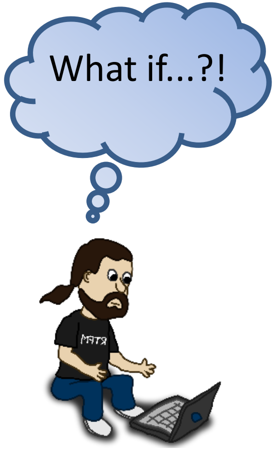


$$pX \Rightarrow qXX$$
$$pX \Rightarrow qYX$$
$$qY \Rightarrow rYY$$
$$rY \Rightarrow r$$
$$rX \Rightarrow pX$$

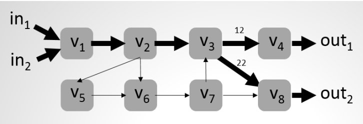
Pushdown Automaton and
Prefix Rewriting Systems

Approach: Autom… ods

Use cases: Sysadmin ***issues queries*** to test certain properties, or do it on a ***regular basis*** automatically!

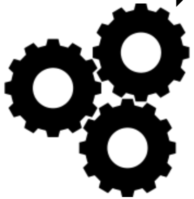


FT	In-I	In-Label	Out-I	op
τ_{v_1}	in_1	$\bot$	(v_1, v_2)	$push(10)$
	in_2	$\bot$	(v_1, v_2)	$push(20)$
τ_{v_2}	(v_1, v_2)	10	(v_2, v_3)	$swap(11)$
	(v_1, v_2)	20	(v_2, v_3)	$swap(21)$
τ_{v_3}	(v_2, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_2, v_3)	21	(v_3, v_8)	$swap(22)$
	(v_7, v_3)	11	(v_3, v_4)	$swap(12)$
	(v_7, v_3)	21	(v_3, v_8)	$swap(22)$
τ_{v_4}	(v_3, v_4)	12	out_1	pop
τ_{v_5}	(v_2, v_5)	40	(v_5, v_6)	pop
τ_{v_6}	(v_2, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	30	(v_6, v_7)	$swap(31)$
	(v_5, v_6)	61	(v_6, v_7)	$swap(62)$
	(v_5, v_6)	71	(v_6, v_7)	$swap(72)$
τ_{v_7}	(v_6, v_7)	31	(v_7, v_3)	pop
	(v_6, v_7)	62	(v_7, v_3)	$swap(11)$
	(v_6, v_7)	72	(v_7, v_8)	$swap(22)$
τ_{v_8}	(v_3, v_8)	22	out_2	pop
	(v_7, v_8)	22	out_2	pop



local FFT	Out-I	In-Label	Out-I	op
τ_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$push(30)$
	(v_2, v_3)	21	(v_2, v_6)	$push(30)$
	(v_2, v_6)	30	(v_2, v_5)	$push(40)$
global FFT	**Out-I**	**In-Label**	**Out-I**	**op**
τ'_{v_2}	(v_2, v_3)	11	(v_2, v_6)	$swap(61)$
	(v_2, v_3)	21	(v_2, v_6)	$swap(71)$
	(v_2, v_6)	61	(v_2, v_5)	$push(40)$
	(v_2, v_6)	71	(v_2, v_5)	$push(40)$

Compilation



Interpretation

$$pX \Rightarrow qXX$$
$$pX \Rightarrow qYX$$
$$qY \Rightarrow rYY$$
$$rY \Rightarrow r$$
$$rX \Rightarrow pX$$

Router **configurations** (Cisco, Juniper, etc.)

Pushdown Automaton and **Prefix Rewriting Systems**

Jensen et al. **P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures**. 14th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2018.

AalWiNes Tool

Query: regular expression

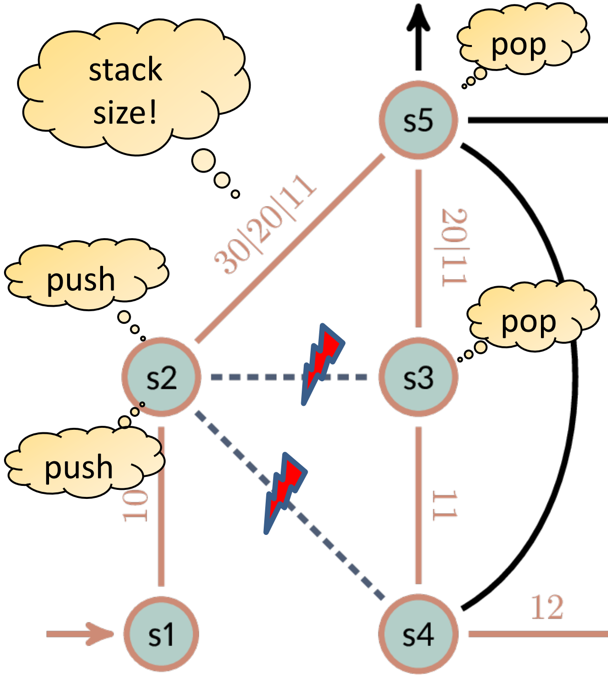
Witness

Dozens of networks

Online demo: https://demo.aalwines.cs.aau.dk/
Source code: https://github.com/DEIS-Tools/AalWiNes

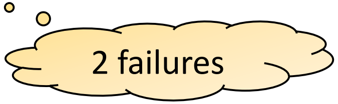
Example

Can traffic starting with [] go through s5, under up to k=2 failures?



Query: 3 regular expressions (initial and final header, route)

k=2 [] s1 >> s5 >> s7 []



YES

(Polynomial time!)

Why AalWiNes is Fast (Polytime): Automata Theory

- For fast verification, we can use the result by **Büchi**: the set of all reachable configurations of a pushdown automaton a is regular set
- We hence simply use Nondeterministic Finite Automata (NFAs) when reasoning about the pushdown automata
- The resulting **regular operations** are all polynomial time

