The Art of Transiently Consistent Route Updates

Stefan Schmid

Aalborg University

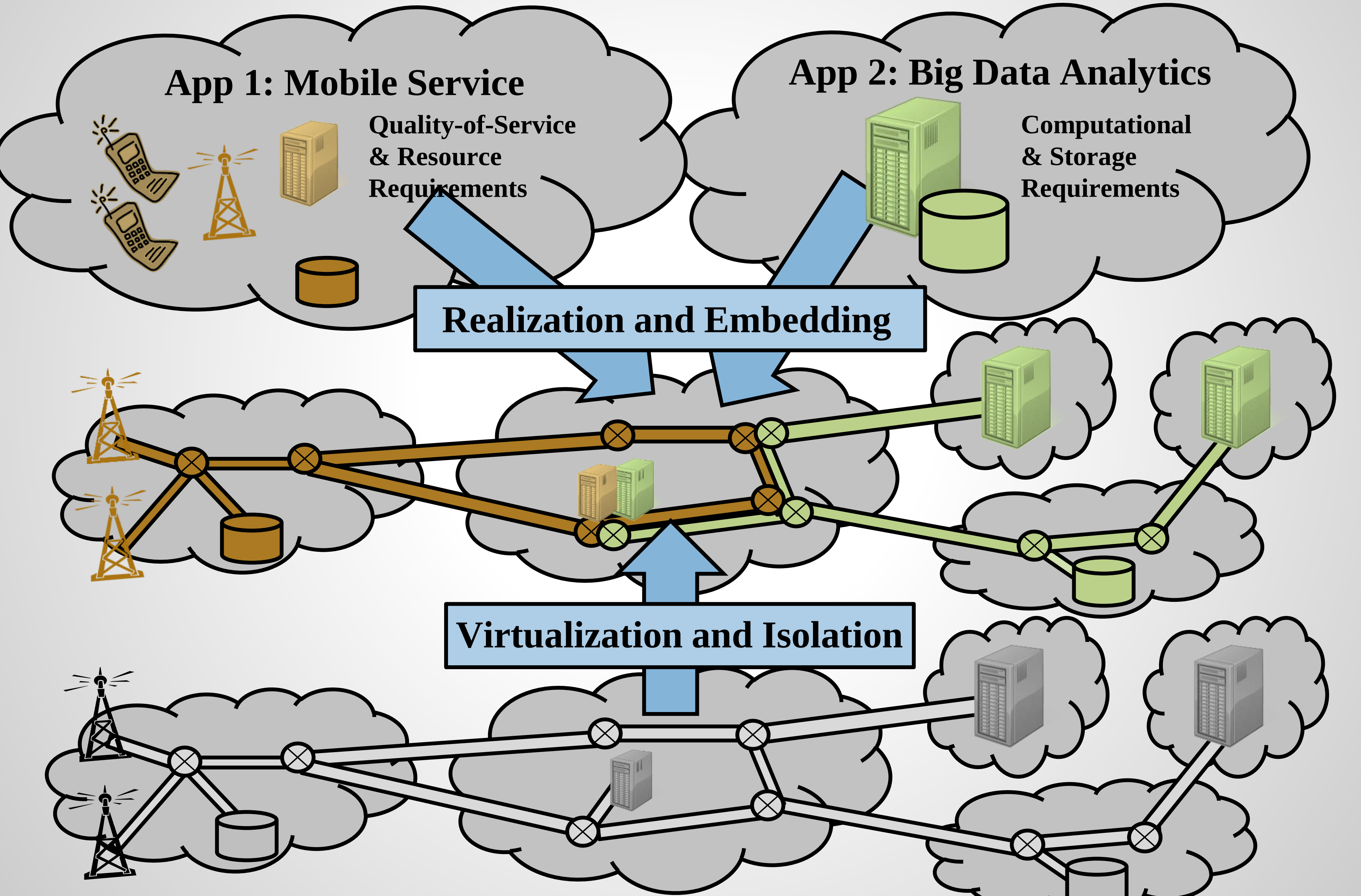
Joint work mainly with:
Arne Ludwig, Jan Marcinkowski,
Szymon Dudycz, Matthias Rost

Modern Networked Systems: Programmable and Virtualized

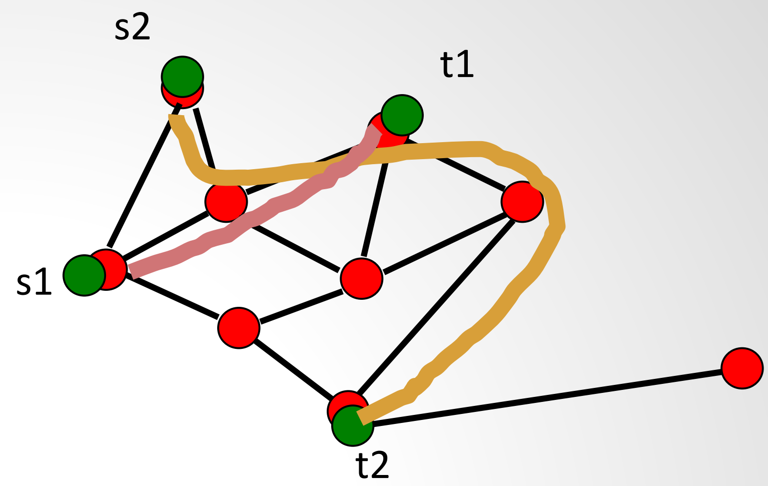
New flexibilities but also challenges: Great time to be a scientist! ☺

"We are at an interesting inflection point!"
Keynote by George Varghese
at SIGCOMM 2014

Challenge 1: Predictable Performance with Resource Sharing = Multi-Dimensional Performance Isolation

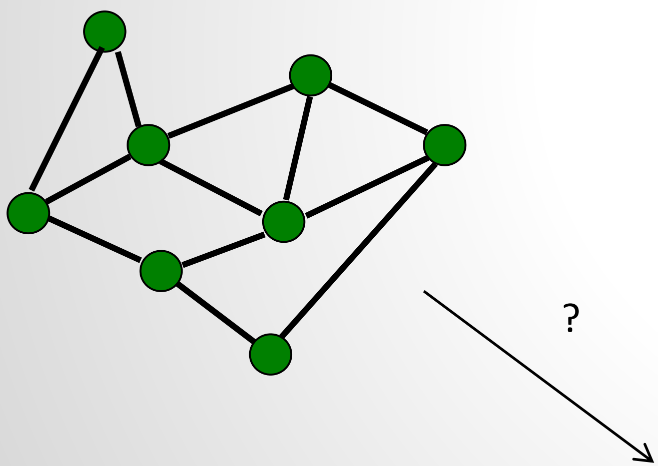


Challenge 2: Exploiting Allocation Flexibilities Non-Trivial



Start simple: exploit flexible routing between given tasks/VMs

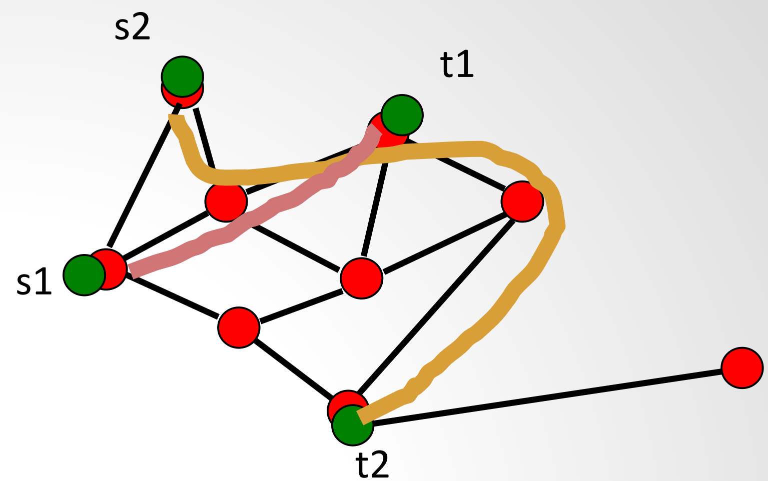
- Integer multi-commodity flow problem with 2 flows?
- Oops: NP-hard



Forget about paths: exploit tasks/VM placement flexibilities!

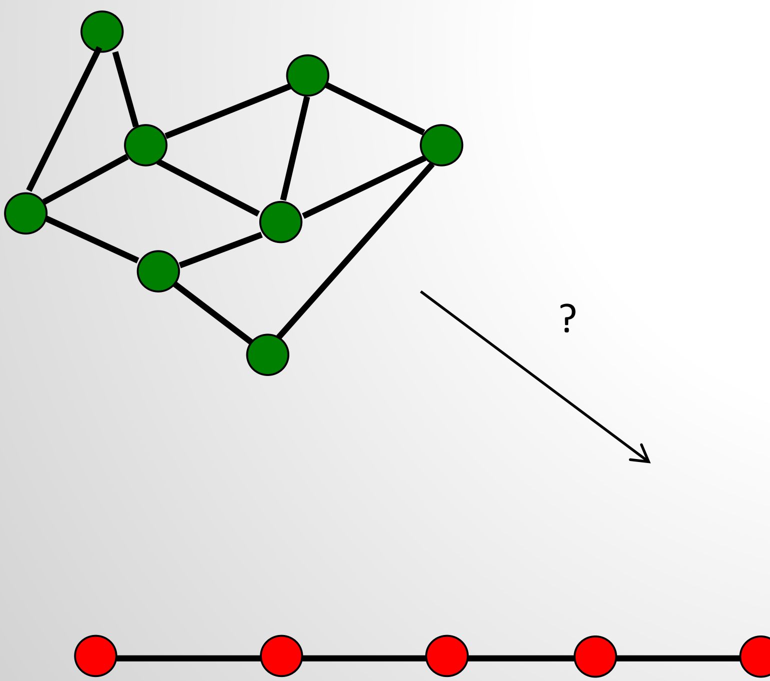
- Most simple: Minimum Linear Arrangement without capacities
- NP-hard ☹

Challenge 2: Exploiting Allocation Flexibilities Non-Trivial



Start simple: exploit flexible routing between given tasks/VMs

- Integer multi-commodity flow problem with 2 flows?
- Oops: NP-hard



Forget about paths: exploit tasks/VM placement flexibilities!

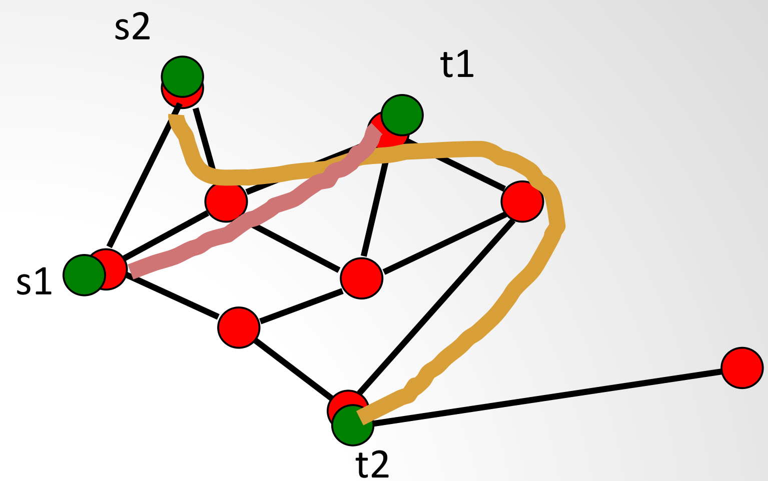
- Most simple: Minimum Linear Arrangement without capacities
- NP-hard ☹

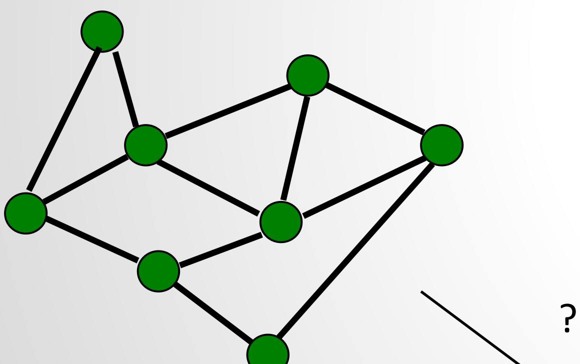
[Beyond the Stars: Revisiting Virtual Cluster Embeddings]
Matthias Rost, Carlo Fuerst, and Stefan Schmid.
ACM SIGCOMM Computer Communication Review (**CCR**), July 2015..

Challenge 2: Exploiting Allocation Flexibilities Non-Trivial

Start simple: exploit flexible routing between given tasks/VMs

- Integer multi-commodity flow problem with 2 flows?
- Oops: NP-hard





Not only on Clos, but e.g., also on Ankit's Jellyfish etc.!

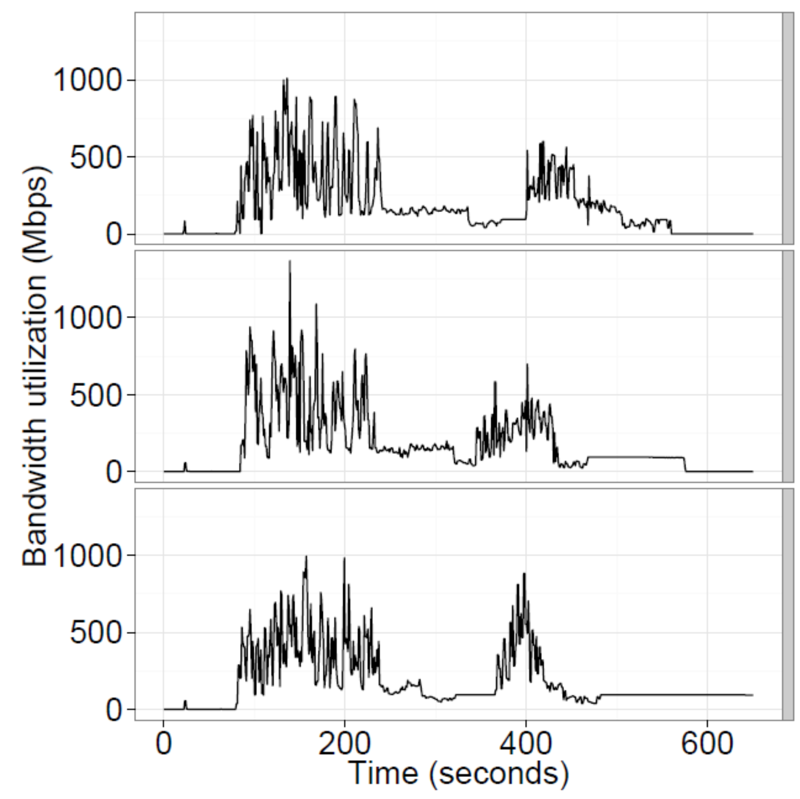
Forget about paths: exploit tasks/VM placement flexibilities!

- Most simple: Minimum Linear Arrangement without capacities
- NP-hard ☹

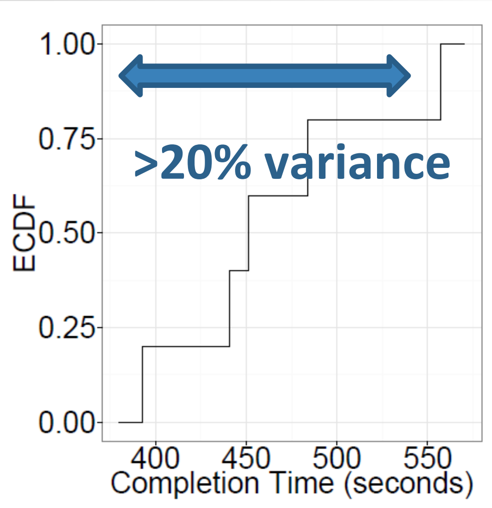
Beyond the Stars: Revisiting Virtual Cluster Embeddings
Matthias Rost, Carlo Fuerst, and Stefan Schmid.
ACM SIGCOMM Computer Communication Review (**CCR**), July 2015..

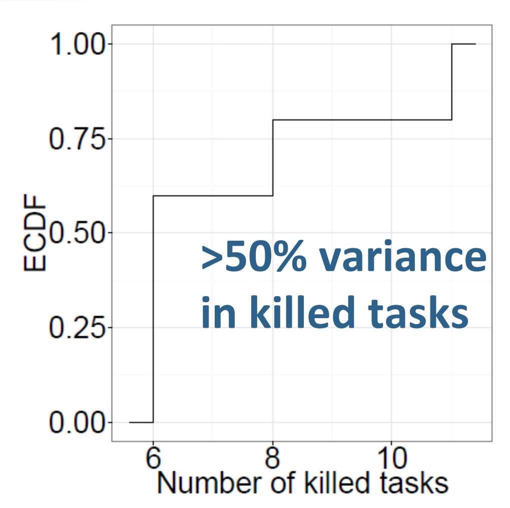
Challenge 3: Dealing with Uncertainty

- Hadoop and scale-out data bases generate much network traffic
- **Temporal** resource patterns are hard to predict
- Resource allocations must be changed **online**
- **Tradeoffs:**
  - **overprovisiong vs efficiency**
  - **benefit vs cost of reconfigurations!**



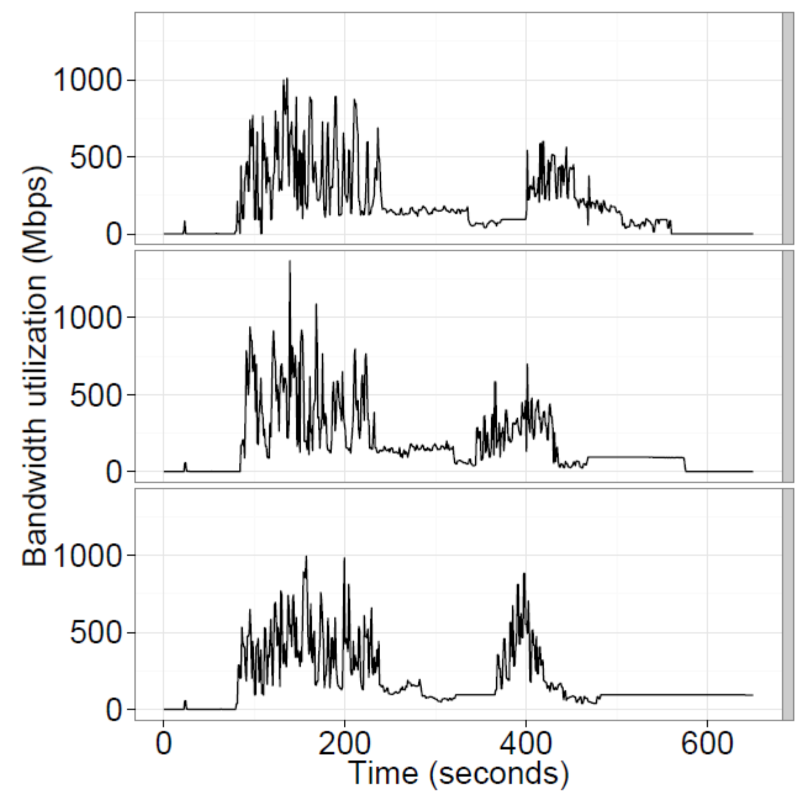
Bandwidth utilization of 3 different runs of the same **TeraSort workload** (**without interference**)