Julius Richard Büchi

1924-1984

Swiss logician

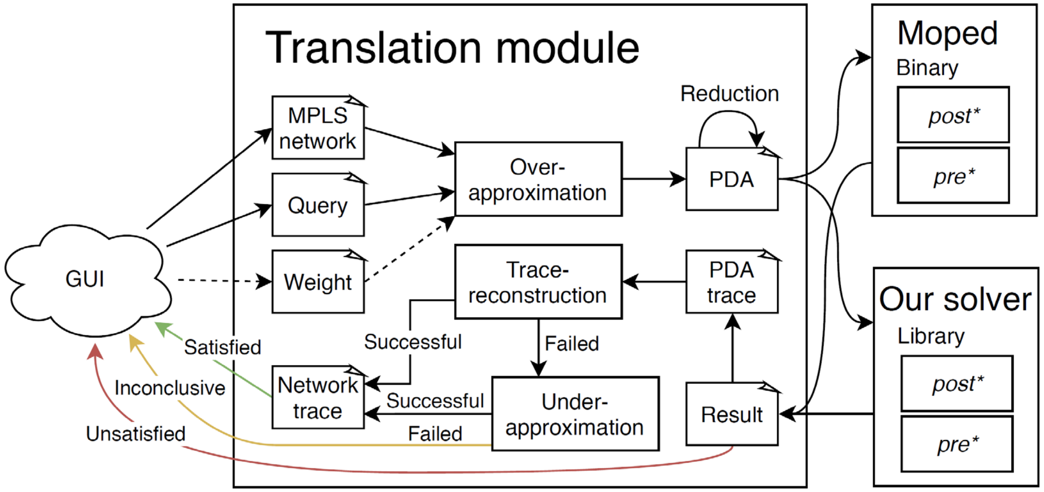
AalWiNes

Part 1: Parses query and constructs Push-Down System (PDS)

- In Python 3

Part 2: Reachability analysis of constructed PDS

- Using ***Moped*** tool



Resp. our new weighted extension and much faster implementation in C++.

Network Model

- Network: a 7-tuple

Links

Outgoing interfaces

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Nodes

Incoming interfaces

Set of labels in packet header

Network Model

- Network: a 7-tuple

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Interface function

Interface function: maps outgoing interface to next hop node and incoming interface to previous hop node

$$\lambda_v : I_v^{in} \cup I_v^{out} \to V$$

That is: $(\lambda_v(in), v) \in E$ and $(v, \lambda_v(out)) \in E$

Network Model

- Network: a 7-tuple

$$N = (V, E, I_v^{in}, I_v^{out}, \lambda_v, L, \delta_v^F)$$

Routing function

Routing function: for each set of failed links $F \subseteq E$, the routing function

$$\delta_v^F : I_v^{in} \times L^* \to 2^{(I^{out} \times L^*)}$$

defines, for all incoming interfaces and packet headers, outgoing interfaces together with modified headers.

Routing

Packet routing sequence can be represented using **sequence of tuples**:

... on interface...

... forwards it to live next hop...

... given that these links are down.

$$(v_i, in_i, h_i, out_i, h_{i+1}, F_i)$$

Node receives...

... packet with header...

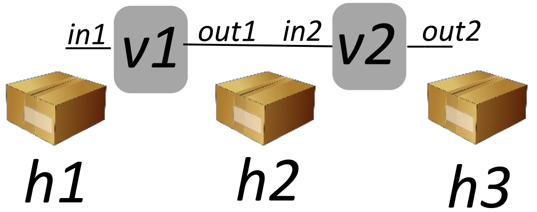
... with new header..

- Example: **routing** (in)finite sequence of tuples

$$(v_1, in_1, h_1, out_1, h_2, F_1),$$

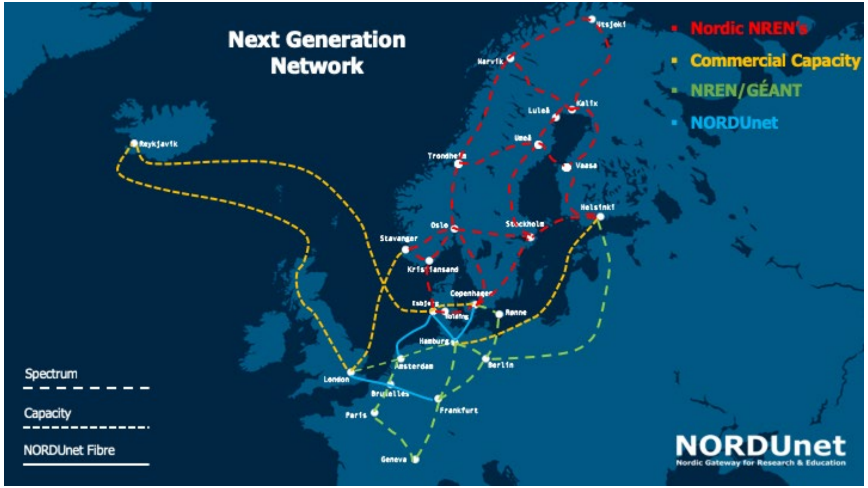
$$(v_2, in_2, h_2, out_2, h_3, F_2),$$

$$\ldots$$



Case Study: NORDUnet

- Regional service provider
- **24 MPLS routers** geographically distributed across several countries
- Running **Juniper** operating system
- More than 30,000 labels
- Ca. **1 million** forwarding rules in our model
- For most queries of operators: answer *within seconds*



Generalizes to Quantitative Properties

- AalWiNes can also be used to test **quantitative properties**

- If query is satisfied, find trace that minimizes:
  - **Hops**
  - Latency (based on a latency value per link)
  - Tunnels

Transitions annotated
with weights.

- Approach: **weighted** pushdown automata
  - Fast *poly-time algorithms* exist also for weighted pushdown automata (area of dataflow analysis)
  - Indeed, experiments show: *acceptable overhead* of weighted (quantitative) analysis

Conclusion

- Fast rerouting requires *local decision making*
- Different fault-tolerance *metrics*: ideal resilience, perfect resilience
- What can be achieved depends on *what can be matched* locally
- Locally *balancing load* under failures is hard, but randomization helps

What About The Control Plane?

Still many open questions too, see e.g., *TACAS 2021*

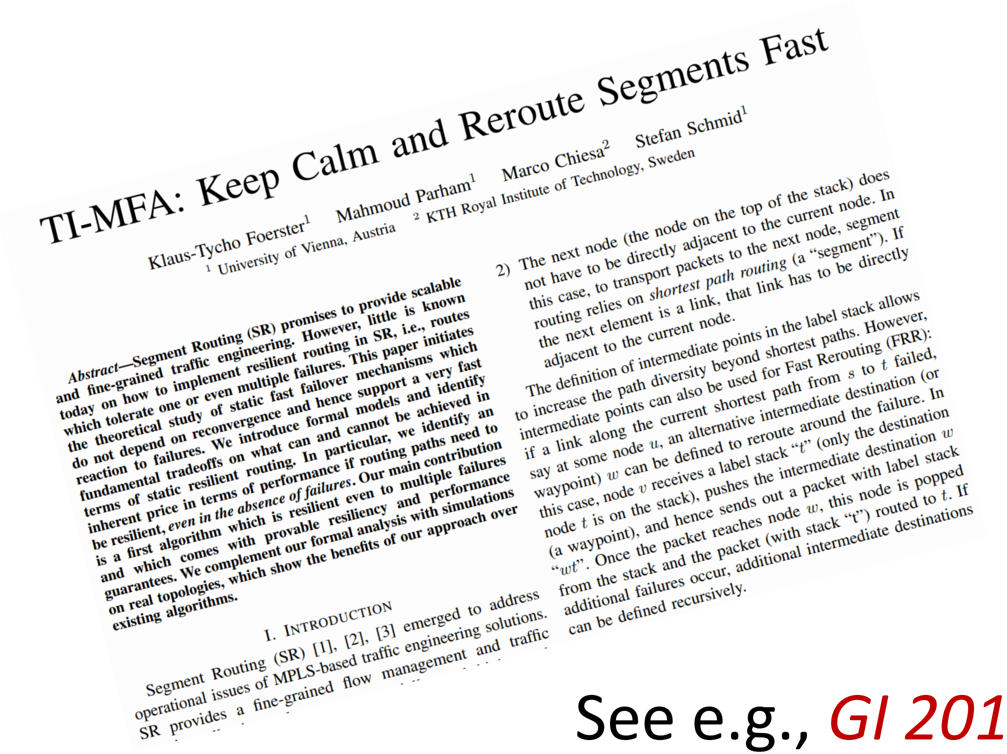
Resilient Capacity-Aware Routing

Stefan Schmid[1], Nicolas Schnepf[2], and Jiří Srba[2]

[1] Faculty of Computer Science, University of Vienna
[2] Department of Computer Science, Aalborg University

Abstract. To ensure a high availability, most modern communication networks provide resilient routing mechanisms that quickly change routes upon failures. However, a fundamental algorithmic question underlying such mechanisms is hardly understood: how to efficiently verify whether a given network reroutes flows along *feasible* paths, without violating capacity constraints, for up to k link failures? We chart the algorithmic complexity landscape of resilient routing under link failures, considering shortest path routing based on link weights (e.g., the widely deployed ECMP protocol). We study two models: a *pessimistic* model where flows interfere in a worst-case manner along equal-cost shortest paths, and an *optimistic* model where flows are routed in a best-case manner and we present a complete picture of the algorithmic complexities for these models. We further propose a strategic search algorithm that checks only the critical failure scenarios while still providing correctness guarantees. Our experimental evaluation on a large benchmark of Internet and datacenter topologies confirms an improved performance of our strategic search algorithm by several orders of magnitude.

What About Segment Routing?



TI-MFA: Keep Calm and Reroute Segments Fast

Klaus-Tycho Foerster[1] Mahmoud Parham[1] Marco Chiesa[2] Stefan Schmid[1]
[1] University of Vienna, Austria [2] KTH Royal Institute of Technology, Sweden

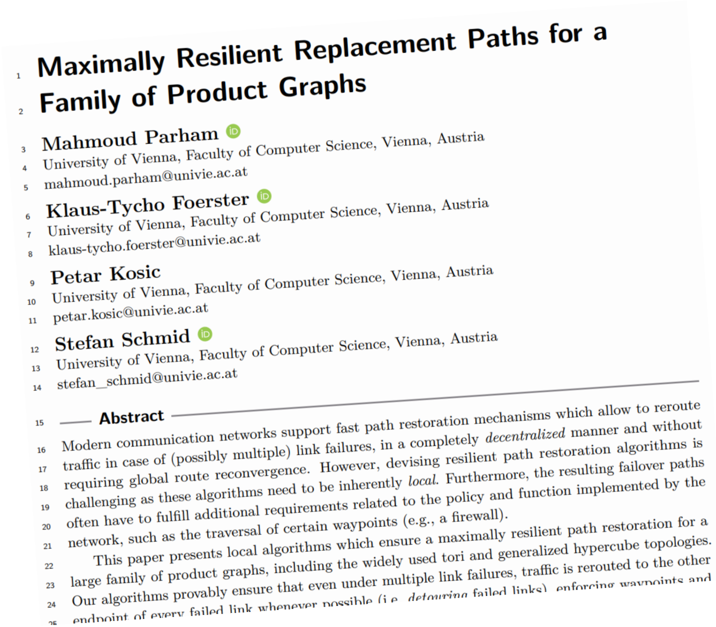
***Abstract*—Segment Routing (SR) promises to provide scalable and fine-grained traffic engineering. However, little is known today on how to implement resilient routing in SR, i.e., routes which tolerate one or even multiple failures. This paper initiates the theoretical study of static fast failover mechanisms which do not depend on reconvergence and hence support a very fast reaction to failures. We introduce formal models and identify fundamental tradeoffs on what can and cannot be achieved in terms of static resilient routing. In particular, we identify an inherent price in terms of performance if routing paths need to be resilient, *even in the absence of failures*. Our main contribution is a first algorithm which is resilient even to multiple failures and which comes with provable resiliency and performance guarantees. We complement our formal analysis with simulations on real topologies, which show the benefits of our approach over existing algorithms.**

I. Introduction

Segment Routing (SR) [1], [2], [3] emerged to address operational issues of MPLS-based traffic engineering solutions. SR provides a fine-grained flow management and traffic

2) The next node (the node on the top of the stack) does not have to be directly adjacent to the current node. In this case, to transport packets to the next node, segment routing relies on *shortest path routing* (a "segment"). If the next element is a link, that link has to be directly adjacent to the current node.

The definition of intermediate points in the label stack allows to increase the path diversity beyond shortest paths. However, intermediate points can also be used for Fast Rerouting (FRR): if a link along the current shortest path from s to t failed, say at some node u, an alternative intermediate destination (or waypoint) w can be defined to reroute around the failure. In this case, node v receives a label stack "t" (only the destination node t is on the stack), pushes the intermediate destination w (a waypoint), and hence sends out a packet with label stack "wt". Once the packet reaches node w, this node is popped from the stack and the packet (with stack "t") routed to t. If additional failures occur, additional intermediate destinations can be defined recursively.