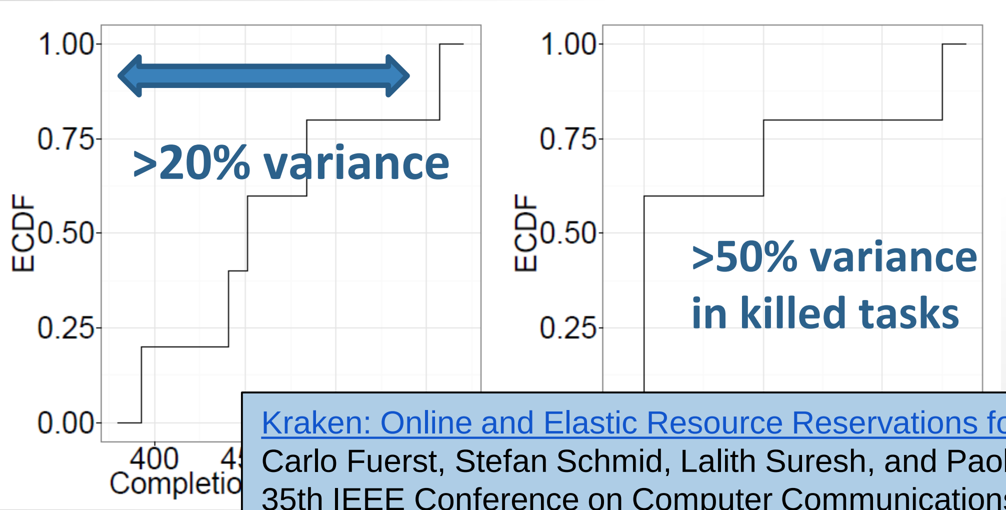


Challenge 3: Dealing with Uncertainty

- Hadoop and scale-out data bases generate much network traffic
- **Temporal** resource patterns are hard to predict
- Resource allocations must be changed **online**
- **Tradeoffs:**
  - **overprovisiong vs efficiency**
  - **benefit vs cost of reconfigurations!**



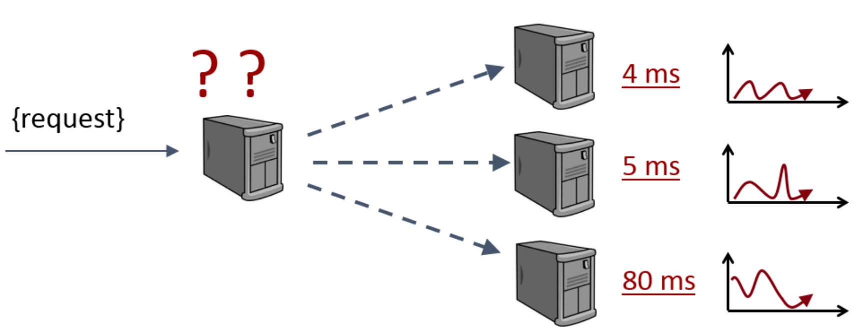
Bandwidth utilization of 3 different runs of the same **TeraSort workload** (**without interference**)

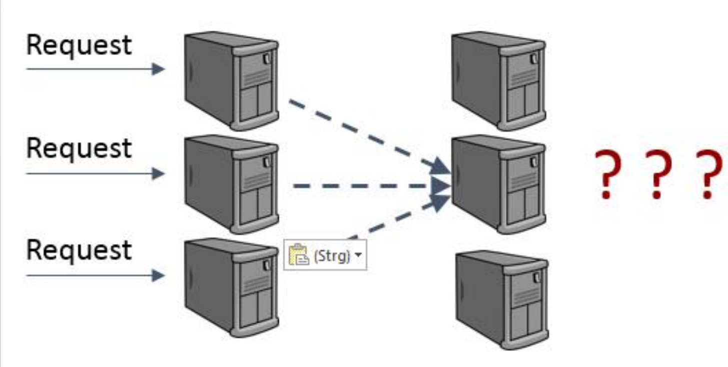


[Kraken: Online and Elastic Resource Reservations for Multi-tenant Datacenters](#)
Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa.
35th IEEE Conference on Computer Communications (**INFOCOM**), San Francisco, California, USA, April 2016.

Challenge 4: Exploiting Redundancy/Selection Flexibilities Non-Trivial

- Replica selection possible in **cloud data stores** (e.g., Cassandra)
- Idea: reduce tail latency
  - Tail matters: requests have many read/writes, a single late one can delay!
  - Stragglers even in well-provisioned systems
- Challenge 1: Heterogeneous and time-varying service times
  - shared resources, log compaction, garbage collection, daemons, etc.

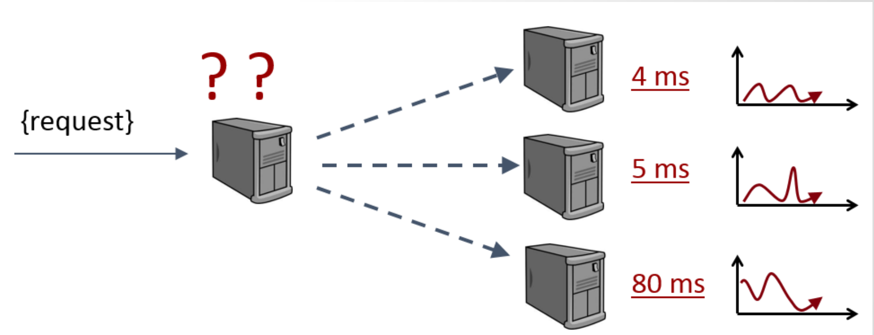


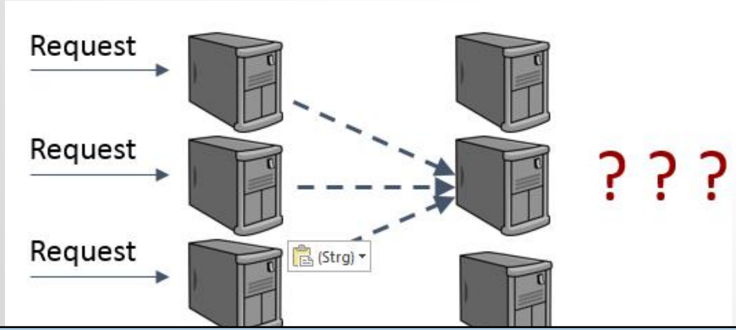


- Challenge 2: Distributed coordination
  - avoid herd-behavior!
  - also a control-theoretic problem

Challenge 4: Exploiting Redundancy/Selection Flexibilities Non-Trivial

- Replica selection possible in **cloud data stores** (e.g., Cassandra)
- Idea: reduce tail latency
  - Tail matters: requests have many read/writes, a single late one can delay!
  - Stragglers even in well-provisioned systems
- Challenge 1: Heterogeneous and time-varying service times
  - shared resources, log compaction, garbage collection, deamons, etc.





- Challenge 2: Distributed coordination
  - avoid herd-behavior!

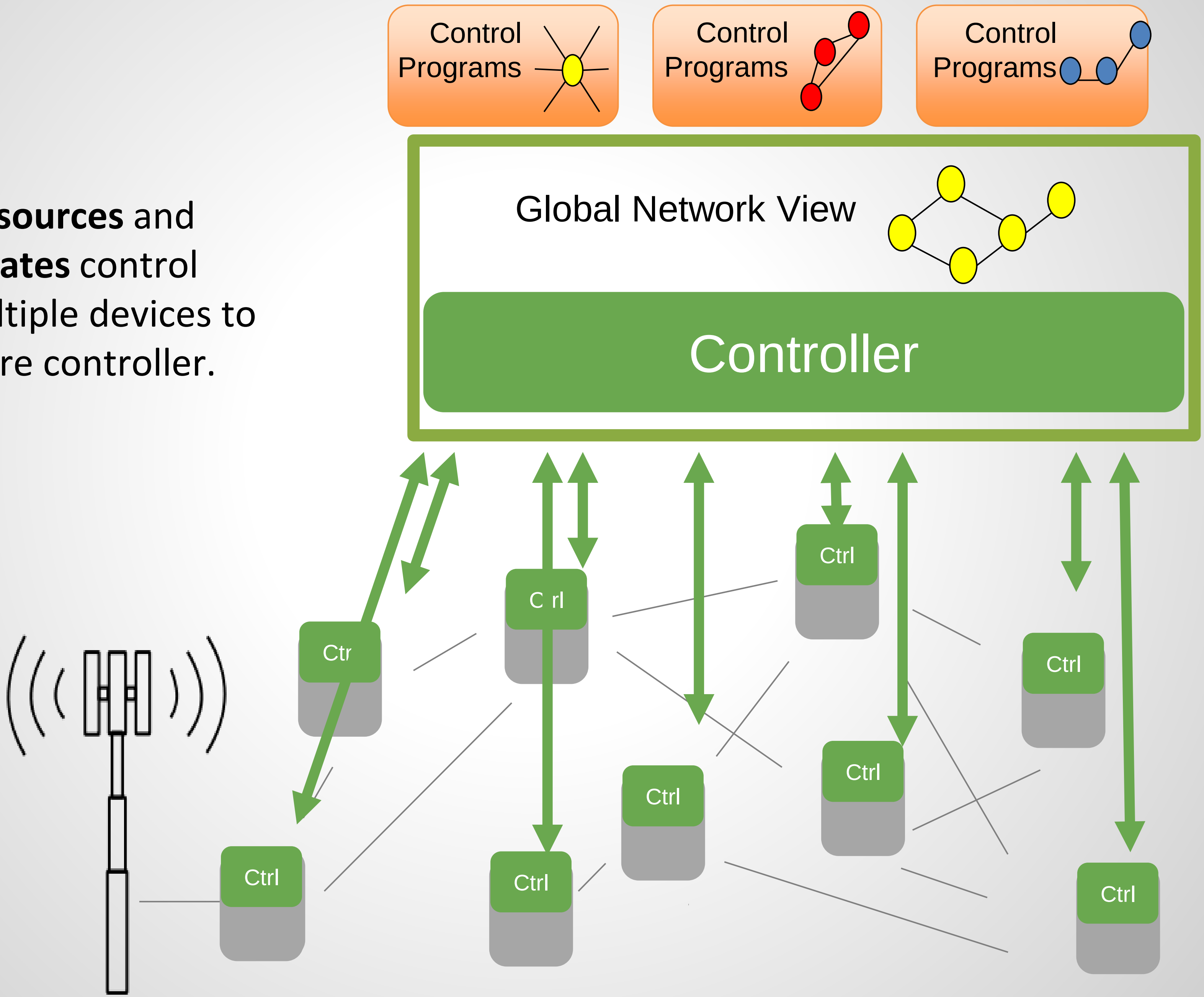
[C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection]
Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann.
12th USENIX Symposium on Networked Systems Design and Implementation (**NSDI**), Oakland, California, USA, May 2015..

Focus Today: Challenges Related to Programmability

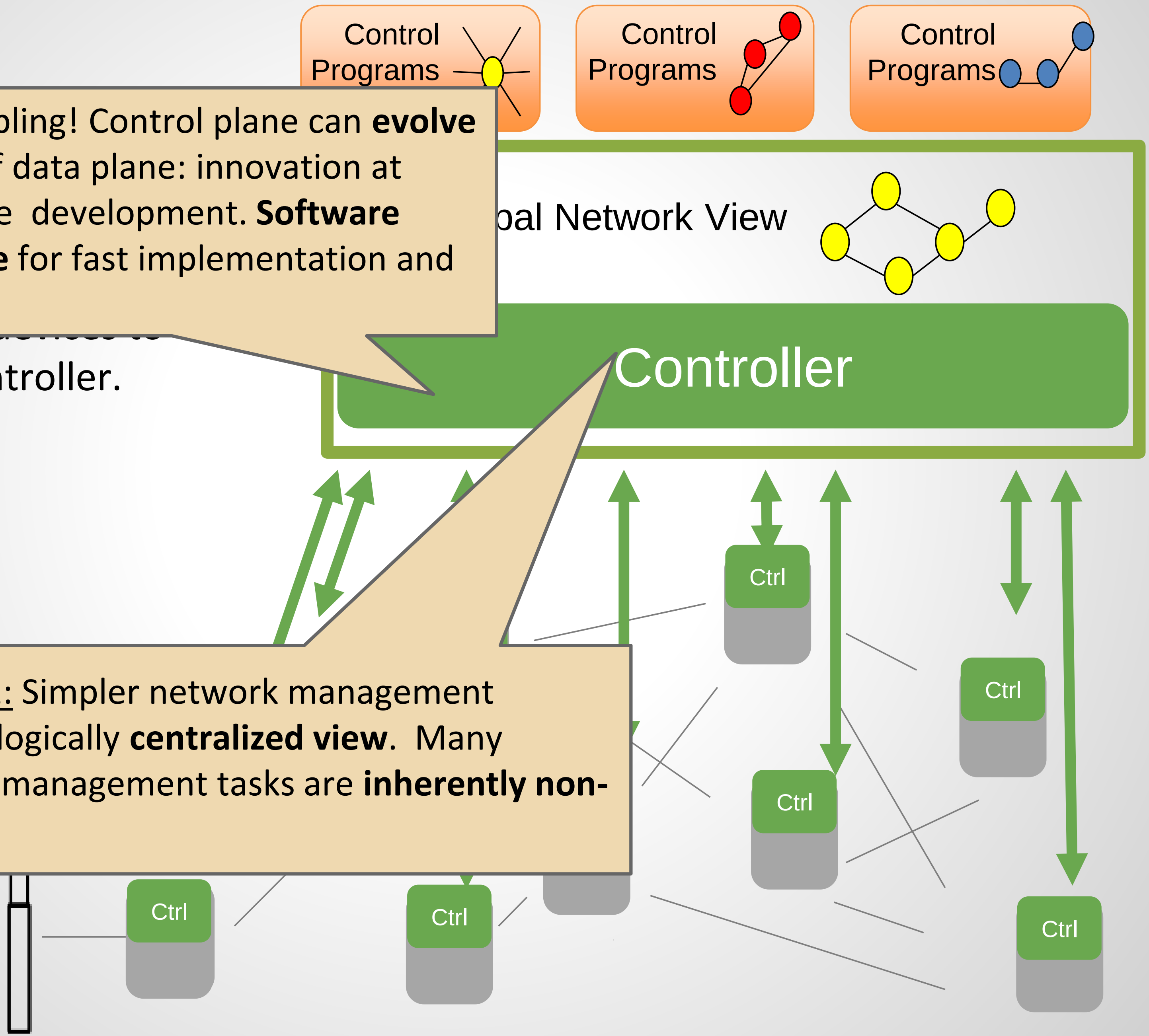
- Also one reason why I am here… ☺
  - German BSI project: How to make governmental networks and datacenters more secure?
- Startup on incremental SDN deployment in Berlin based on our USENIX ATC 2014 paper «Panopticon»
- Today: Network updates

SDN in a Nutshell

SDN **outsources** and **consolidates** control over multiple devices to a software controller.



SDN in a Nutshell

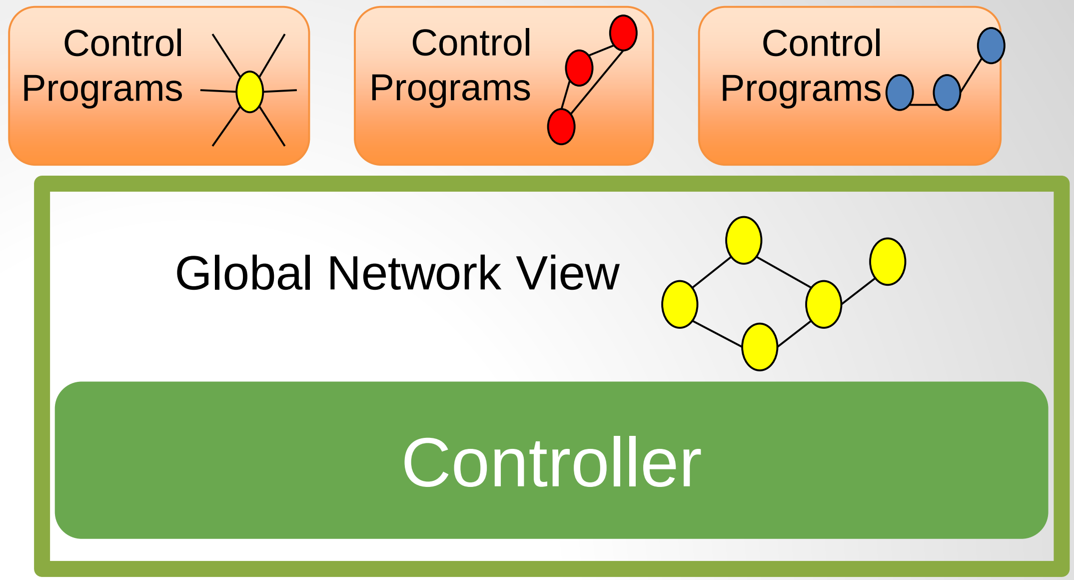


<u>Benefit 1:</u> Decoupling! Control plane can **evolve independently** of data plane: innovation at speed of software development. **Software trumps hardware** for fast implementation and deployment.

a software controller.

<u>Benefit 2:</u> Simpler network management through logically **centralized view**. Many network management tasks are **inherently non-local**.

SDN in a Nutshell



SDN **outsources** and **consolidates** control over multiple devices to a software controller.