Maximally Resilient Replacement Paths for a Family of Product Graphs

Mahmoud Parham
University of Vienna, Faculty of Computer Science, Vienna, Austria
mahmoud.parham@univie.ac.at

Klaus-Tycho Foerster
University of Vienna, Faculty of Computer Science, Vienna, Austria
klaus-tycho.foerster@univie.ac.at

Petar Kosic
University of Vienna, Faculty of Computer Science, Vienna, Austria
petar.kosic@univie.ac.at

Stefan Schmid
University of Vienna, Faculty of Computer Science, Vienna, Austria
stefan_schmid@univie.ac.at

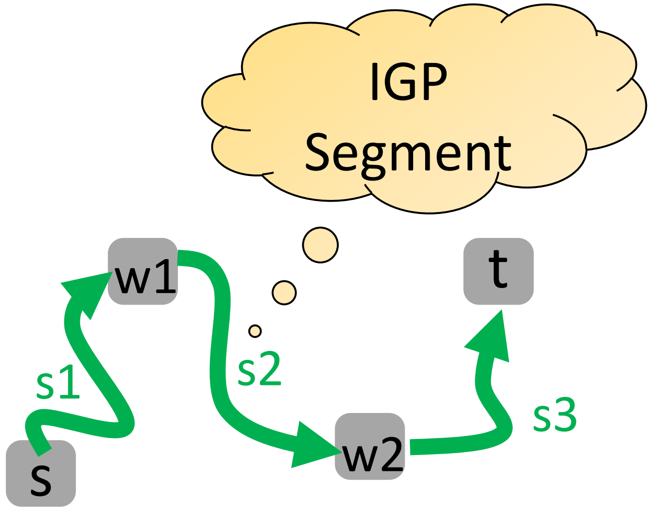
Abstract

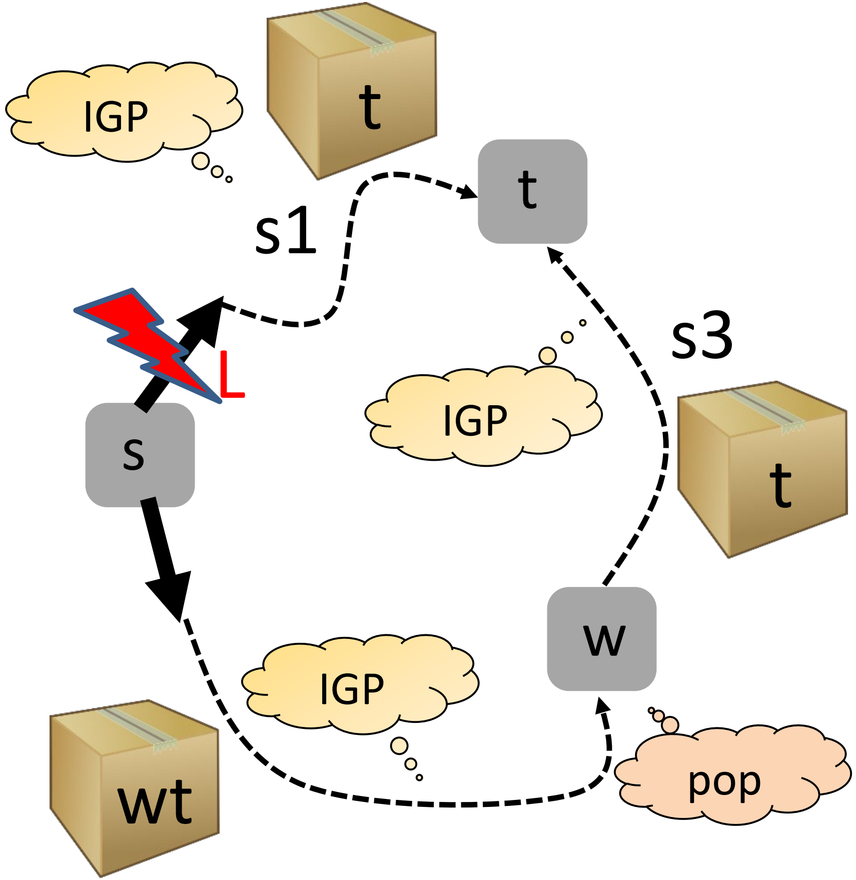
Modern communication networks support fast path restoration mechanisms which allow to reroute traffic in case of (possibly multiple) link failures, in a completely *decentralized* manner and without requiring global route reconvergence. However, devising resilient path restoration algorithms is challenging as these algorithms need to be inherently *local*. Furthermore, the resulting failover paths often have to fulfill additional requirements related to the policy and function implemented by the network, such as the traversal of certain waypoints (e.g., a firewall).

This paper presents local algorithms which ensure a maximally resilient path restoration for a large family of product graphs, including the widely used tori and generalized hypercube topologies. Our algorithms provably ensure that even under multiple link failures, traffic is rerouted to the other endpoint of every failed link whenever possible (i.e. *detouring* failed links), enforcing waypoints and

See e.g., *GI 2018*
and *OPODIS 2020*

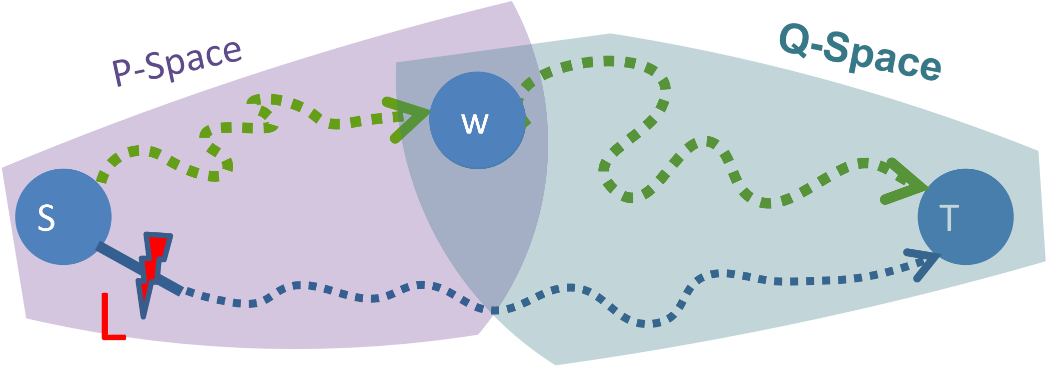
What About Segment Routing?





What About Segment Routing?

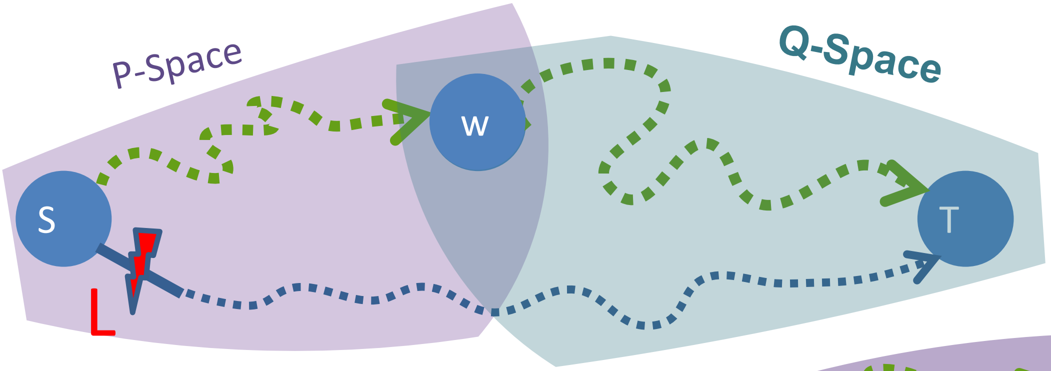
- We need two definitions:
  - **P-Space**: the nodes whose shortest path from S does not use L
  - **Q-Space**: the nodes whose shortest path to T does not use L



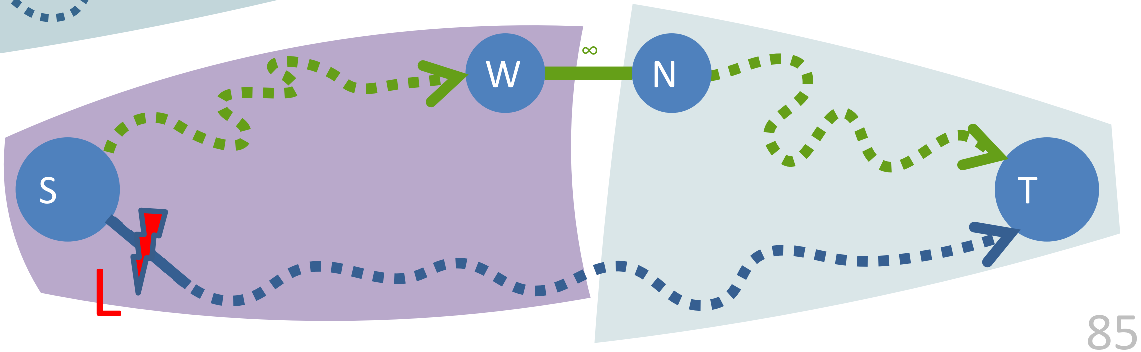
Idea: choose segment endpoint w at intersection!

Two Cases

P-Space and Q-Space: Are **connected** subgraphs, **cover** all nodes, overlap or are adjacent