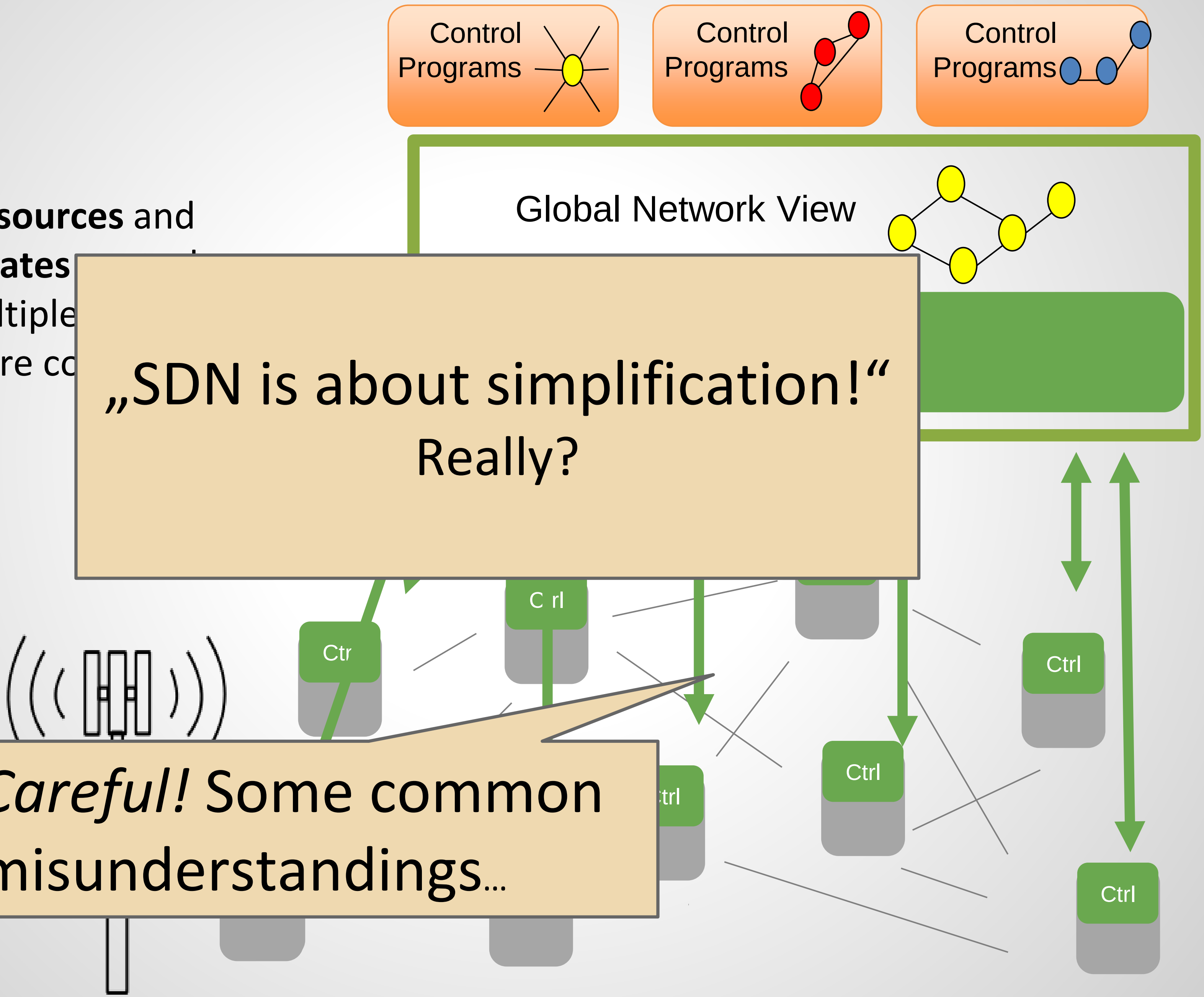
Benefit 3: Standard API OpenFlow is about **generalization**

- Generalize **devices** (L2-L4: switches, routers, middleboxes)
- Generalize **routing and traffic engineering** (not only destination-based)
- Generalize **flow-installation**: coarse-grained rules and wildcards okay, proactive vs reactive installation
- Provide general and logical **network views** to the application

Also: **match-action paradigm** = **formally verifiable** policies.

SDN in a Nutshell

SDN **outsources** and **consolidates** ...
over multiple
a software co

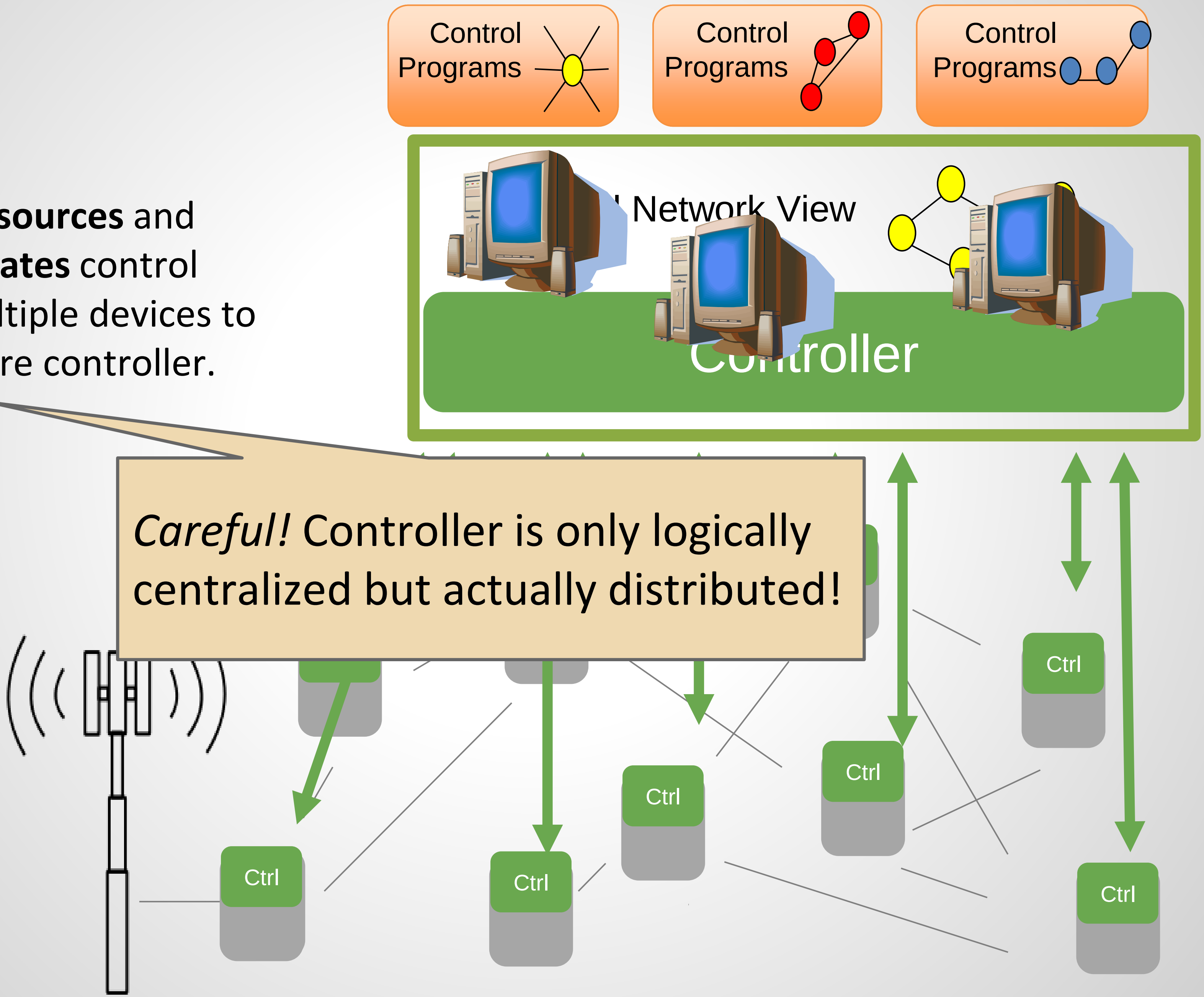


„SDN is about simplification!“
Really?

Careful! Some common misunderstandings…

SDN in a Nutshell

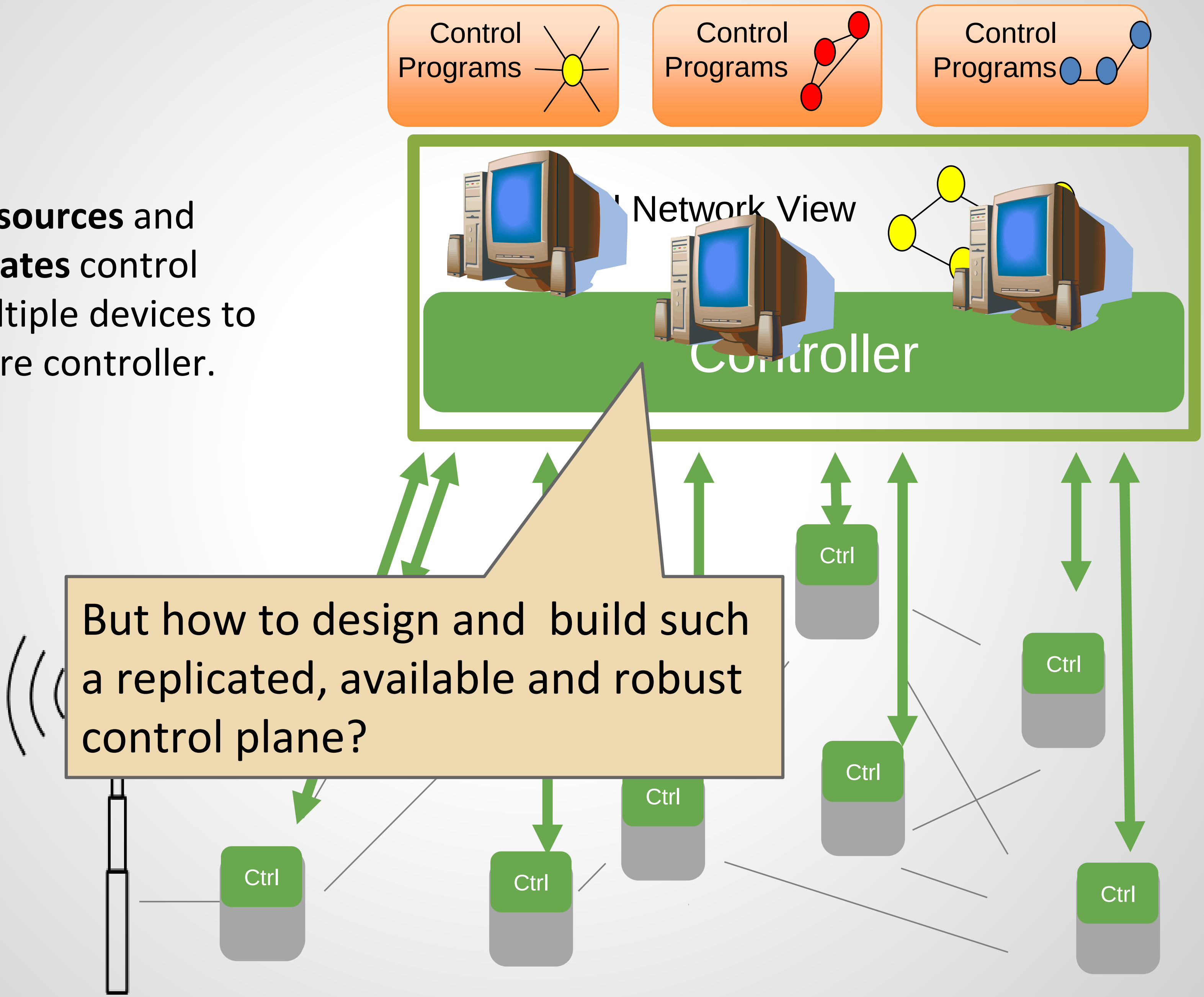
SDN **outsources** and **consolidates** control over multiple devices to a software controller.



Careful! Controller is only logically centralized but actually distributed!

SDN in a Nutshell

SDN **outsources** and **consolidates** control over multiple devices to a software controller.

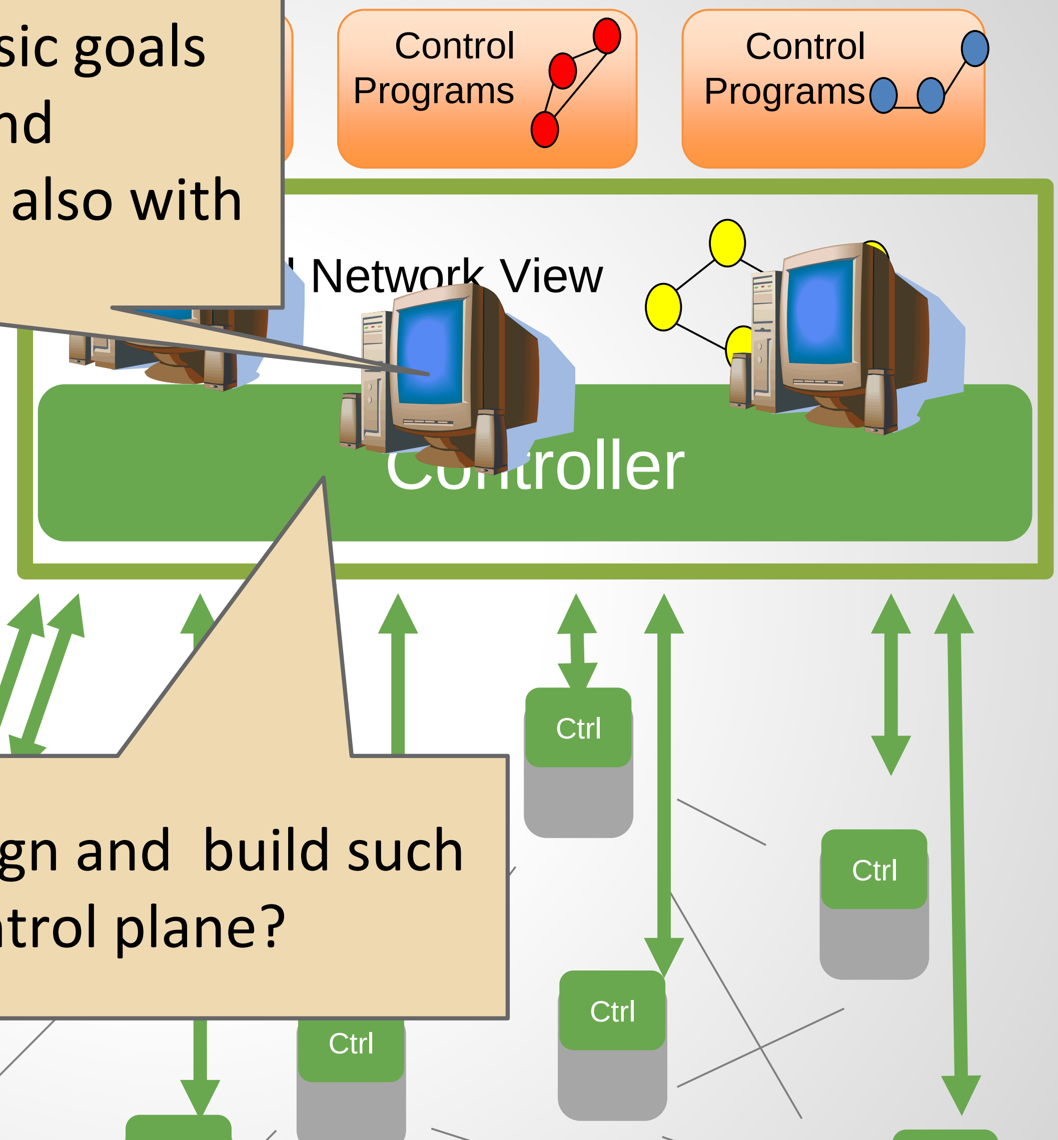


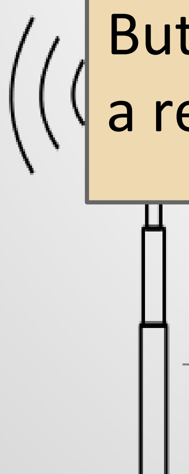
But how to design and build such a replicated, available and robust control plane?

SDN in a Nutshell

Can be seen as a transactional memory problem, with classic goals like safety (linearizability) and liveness (waitfreedom). But also with a twist… ☺

consolidates control over multiple devices to a software controller.



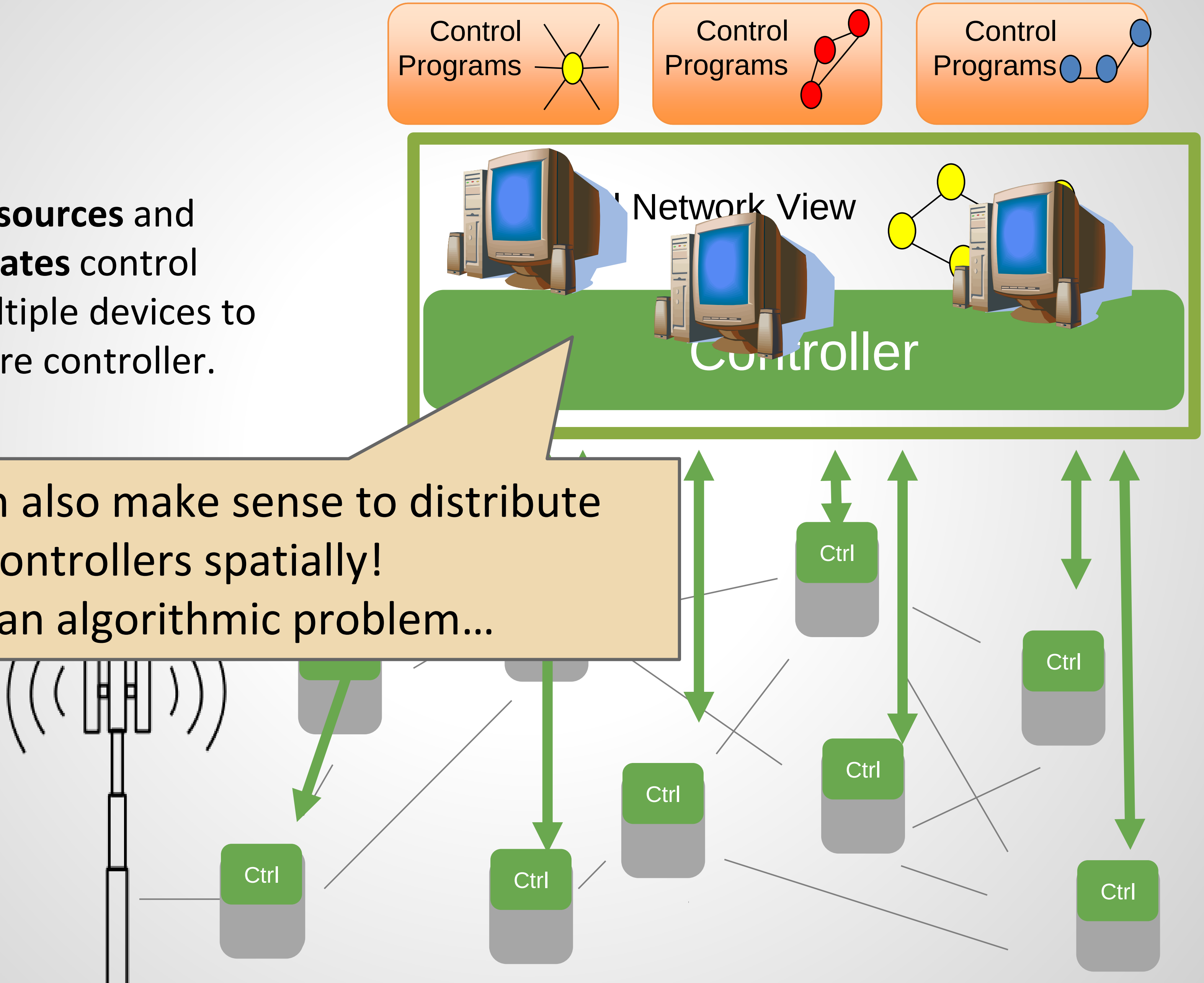


But how to design and build such a replicated control plane?

A Distributed and Robust SDN Control Plane for Transactional Network Updates
Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid.
34th IEEE Conference on Computer Communications (**INFOCOM**), Hong Kong, April 2015..