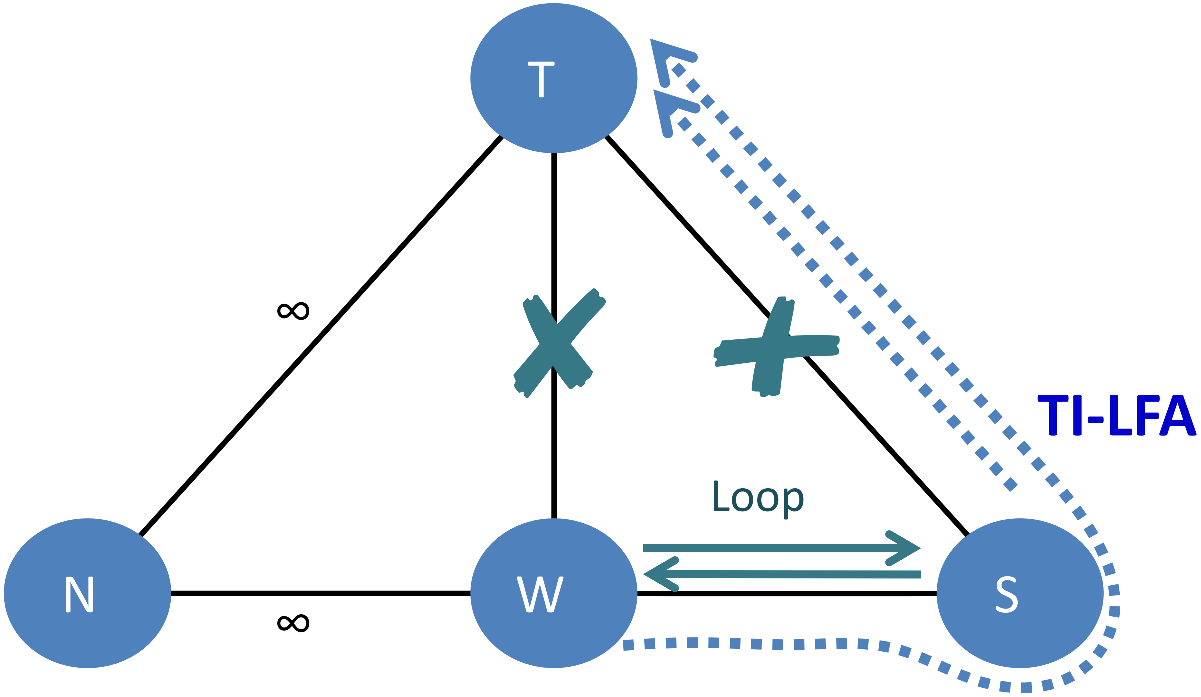


Case 1: S can simply push W

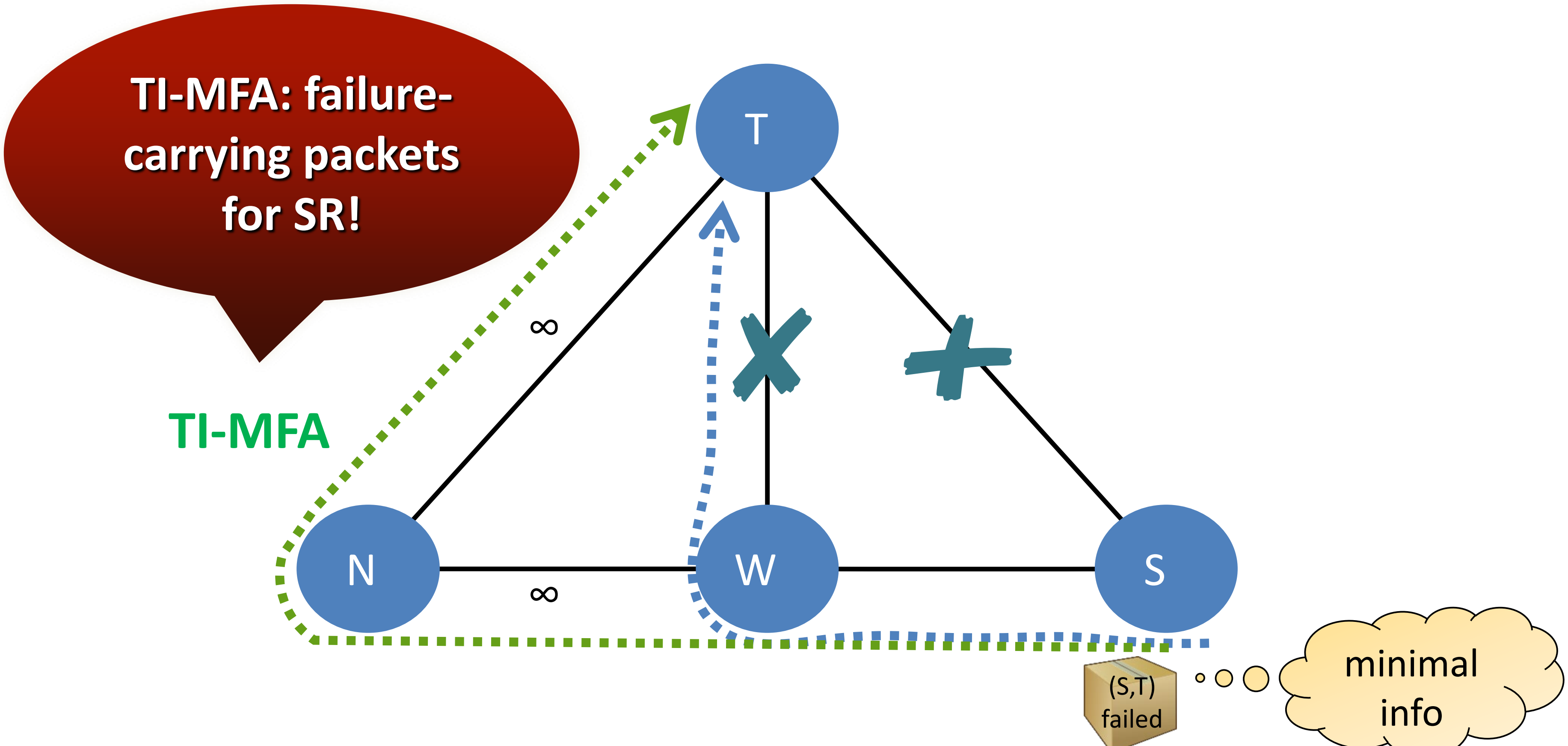


Case 2: S pushes W and (W,N), forces packet to enter Q-space

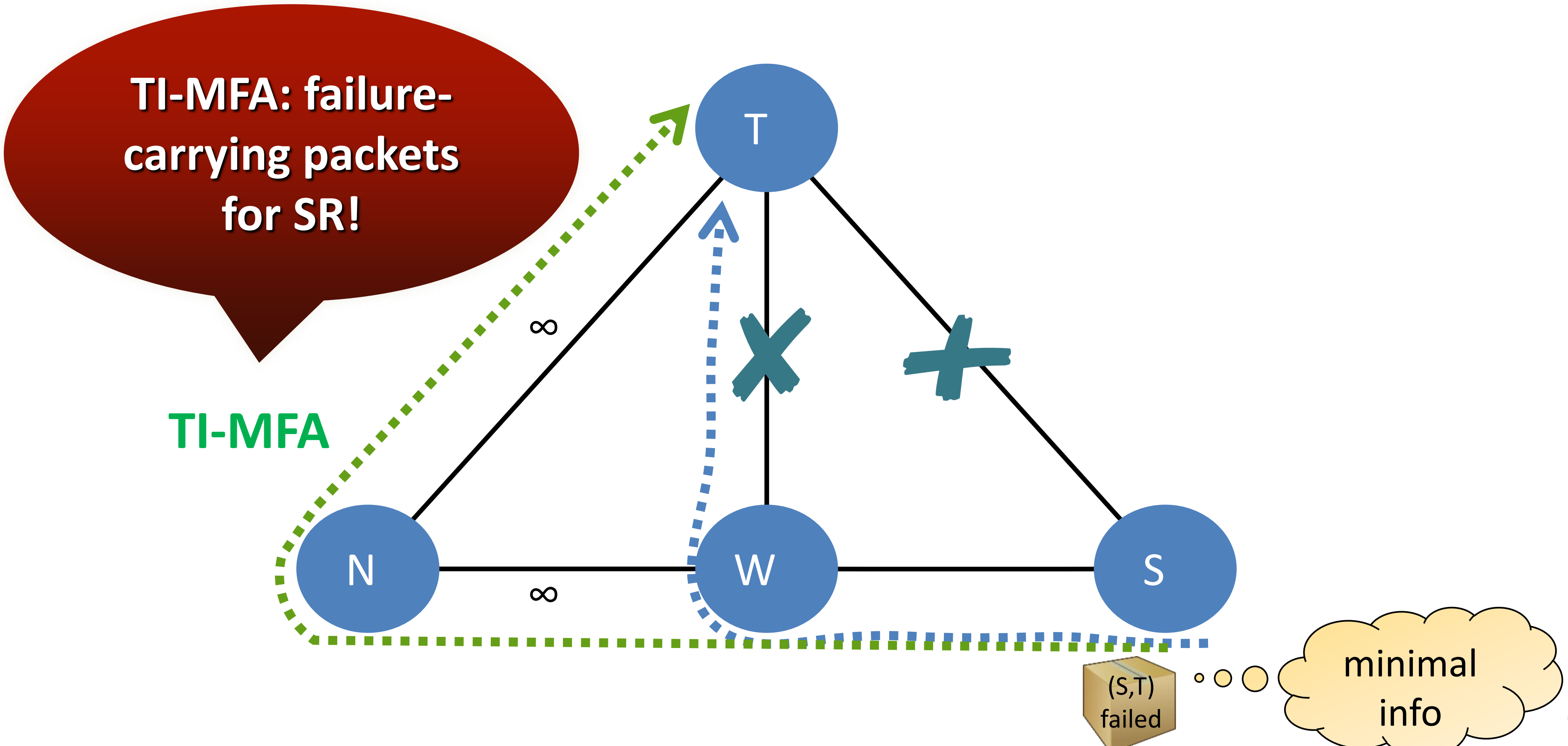
TI-LFA Under Double Failure



TI-MFA Under Double Failure



TI-MFA Under Double Failure



From the viewpoint of the node *S* where the packet hits another failed link:

1. **Flush** the label stack except for the destination *T*

2. Based on all **link failure info** stored in the packet header, compute the segments necessary to reach *T* and the labels accordingly

3. Find the last node on *ShortestPath(S,T)* that a packet can reach from *S* without hitting known failed link (**"repeated TI-LFA on subgraph"**)

   a. Let ***V1*** be this node followed by the link ***(V1,V2)*** on this path

   b. Set the top of **label stack** as *(V1, (V1,V2),…*

   c. Repeat the same for *V2* as the start of next segment and keep repeating until the segment that ends with *T*

4. **Dispatch** the packet (it will reach *T* unless it hits a failure disconnecting the network)

Theorem: TI-MFA tolerates k failures in k-connected network!