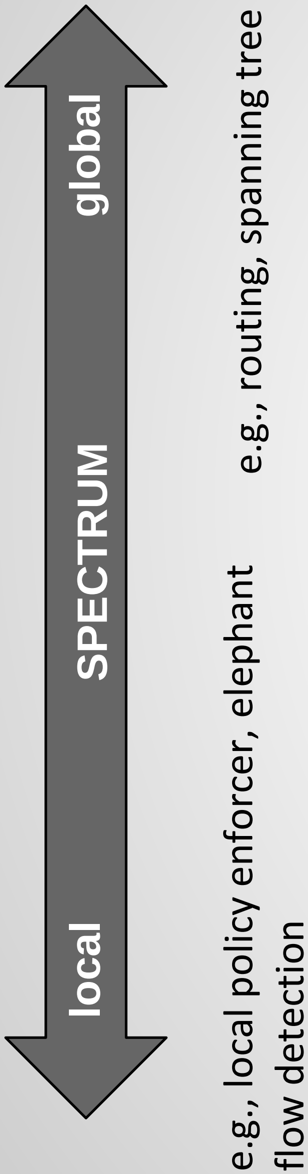
SDN in a Nutshell

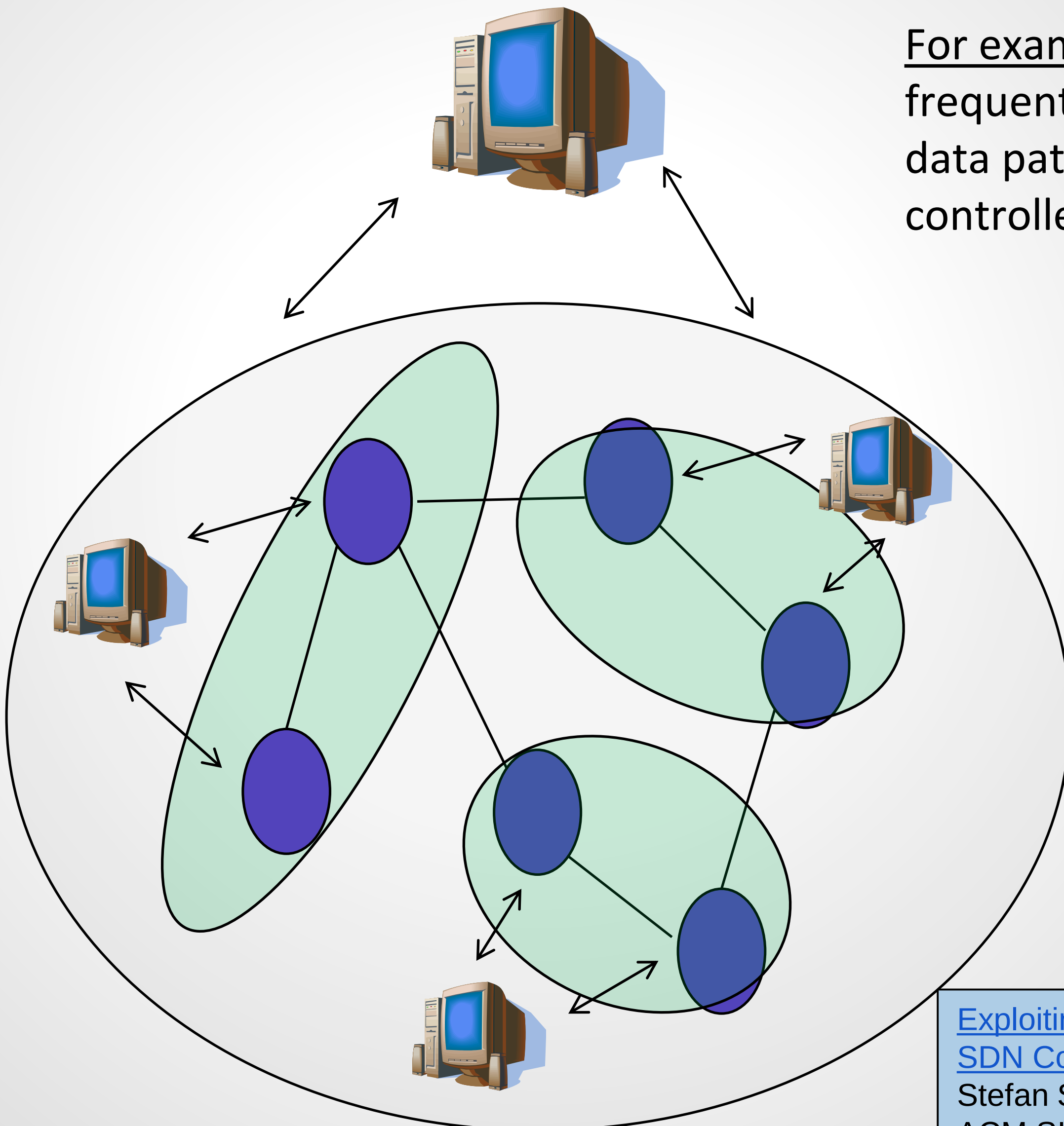
SDN **outsources** and **consolidates** control over multiple devices to a software controller.



It can also make sense to distribute the controllers spatially!
Also an algorithmic problem…

A Distributed Computing Challenge: What can and should be controlled locally?





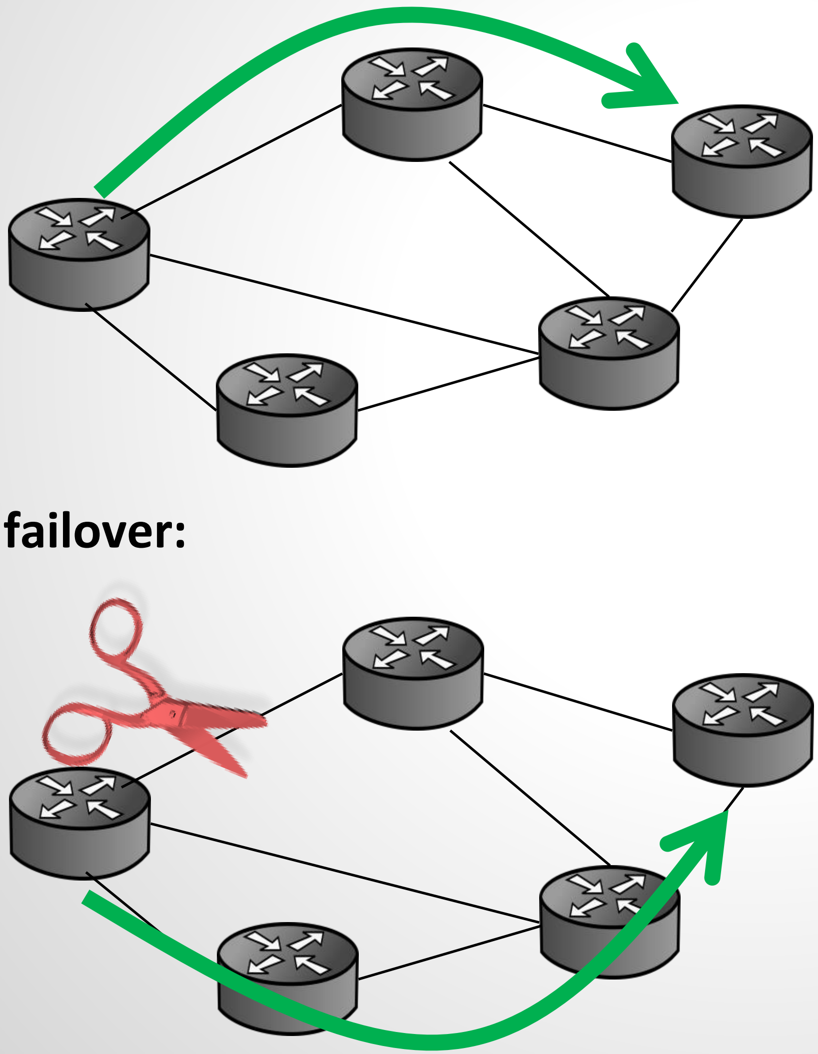
<u>For example:</u> Handle frequent events close to data path, shield global controllers.

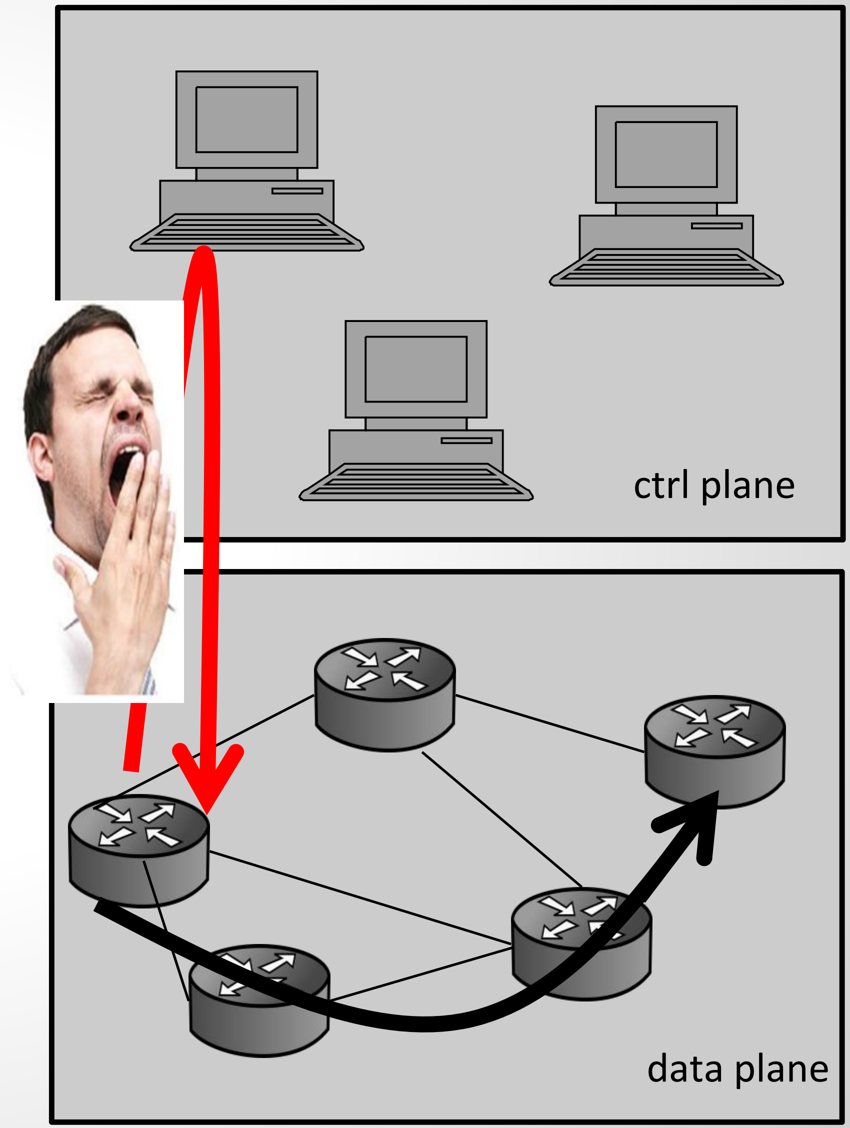
[Exploiting Locality in Distributed SDN Control]
Stefan Schmid and Jukka Suomela. ACM SIGCOMM **HotSDN** 2013.

Some Logic Should Even Remain in Data Plane!

Before failover:

After failover:

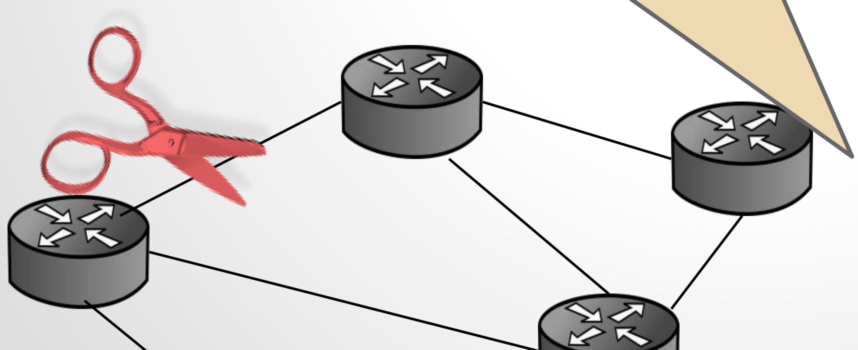


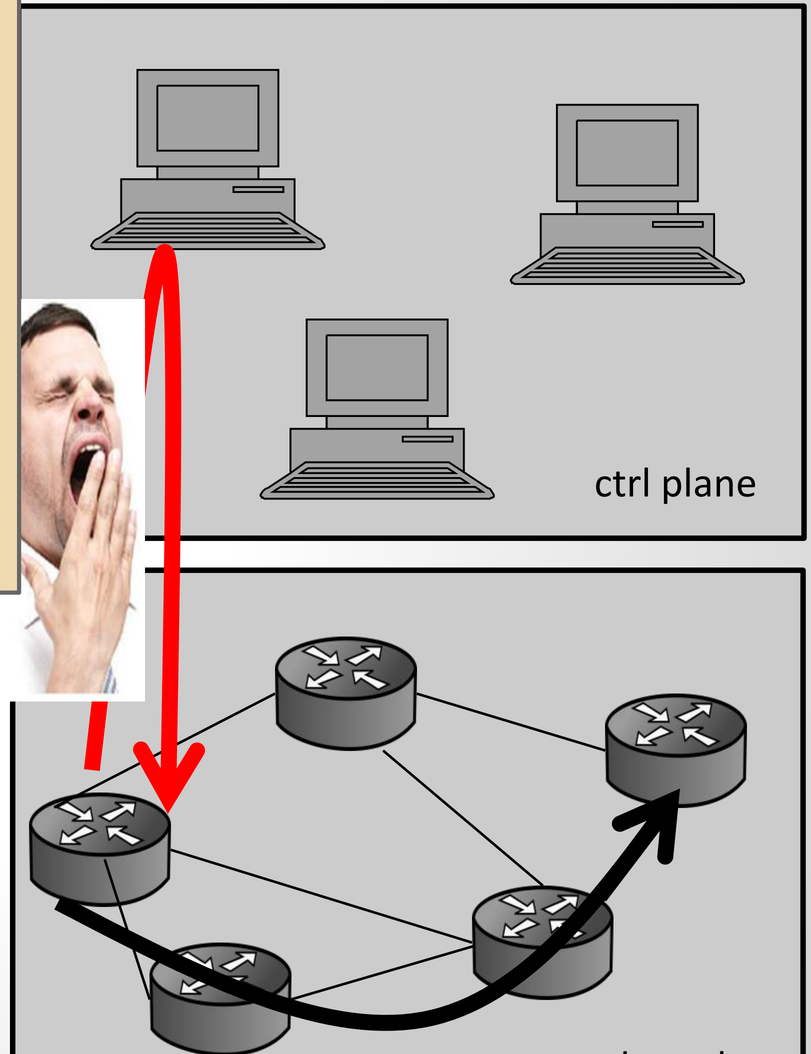


Some Logic Should Even Remain in Data Plane!

Goal: Find routing path (inband) **as long as it exists**! A classic algorithmic problem (traversal with local information) in a **new flavor**. OpenFlow vs MPLS: Cannot implement Ankit et al.'s link reversal algorithm?

After failover:



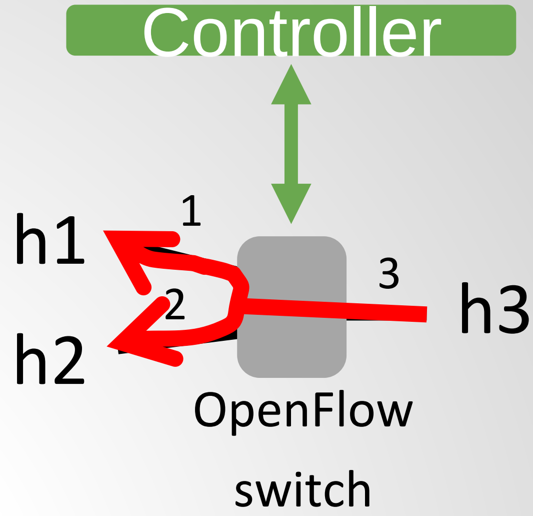


[Provable Data Plane Connectivity with Local Fast Failover: Introducing OpenFlow Graph Algorithms]
Michael Borokhovich, Liron Schiff, and Stefan Schmid.
ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (**HotSDN**), Chicago, Illinois, USA, August 2014.

SDN raises fundamental algorithmic problems even for scenarios with a single controller!*

* And sometimes even a single switch…

Jennifer Rexford's Example: SDN MAC Learning Done Wrong



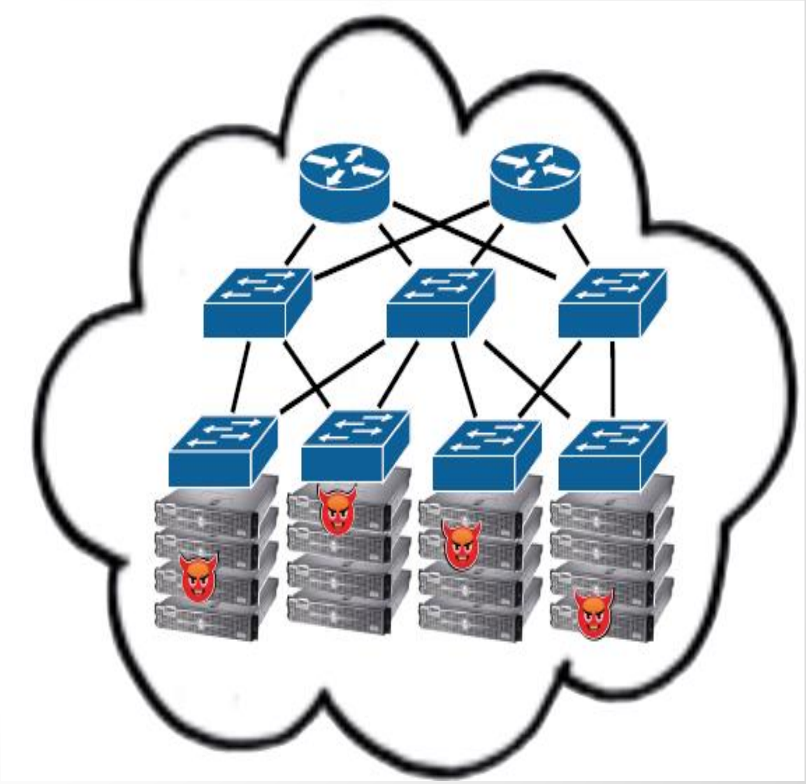
- ❑ MAC learning: The «Hello World»
  - ❑ a bug in early controller versions
- ❑ In legacy networks simple
  - ❑ Flood packets sent to unknown destinations
  - ❑ Learn host's location when it sends packets
- ❑ Pitfalls in SDN: learn sender => miss response
  - ❑ Assume: low priority rule * (no match): send to controller
  - ❑ h1->h2: Add rule h1@port1 (location learned)
  - ❑ Controller misses h2->h1 (as h1 known, h2 stay unknown!)
  - ❑ When h3->h2: flooding forever (learns h3, never learns h2)

Thanks to Jen Rexford for example!

Why Consistency Matters

Important, e.g., in Cloud

What if your traffic was *not* isolated from other tenants during periods of routine maintenance?



Thanks to Nate Foster for example!

Example: Outages

Even technically sophisticated companies are struggling to build networks that provide reliable performance.

We discovered a misconfiguration on this pair of switches that caused what's called a "bridge loop" in the network.

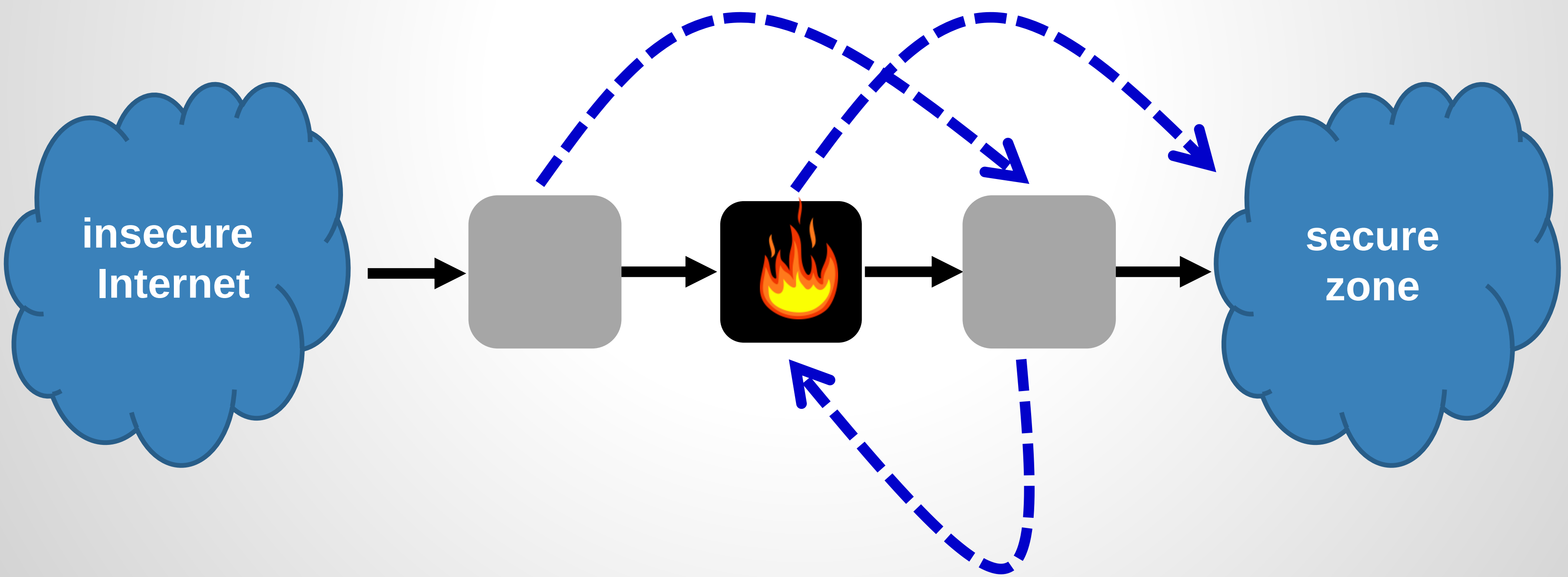
A network change was […] executed incorrectly […] more "stuck" volumes and added more requests to the re-mirroring storm

Service outage was due to a series of internal network events that corrupted router data tables

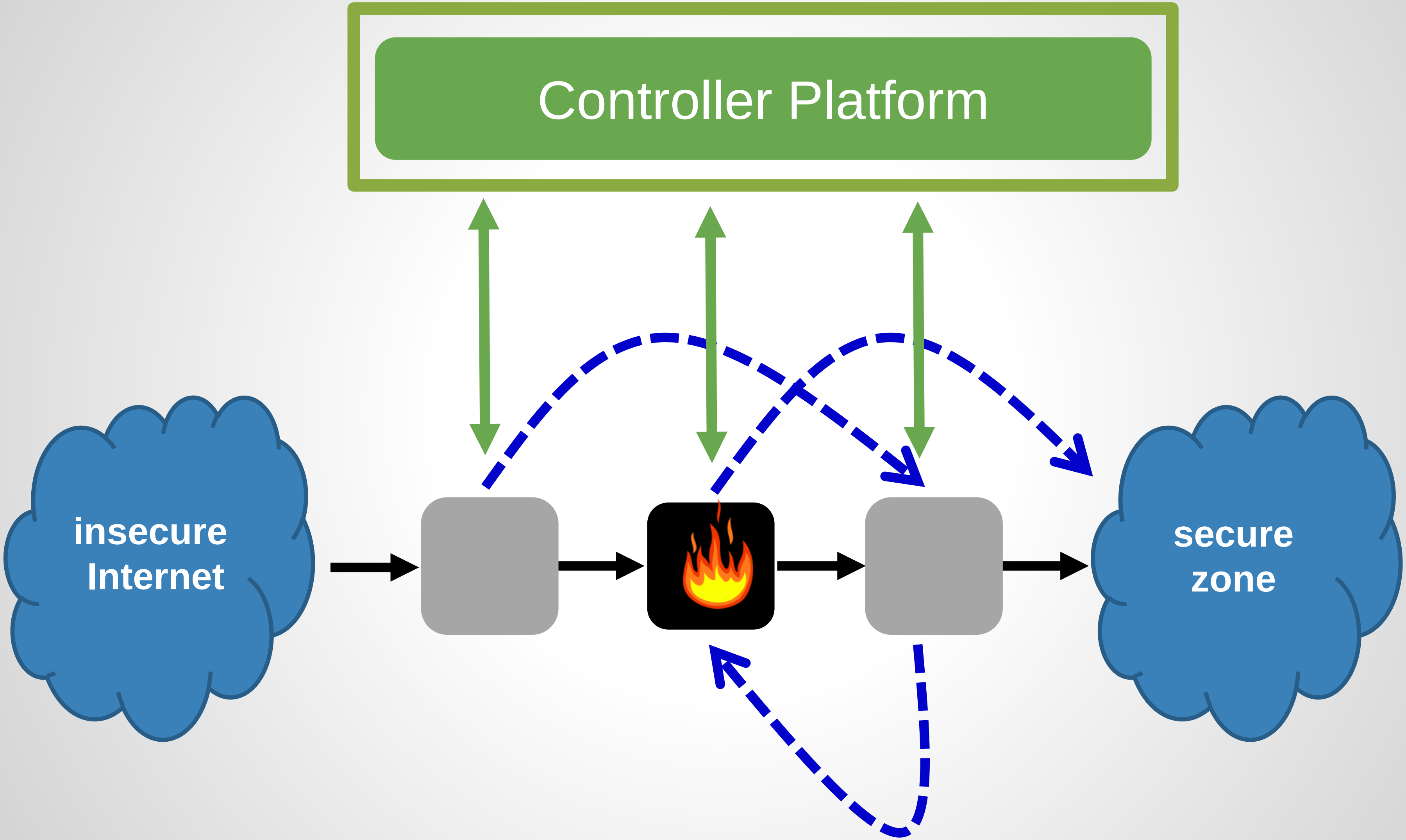
Experienced a network connectivity issue […] interrupted the airline's flight departures, airport processing and reservations systems

Thanks to Nate Foster for examples (at DSDN 2014)!