Proof:

- **Invariant:** by construction, previously hit failures won't be hit again
- *k* failures: by construction the backup path will not use any failed link **seen previously**
- Hence, the packet either hits all the *k* failures or reaches its destination early

Efficient Implementation of FRR?

See e.g.,

CoNEXT 2019

PURR: A Primitive for Reconfigurable Fast Reroute

(hope for the best and program for the worst)

Marco Chiesa
KTH Royal Institute of Technology

Roshan Sedar
Universitat Politècnica de Catalunya

Gianni Antichi
Queen Mary University of London

Michael Borokhovich
Independent Researcher

Andrzej Kamisiński
AGH University of Science and Technology in Kraków

Georgios Nikolaidis
Barefoot Networks

Stefan Schmid
Faculty of Computer Science
University of Vienna

ABSTRACT

Highly dependable communication networks usually rely on some kind of Fast Re-Route (FRR) mechanism which allows to quickly re-route traffic upon failures, entirely in the data plane. This paper studies the design of FRR mechanisms for emerging reconfigurable switches.

Our main contribution is an FRR primitive for *programmable* data planes, PURR, which provides low failover latency and high switch throughput, by *avoiding packet recirculation*. PURR tolerates multiple concurrent failures and comes with minimal memory requirements, ensuring *compact* forwarding tables, by unveiling an intriguing connection to classic "string theory" (*i.e.*, stringology), and in particular, the shortest common supersequence problem. PURR is well-suited for high-speed match-action forwarding architectures (e.g., PISA) and supports the implementation of arbitrary network-wide FRR mechanisms. Our simulations and prototype implementation (on an FPGA and Tofino) show that PURR improves

ACM Reference Format:
Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. 2019. PURR: A Primitive for Reconfigurable Fast Reroute: (hope for the best and program for the worst). In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359989.3365410

1 INTRODUCTION

Emerging applications, e.g., in the context of business [21] and entertainment [57], pose stringent requirements on the dependability and performance of the underlying communication networks, which have become a critical infrastructure of our digital society. In order to meet such requirements, many communication networks provide *Fast Re-Route* (FRR) mechanisms [5, 39, 64] which allow to quickly reroute traffic upon unexpected failures, entirely in the

A Recent Survey

A Survey of Fast-Recovery Mechanisms in Packet-Switched Networks
Marco Chiesa, Andrzej Kamisinski, Jacek Rak, Gabor Retvari, and Stefan Schmid.
IEEE Communications Surveys and Tutorials (**COMST**), 2021.

References

[On the Feasibility of Perfect Resilience with Local Fast Failover]
Klaus-Tycho Foerster, Juho Hirvonen, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan.
SIAM Symposium on Algorithmic Principles of Computer Systems (**APOCS**), Alexandria, Virginia, USA, January 2021.

[Brief Announcement: What Can(not) Be Perfectly Rerouted Locally]
Klaus-Tycho Foerster, Juho Hirvonen, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan.
International Symposium on Distributed Computing (**DISC**), Freiburg, Germany, October 2020.

[Improved Fast Rerouting Using Postprocessing]
Klaus-Tycho Foerster, Andrzej Kamisinski, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan.
IEEE Transactions on Dependable and Secure Computing (**TDSC**), 2020.

[Resilient Capacity-Aware Routing]
Stefan Schmid, Nicolas Schnepf and Jiri Srba.
27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (**TACAS**), Virtual Conference, March 2021.

[AalWiNes: A Fast and Quantitative What-If Analysis Tool for MPLS Networks]
Peter Gjøl Jensen, Morten Konggaard, Dan Kristiansen, Stefan Schmid, Bernhard Clemens Schrenk, and Jiri Srba.
16th ACM International Conference on emerging Networking EXperiments and Technologies (**CoNEXT**), Barcelona, Spain, December 2020.

[P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures]
Jesper Stenbjerg Jensen, Troels Beck Krogh, Jonas Sand Madsen, Stefan Schmid, Jiri Srba, and Marc Tom Thorgersen.
14th ACM International Conference on emerging Networking EXperiments and Technologies (**CoNEXT**), Heraklion/Crete, Greece, December 2018.

[Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks]
Stefan Schmid and Jiri Srba.
37th IEEE Conference on Computer Communications (**INFOCOM**), Honolulu, Hawaii, USA, April 2018.

More References

Randomized Local Fast Rerouting for Datacenter Networks with Almost Optimal Congestion
Gregor Bankhamer, Robert Elsässer, and Stefan Schmid..
International Symposium on Distributed Computing (**DISC**), Freiburg, Germany, October 2021.

Bonsai: Efficient Fast Failover Routing Using Small Arborescences
Klaus-Tycho Foerster, Andrzej Kamisinski, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan.
49th IEEE/IFIP International Conference on Dependable Systems and Networks (**DSN**), Portland, Oregon, USA, June 2019.

CASA: Congestion and Stretch Aware Static Fast Rerouting
Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan
38th IEEE Conference on Computer Communications (**INFOCOM**), Paris, France, April 2019.

Load-Optimal Local Fast Rerouting for Dense Networks
Michael Borokhovich, Yvonne-Anne Pignolet, Gilles Tredan, and Stefan Schmid.
IEEE/ACM Transactions on Networking (**TON**), 2018.

PURR: A Primitive for Reconfigurable Fast Reroute
Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisinski, Georgios Nikolaidis, and Stefan Schmid.
15th ACM International Conference on emerging Networking EXperiments and Technologies (**CoNEXT**), Orlando, Florida, USA, December 2019.
Artefact Evaluation: Available, Functional, Reusable.

On the Resiliency of Static Forwarding Tables
In IEEE/ACM Transactions on Networking (**ToN**), 2017
M. Chiesa, I. Nikolaevskiy, S. Mitrovic, A. Gurtov, A. Madry, M. Schapira, S. Shenker


Questions?