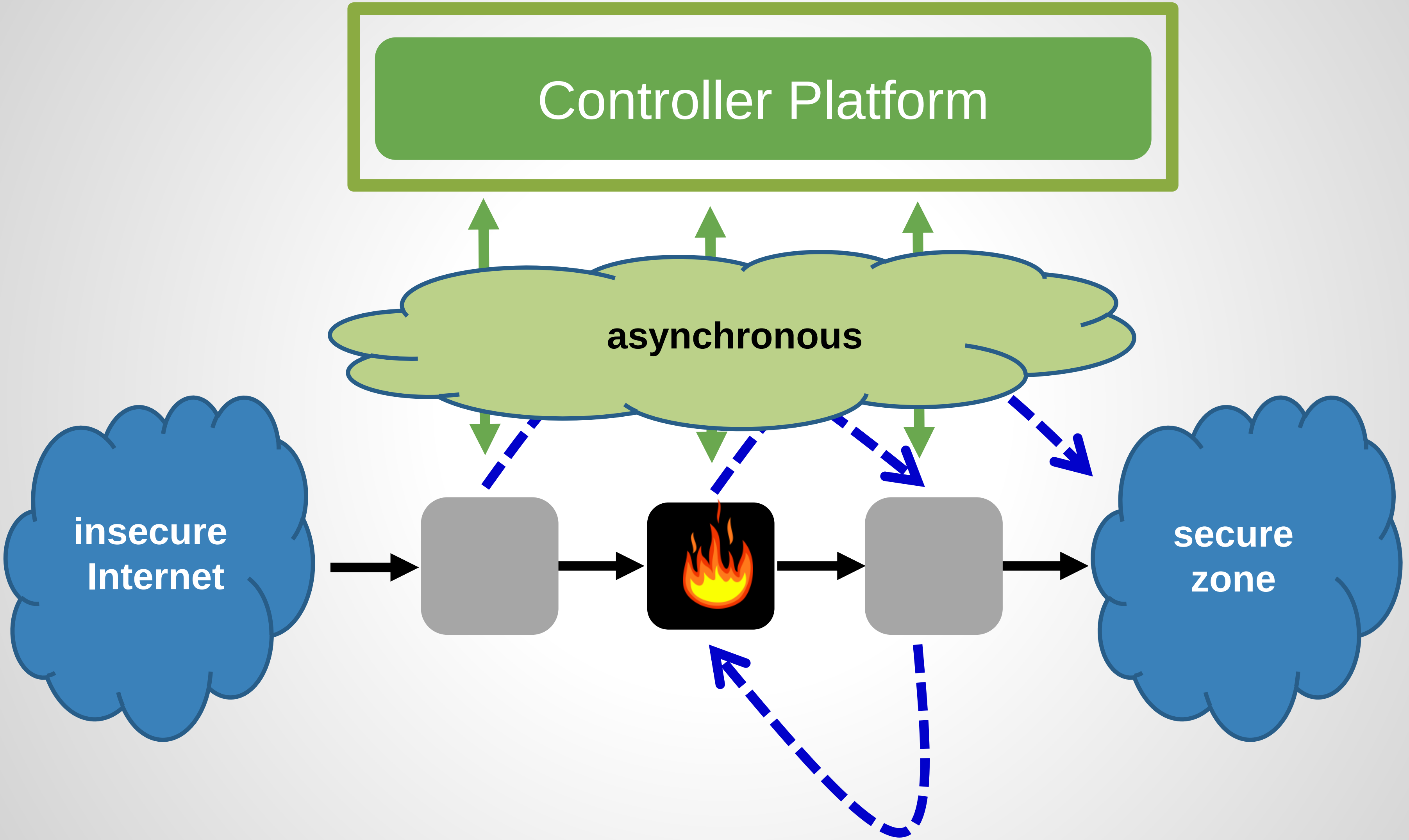
Challenge: Multi-Switch Updates



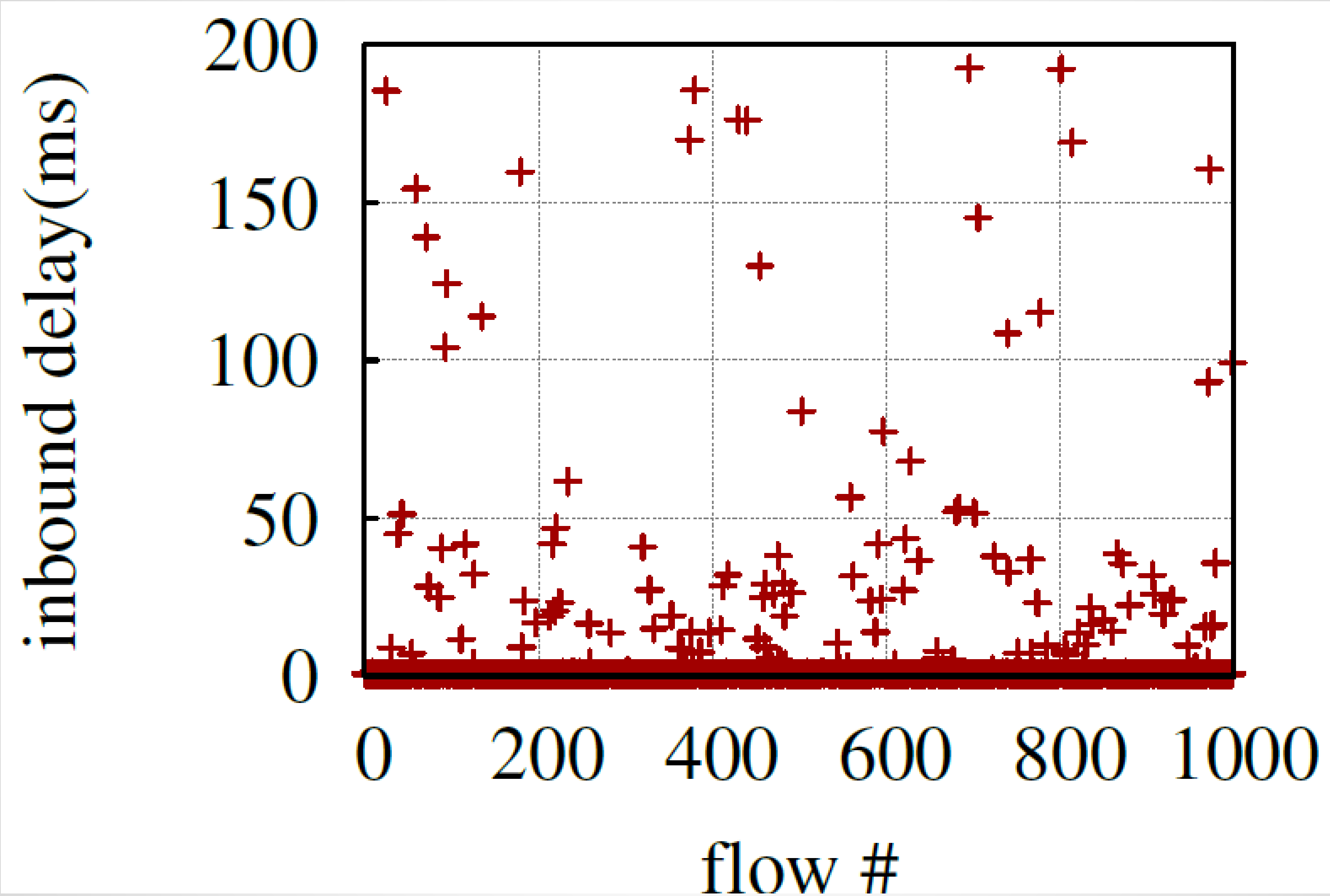
Challenge: Multi-Switch Updates



Challenge: Multi-Switch Updates



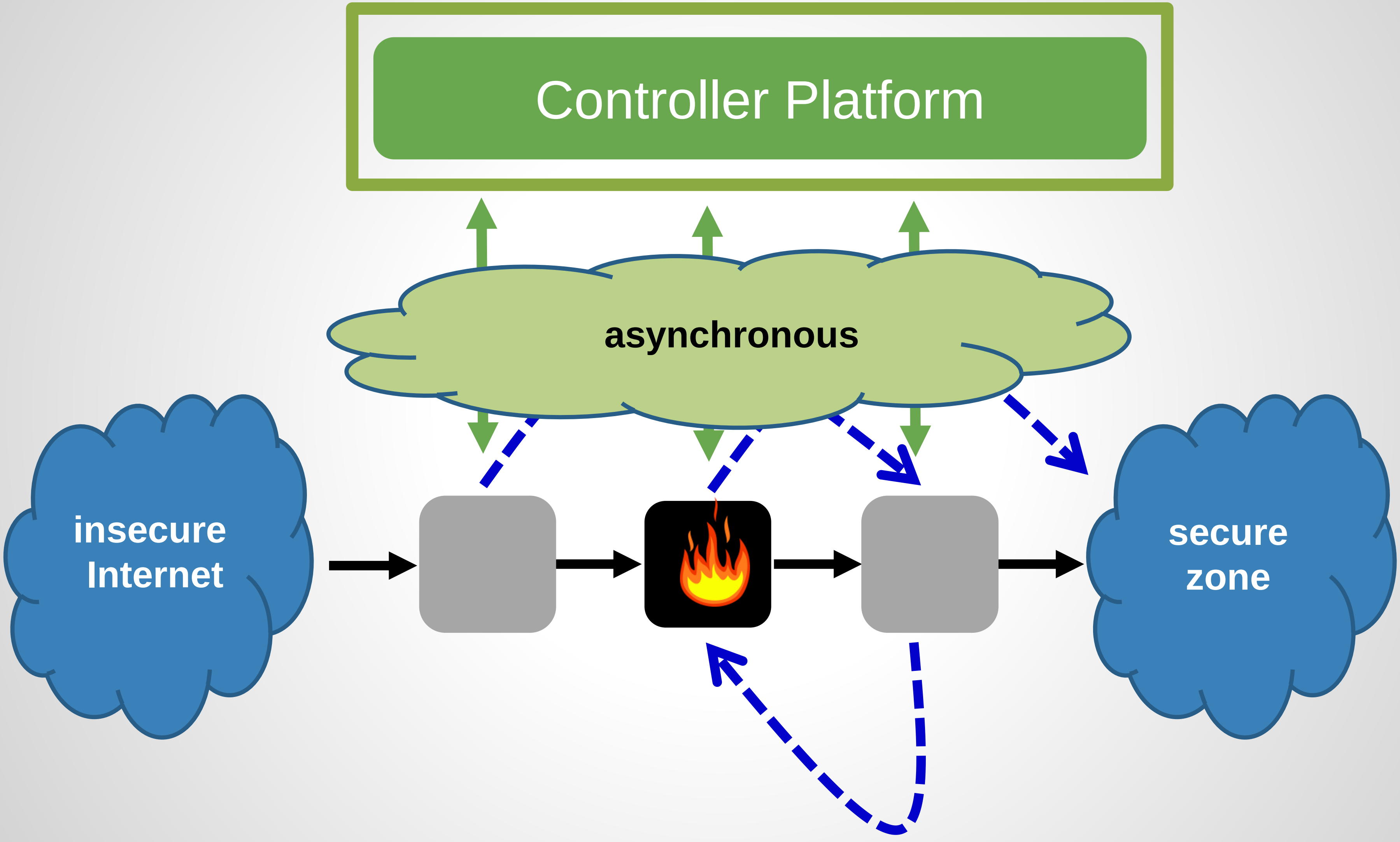
An Asynchronous Distributed System



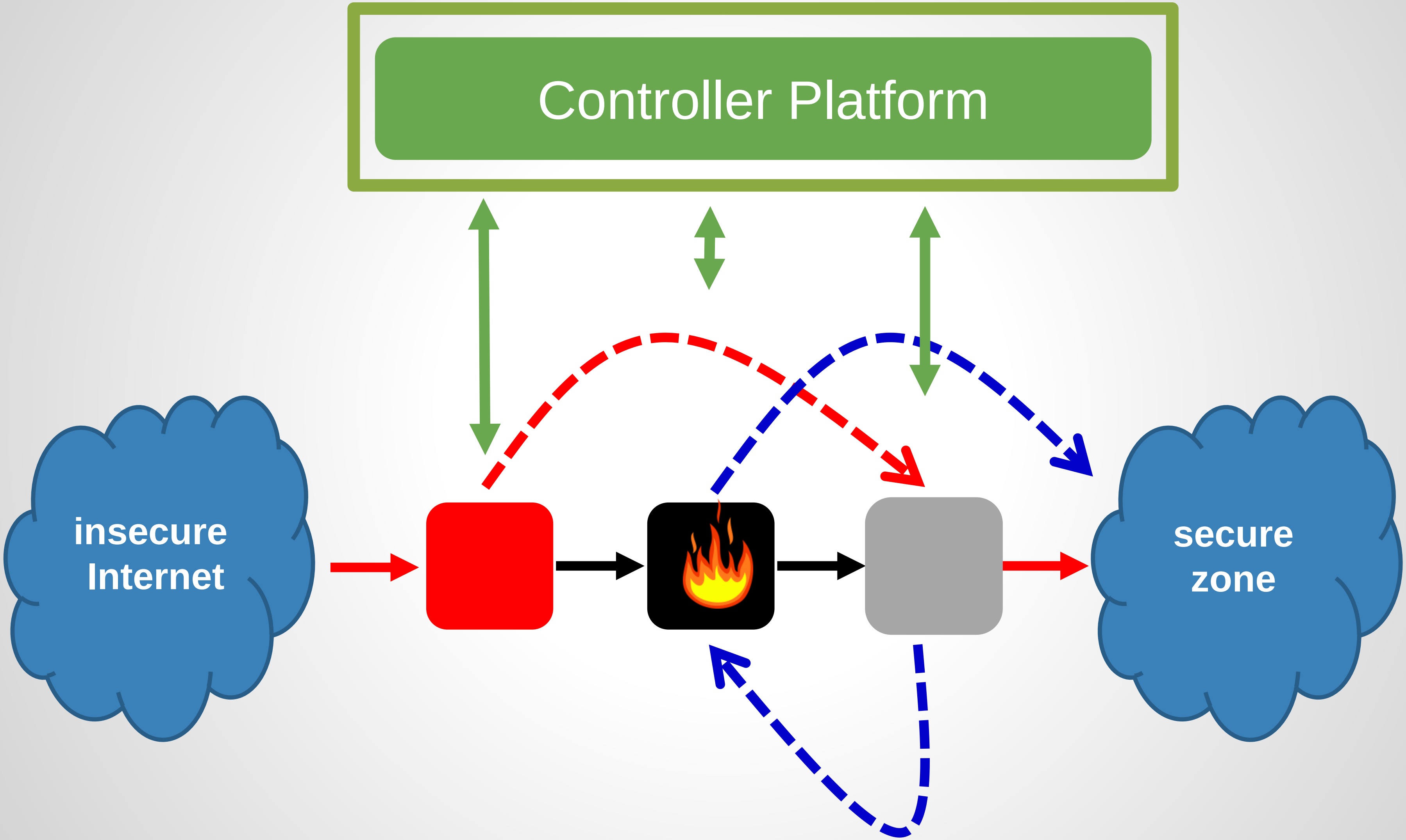
He et al., ACM SOSR 2015: without network latency

Jin et al., ACM SIGCOMM 2014: even higher variance

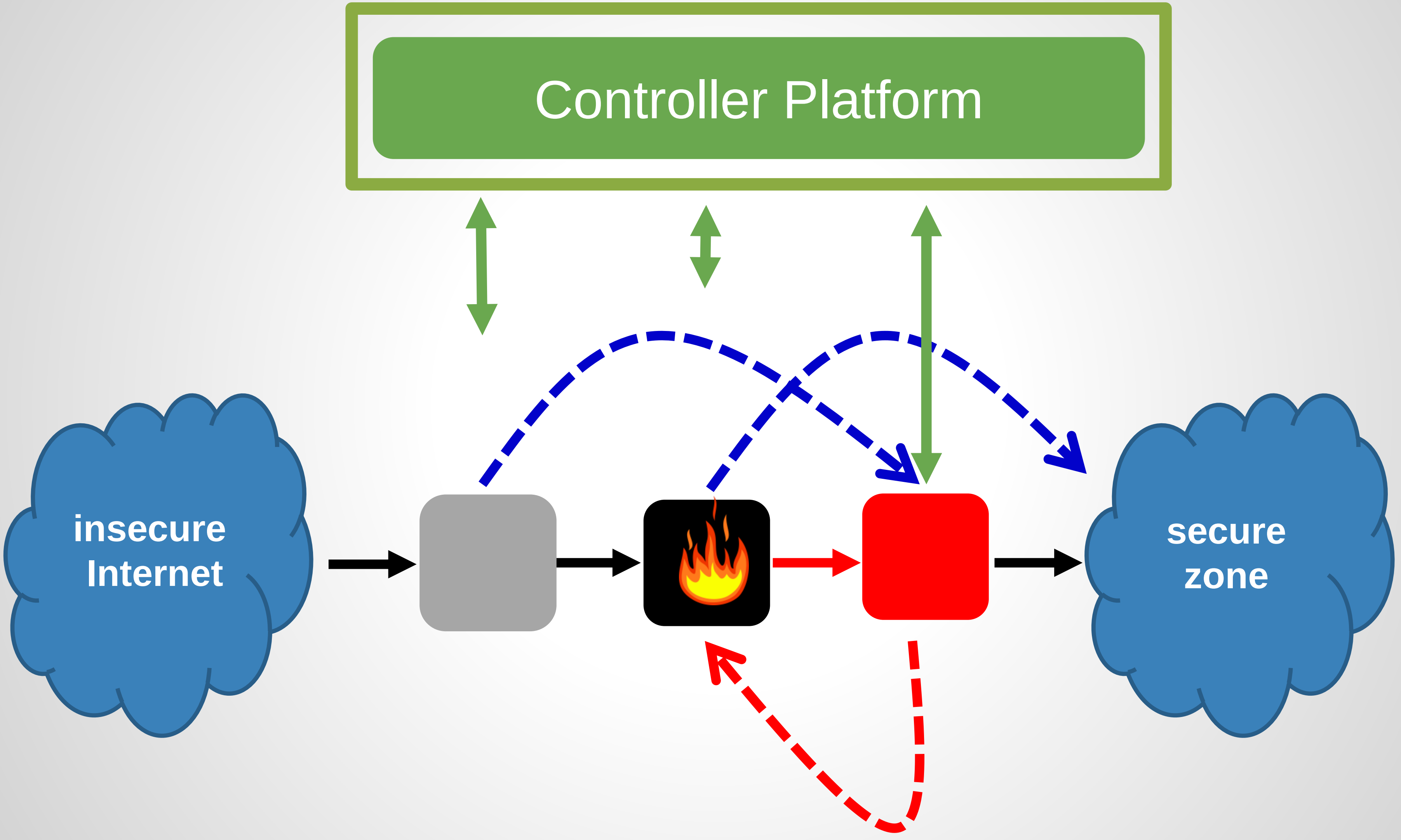
What Can Go Wrong?



Example 1: Bypassed Waypoint



Example 2: *Transient* Loop



What kind and level of consistency is needed?

What kind and level of consistency is needed?

It depends ☺

The Spectrum of Consistency

per-packet consistency

Reitblatt et al., SIGCOMM 2012

correct network virtualization

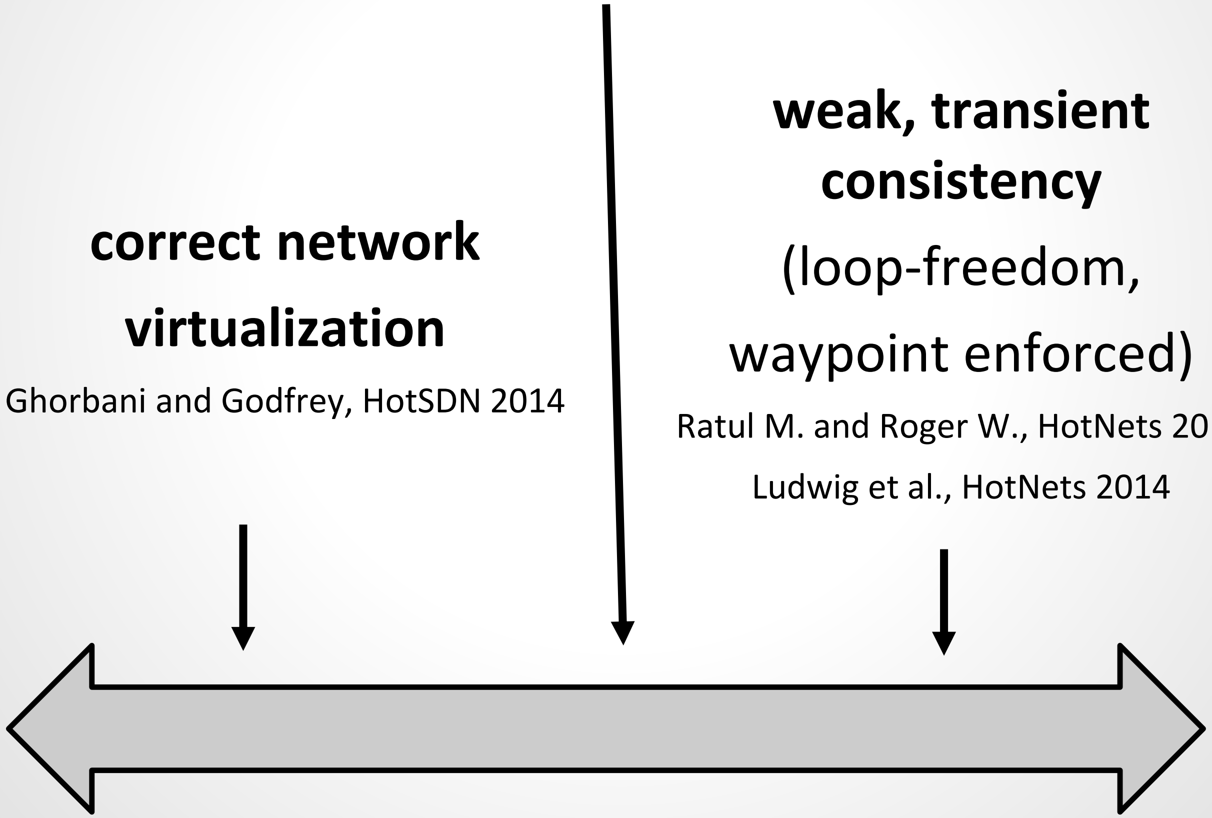
Ghorbani and Godfrey, HotSDN 2014

weak, transient consistency

(loop-freedom, waypoint enforced)

Ratul M. and Roger W., HotNets 2014

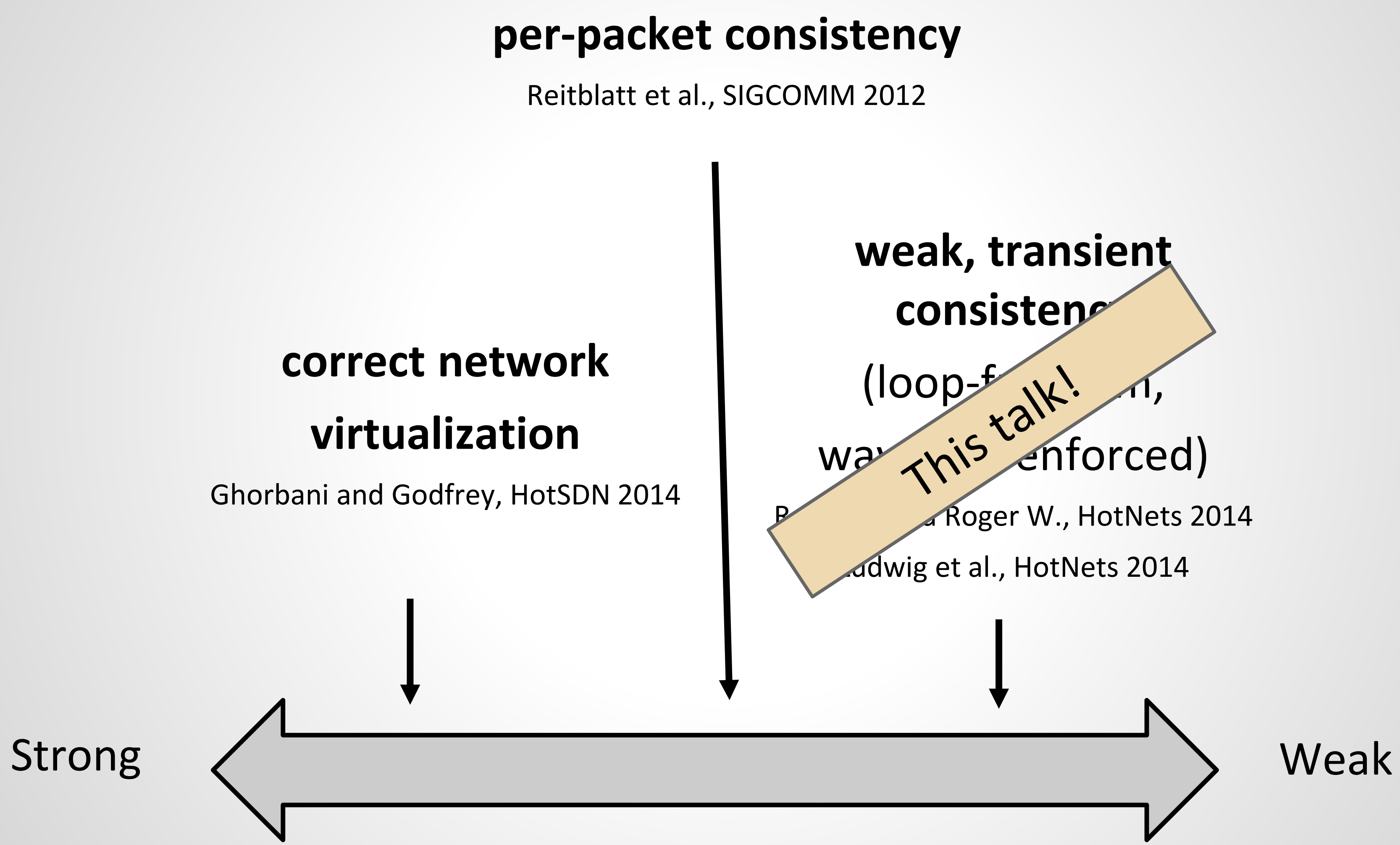
Ludwig et al., HotNets 2014



Strong

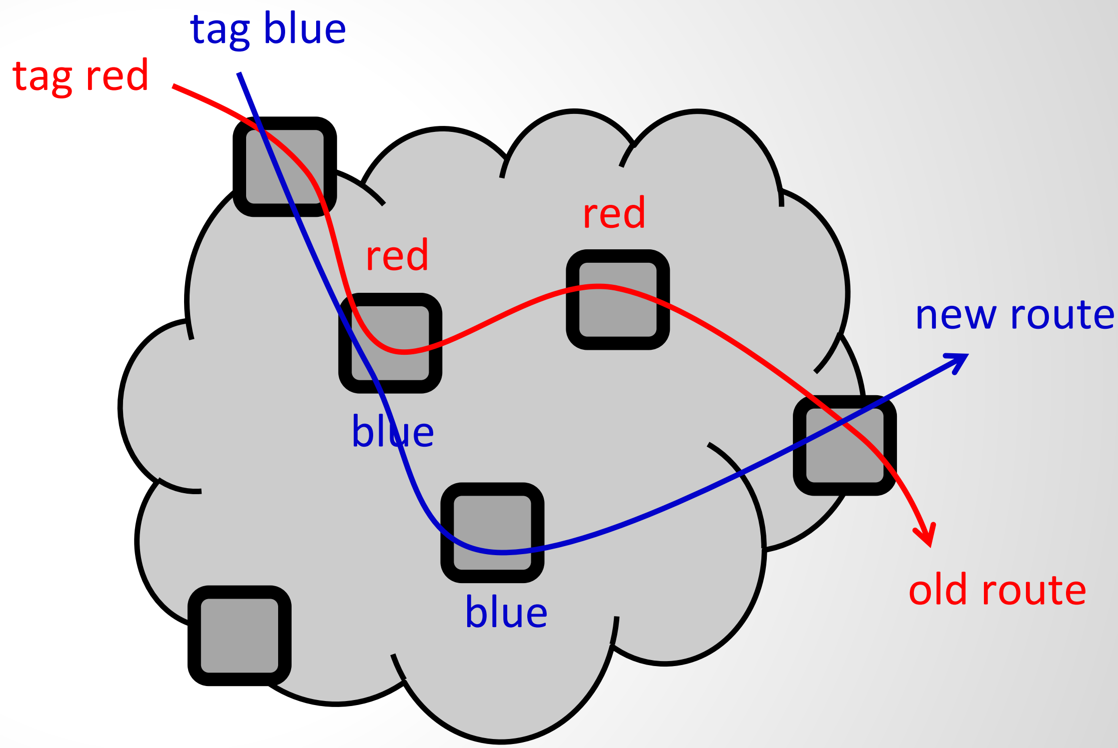
Weak

The Spectrum of Consistency



Almost everything can be solved with tagging…

- Old route: red
- New route: blue
- 2-Phase Update:
  - Install blue flow rules internally
  - Flip tag at ingress ports



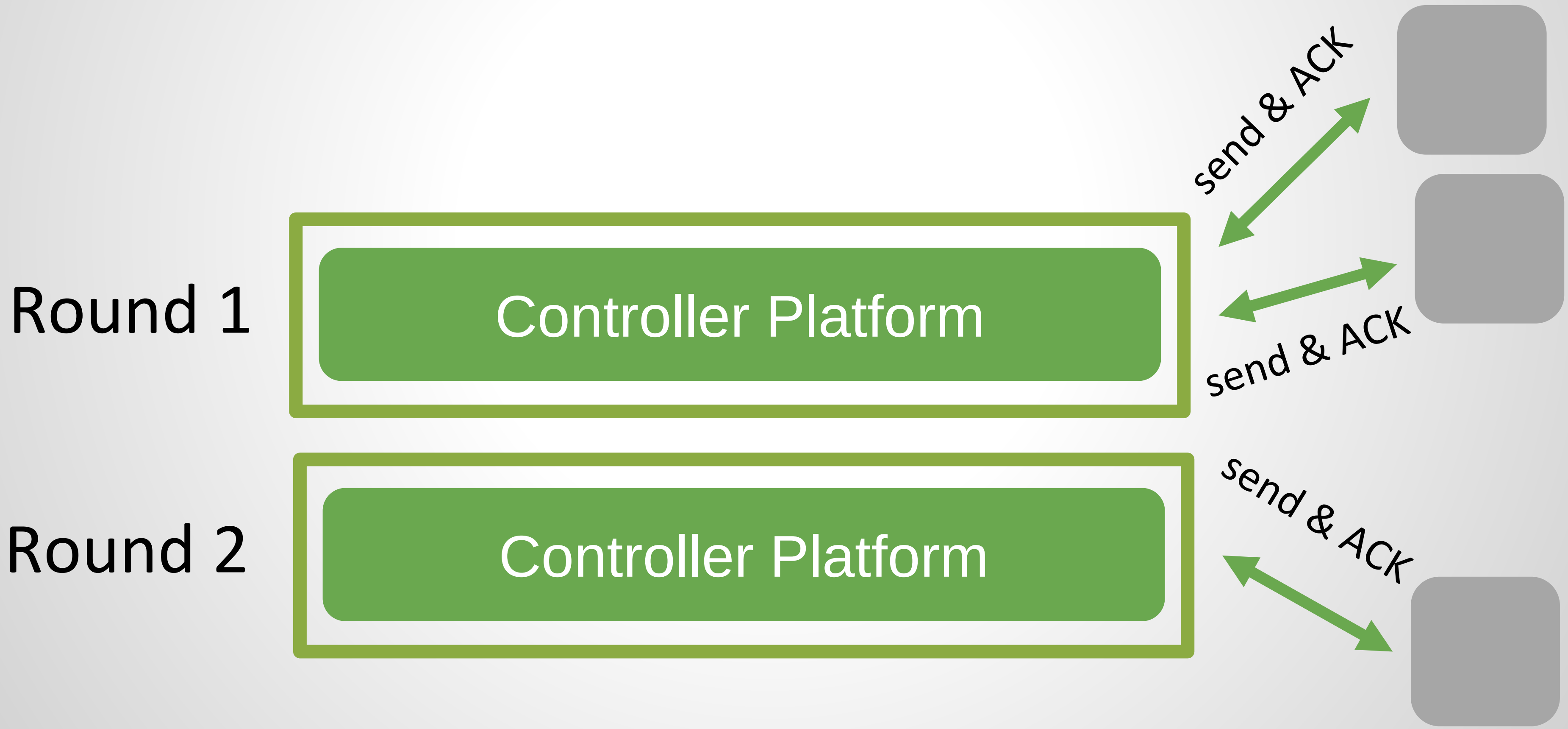
The Case Against Tagging

- Correctness:
  - Where to tag? Don’t interfere with existing protocols!
  - Tagging in the presence of middleboxes?
- Overhead:
  - Header space is limited
  - Looking up special header fields and tagging: extra latency?
  - The approach requires extra rules on the switch (TCAM memory is a scarce resource)
  - Coordination problem for distributed controllers?
- Late updates:
  - Updates start taking place late*