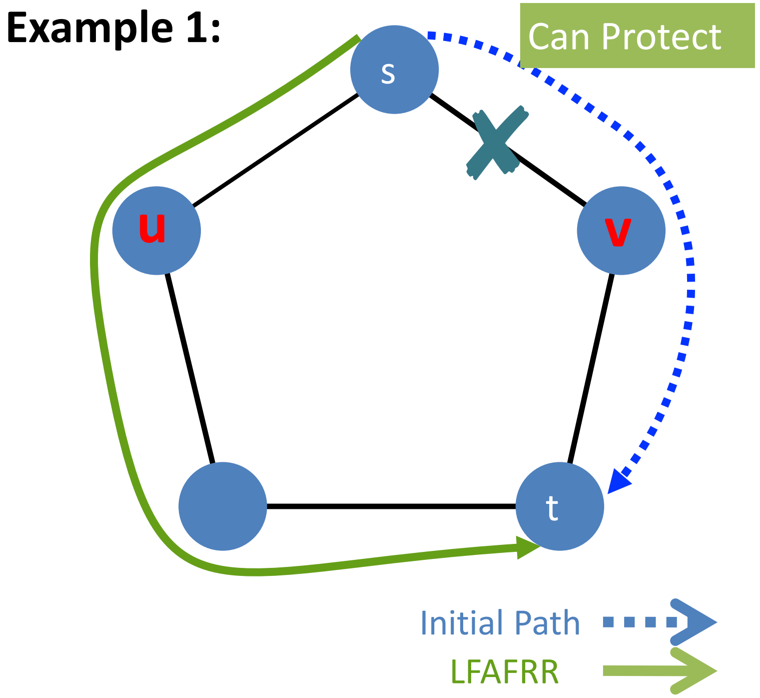
Backup Slides

Remark: Traditional Approach LFA

- Traditionally: forwarding along **shortest paths**
- **Loop-Free Alternative (LFA):** failover to alternative neighbor, from there shortest path

Example 1:

- If *(s,v)* fails, s can *failover to u*
- *u* has shortest path to *t* that does not go through (*s*,*v*) again
- *WORKS*: can protect (*s*,*v*)

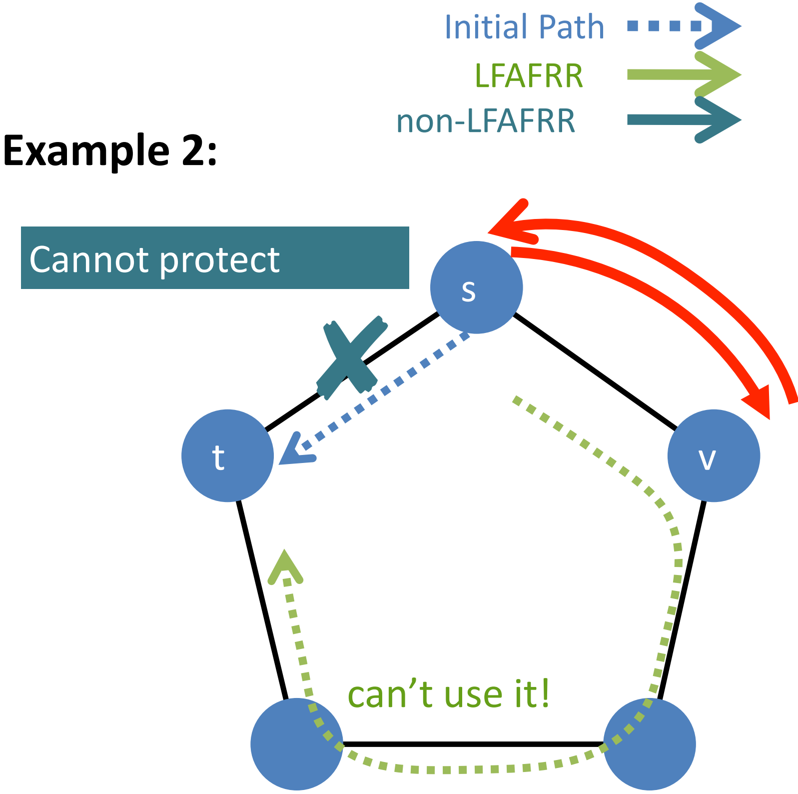


Remark: Traditional Approach LFA

- Traditionally: forwarding along **shortest paths**
- **Loop-Free Alternative (LFA):** failover to alternative neighbor, from there shortest path

Example 2:

- If (s,t) fails, s can only try to failover to v
- However, when v's shortest route to t goes along s again: loop
- *DOES NOT WORK*: Cannot protect (s,t)

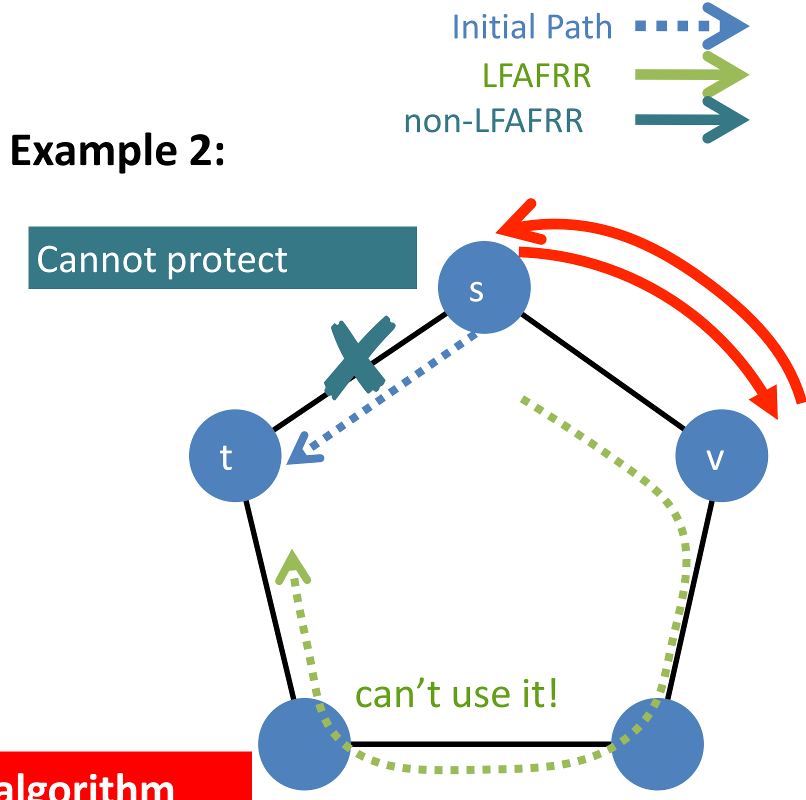


Remark: Traditional Approach LFA

- Traditionally: forwarding along **shortest paths**
- **Loop-Free Alternative (LFA):** failover to alternative neighbor, from there shortest path

Example 2:

- If (s,t) fails, s can only try to failover to v
- However, when v's shortest route to t goes along s again: loop
- *DOES NOT WORK*: Cannot protect (s,t)



Even though loop-free alternative path exists, an LFA algorithm cannot use it. Protection ratio of LFA depends on topology.