* Mahajan & Wattenhofer, ACM HotNets 2013

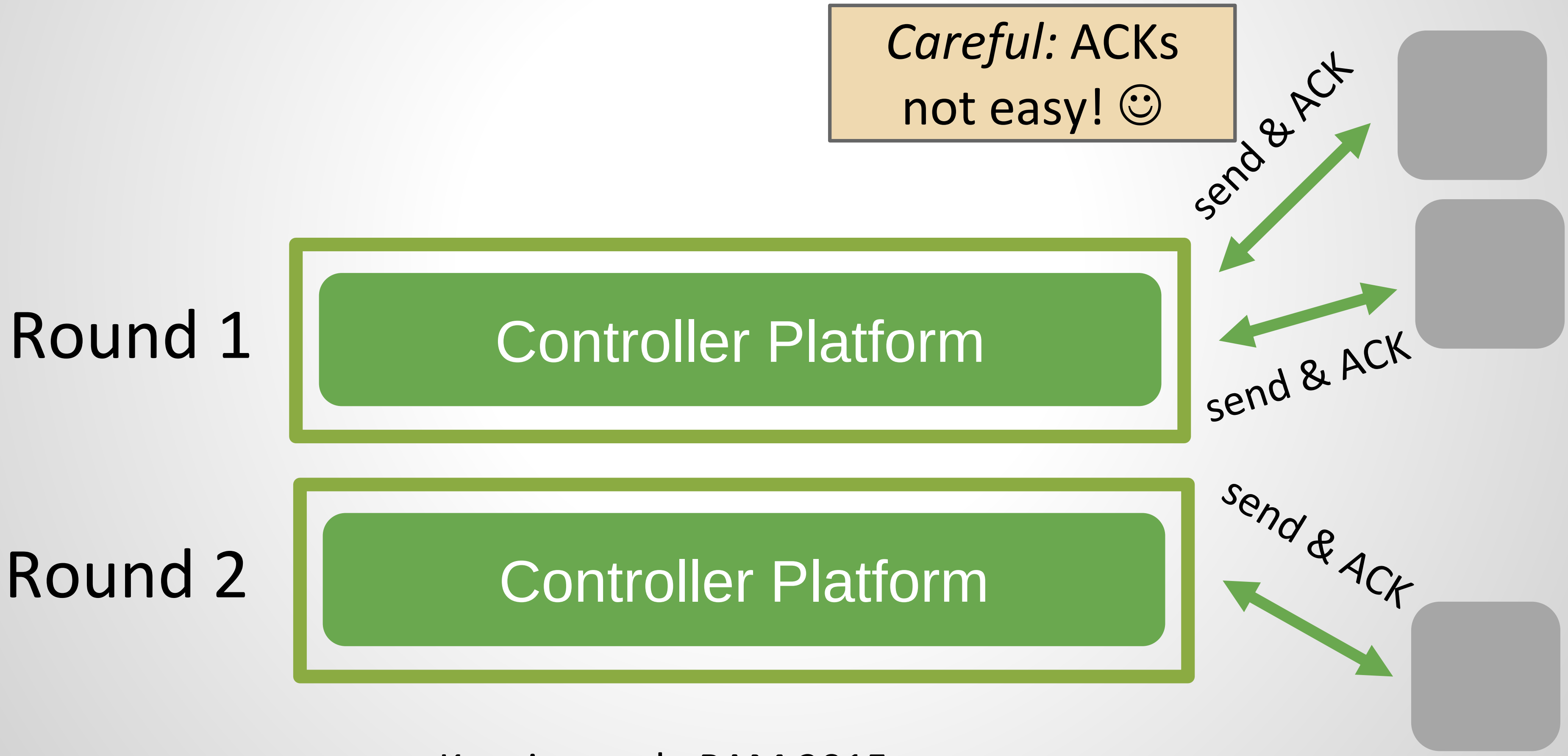
Transient Consistency: Model

Idea: Keep consistent by updating in multiple rounds



Transient Consistency: Model

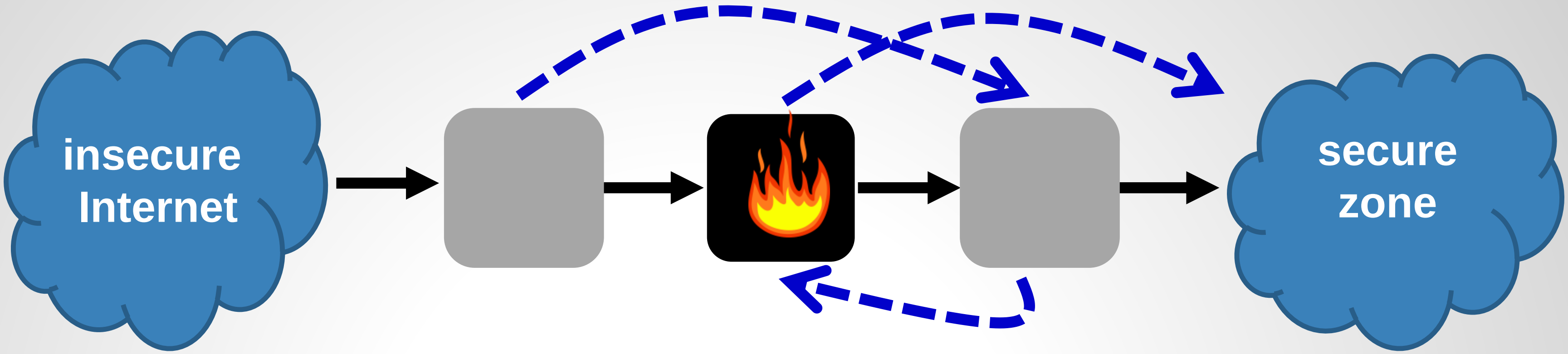
Idea: Keep consistent by updating in multiple rounds



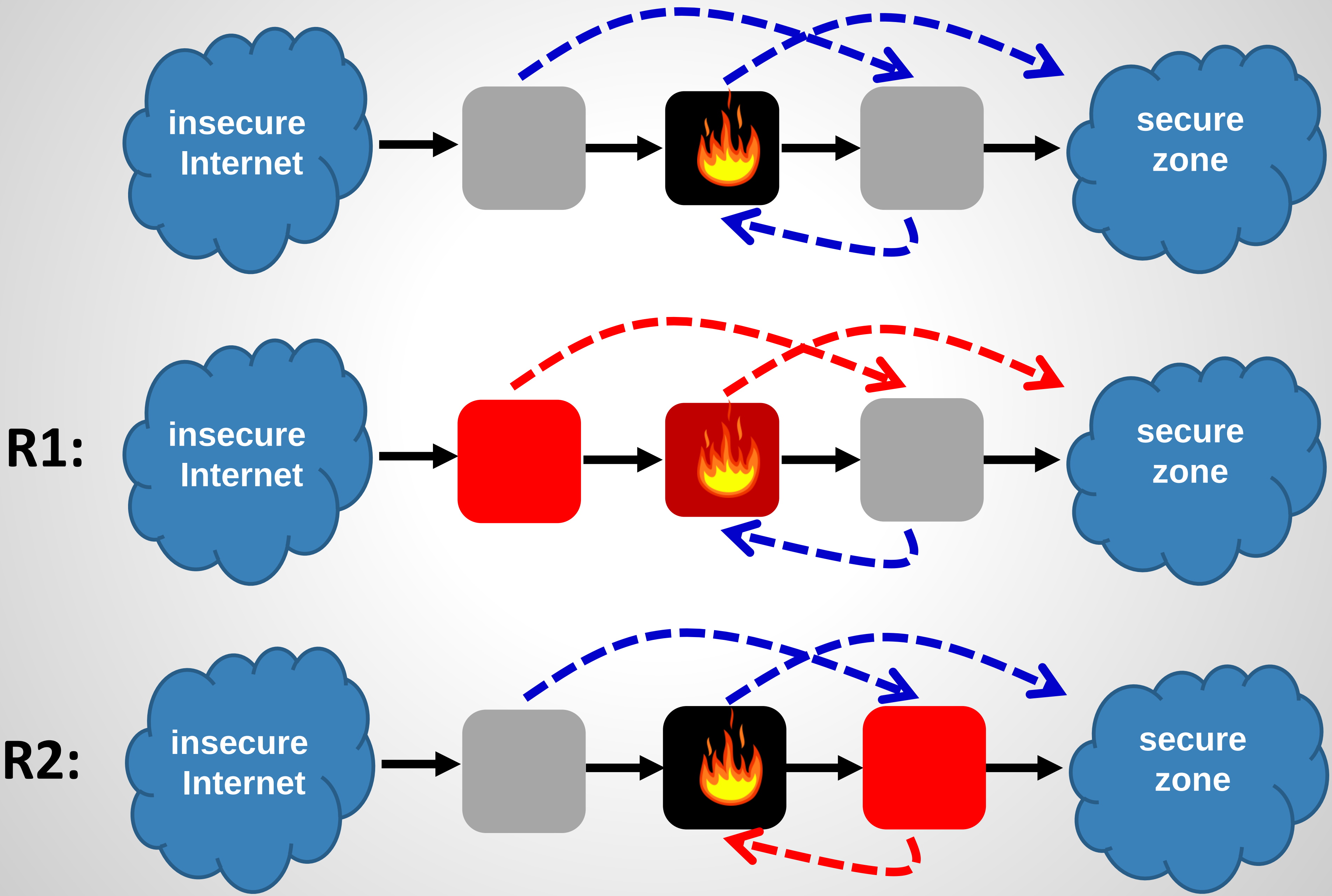
Kuzniar et al., PAM 2015

Kuzniar et al., ACM CONEXT 2014

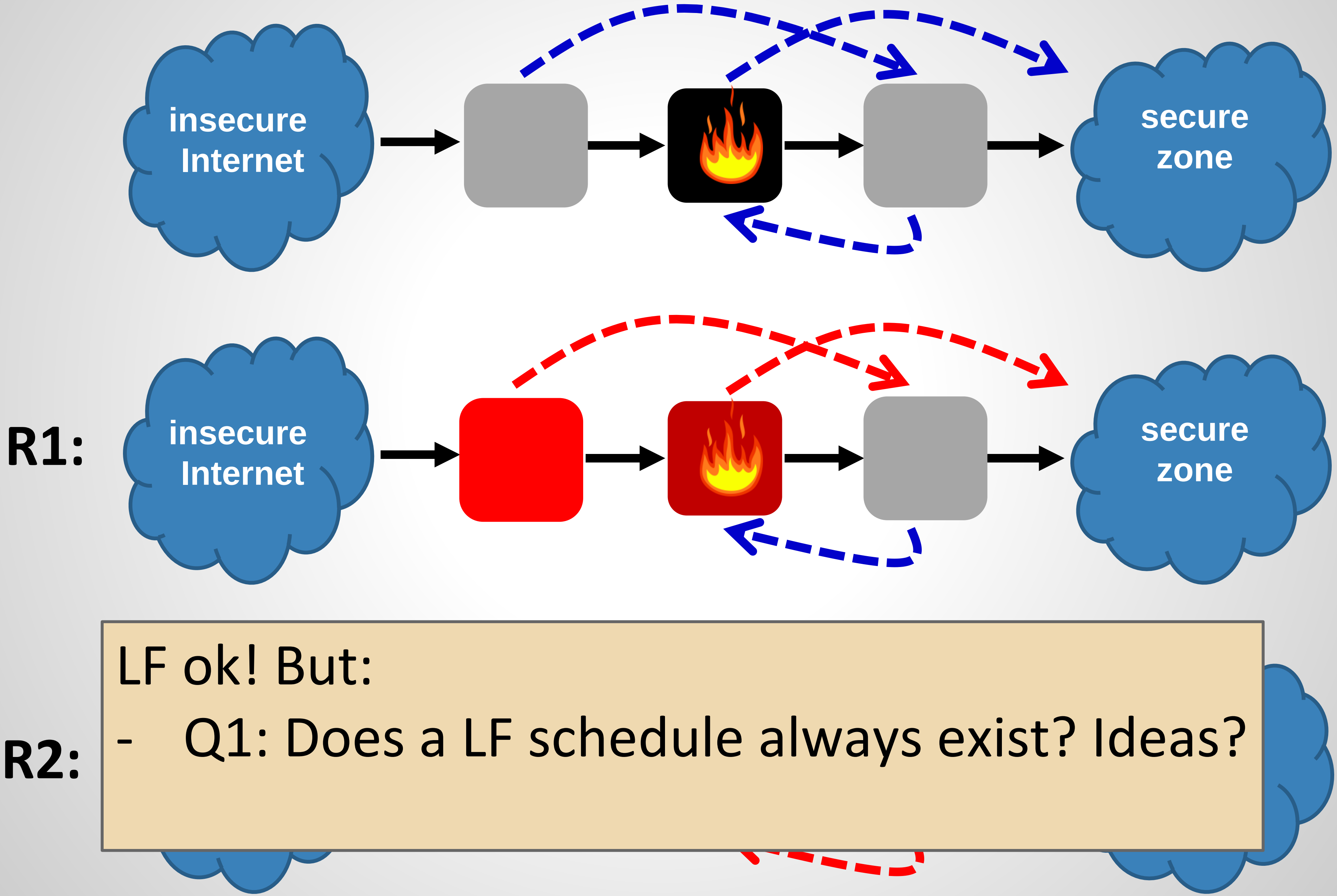
Going Back to Our Examples: LF Update?



Going Back to Our Examples: LF Update!



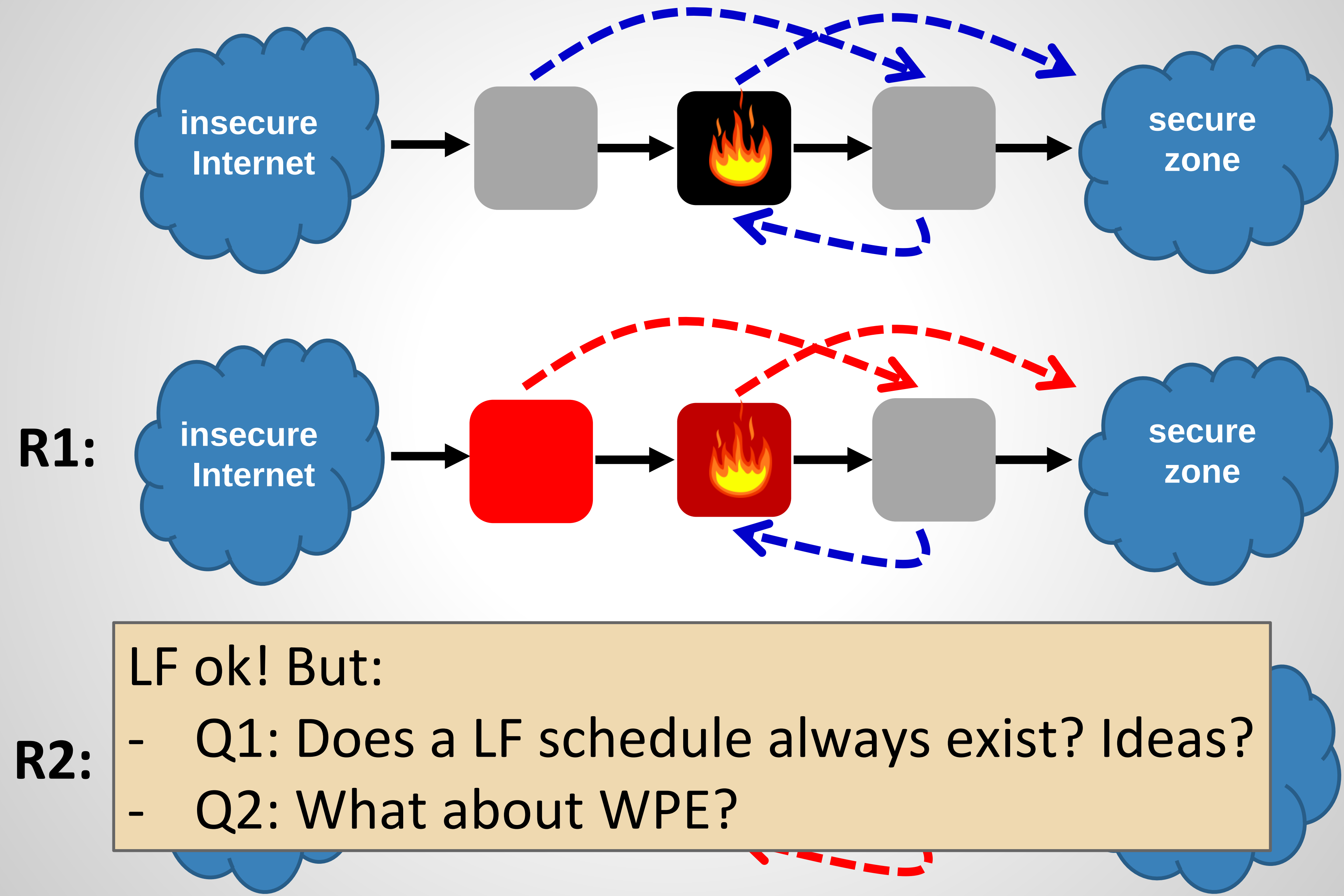
Going Back to Our Examples: LF Update!



LF ok! But:

- Q1: Does a LF schedule always exist? Ideas?

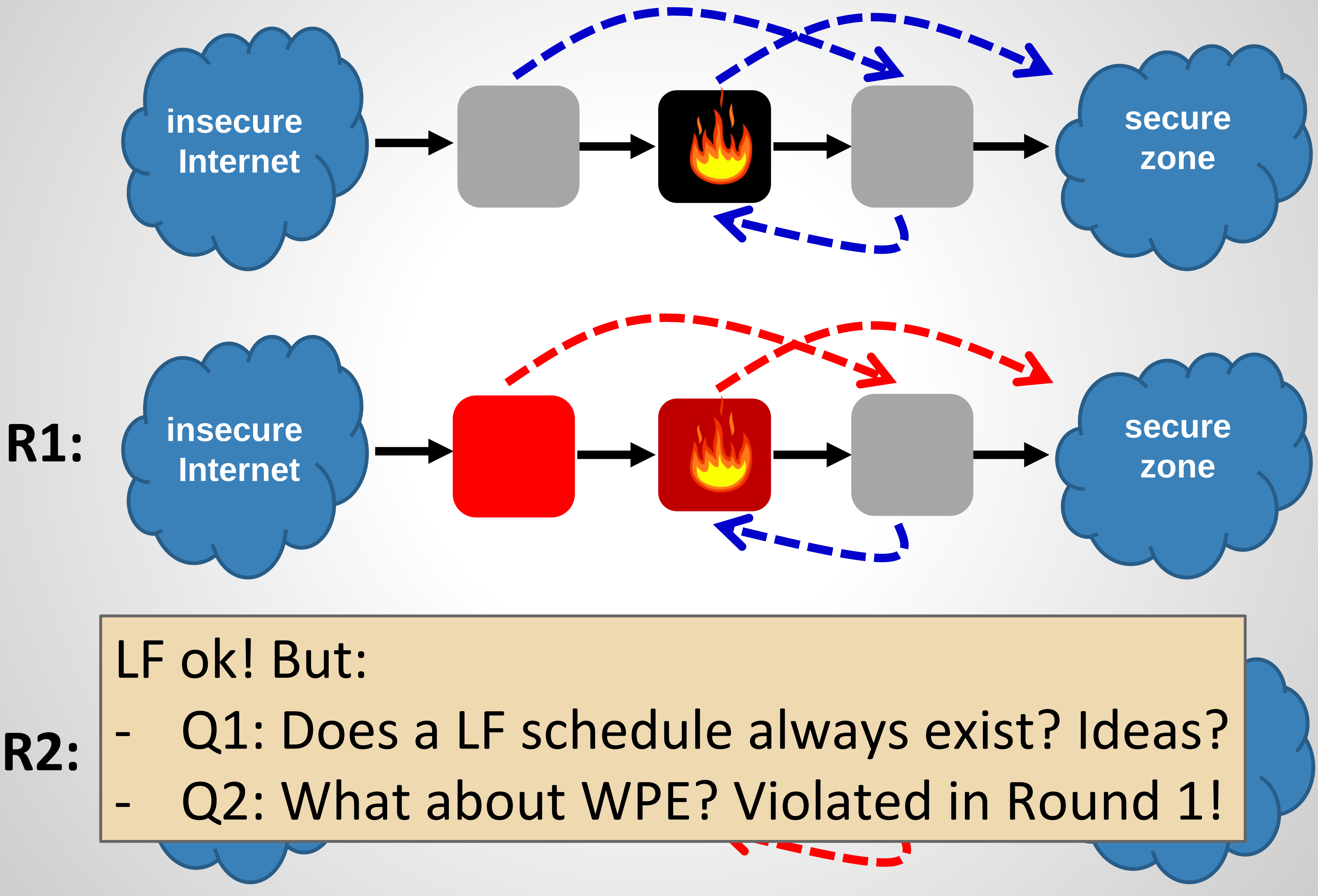
Going Back to Our Examples: LF Update!



LF ok! But:

- Q1: Does a LF schedule always exist? Ideas?
- Q2: What about WPE?

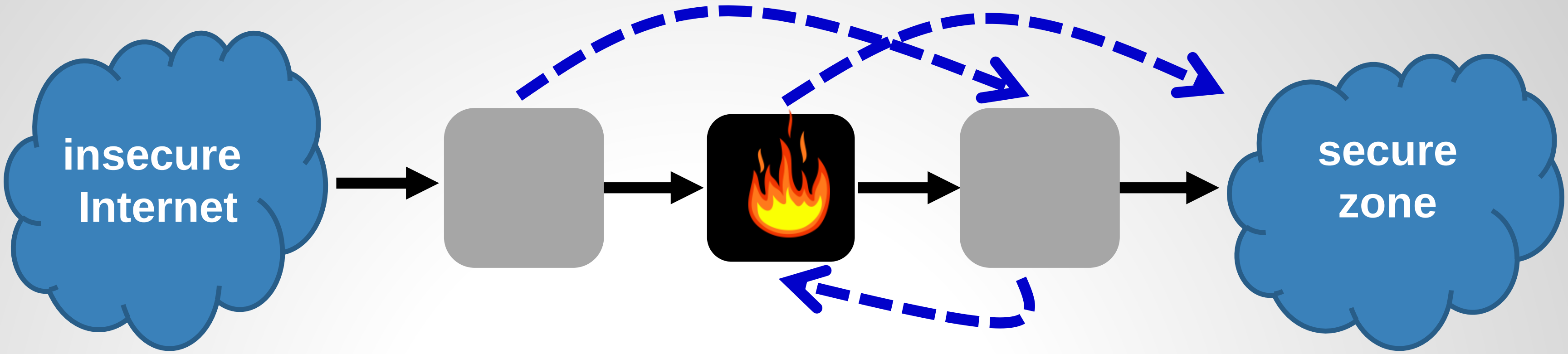
Going Back to Our Examples: LF Update!



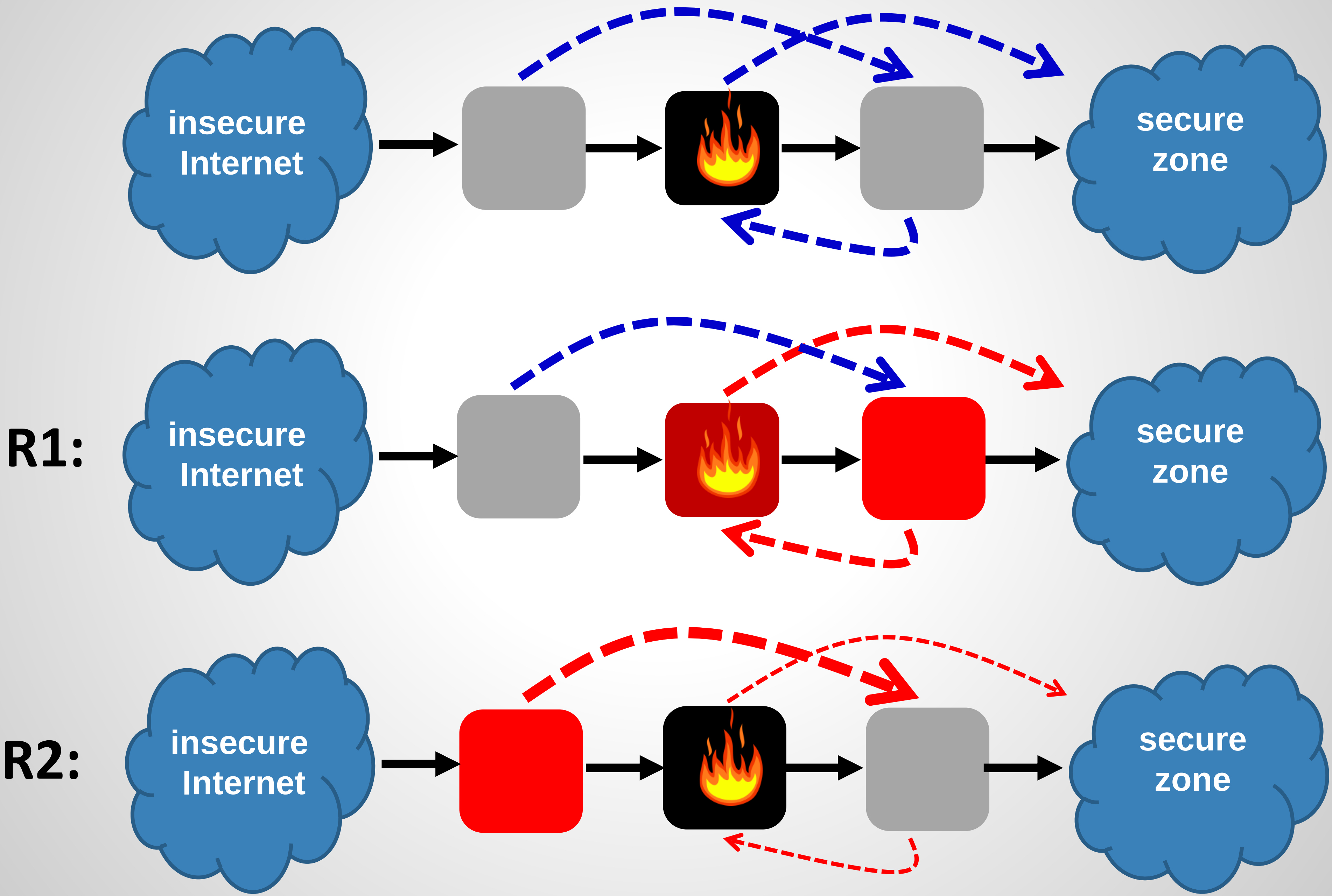
LF ok! But:

- Q1: Does a LF schedule always exist? Ideas?
- Q2: What about WPE? Violated in Round 1!

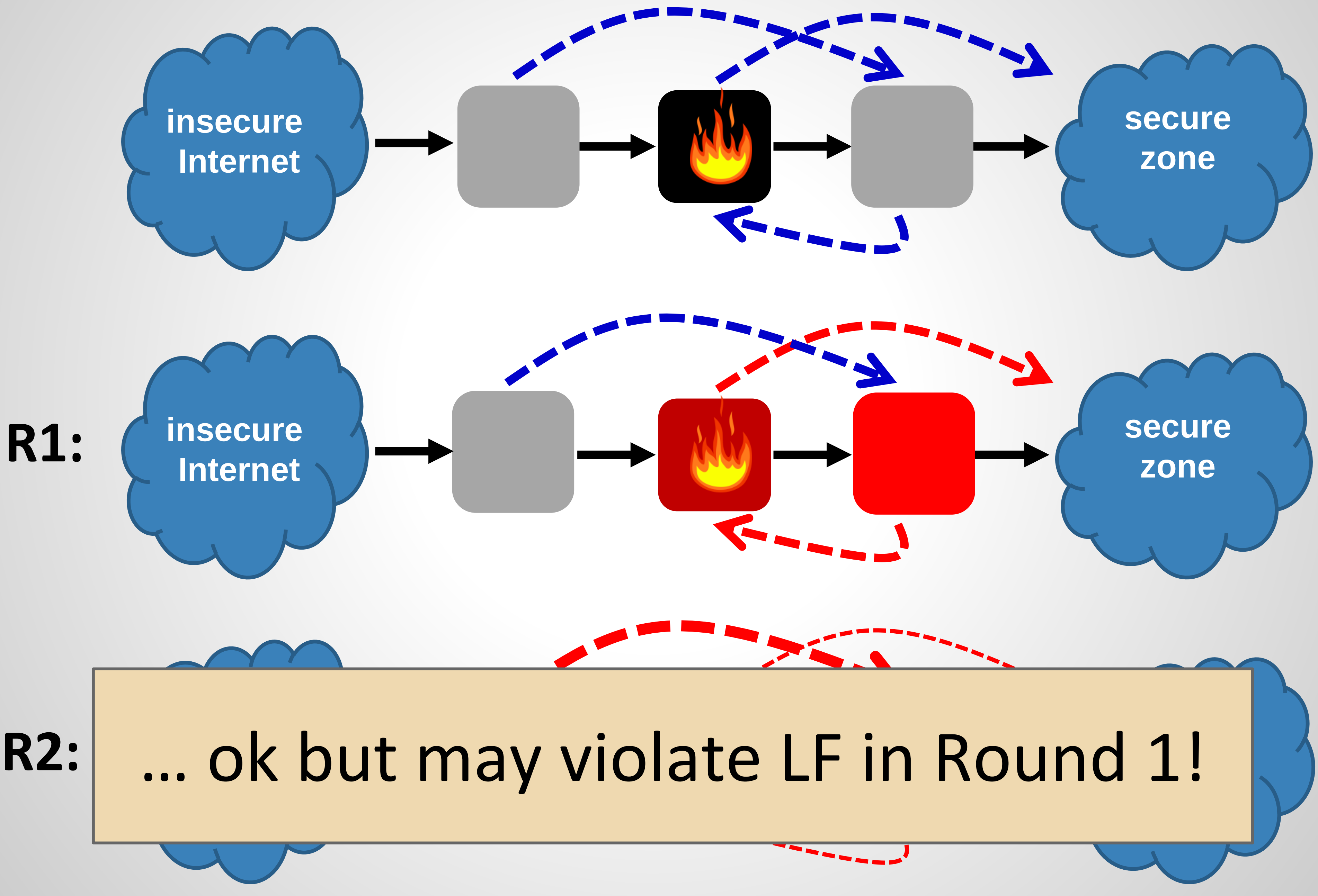
Going Back to Our Examples: WPE Update?



Going Back to Our Examples: WPE Update!

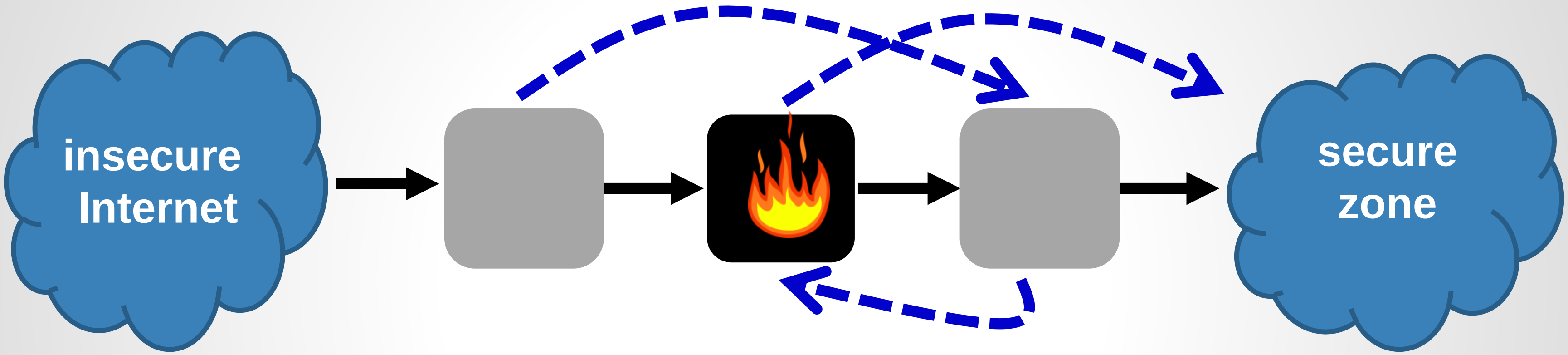


Going Back to Our Examples: WPE Update!

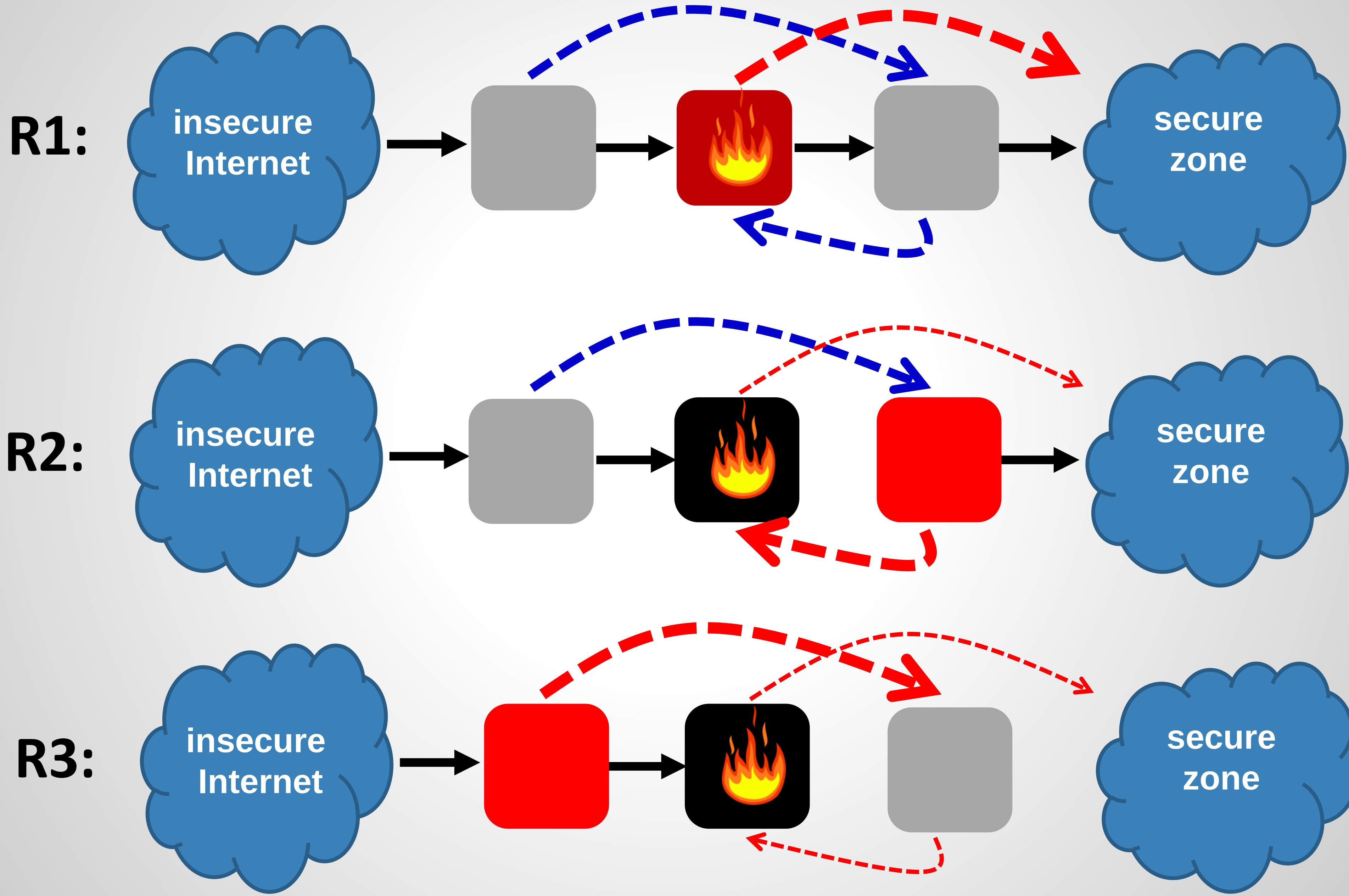


... ok but may violate LF in Round 1!

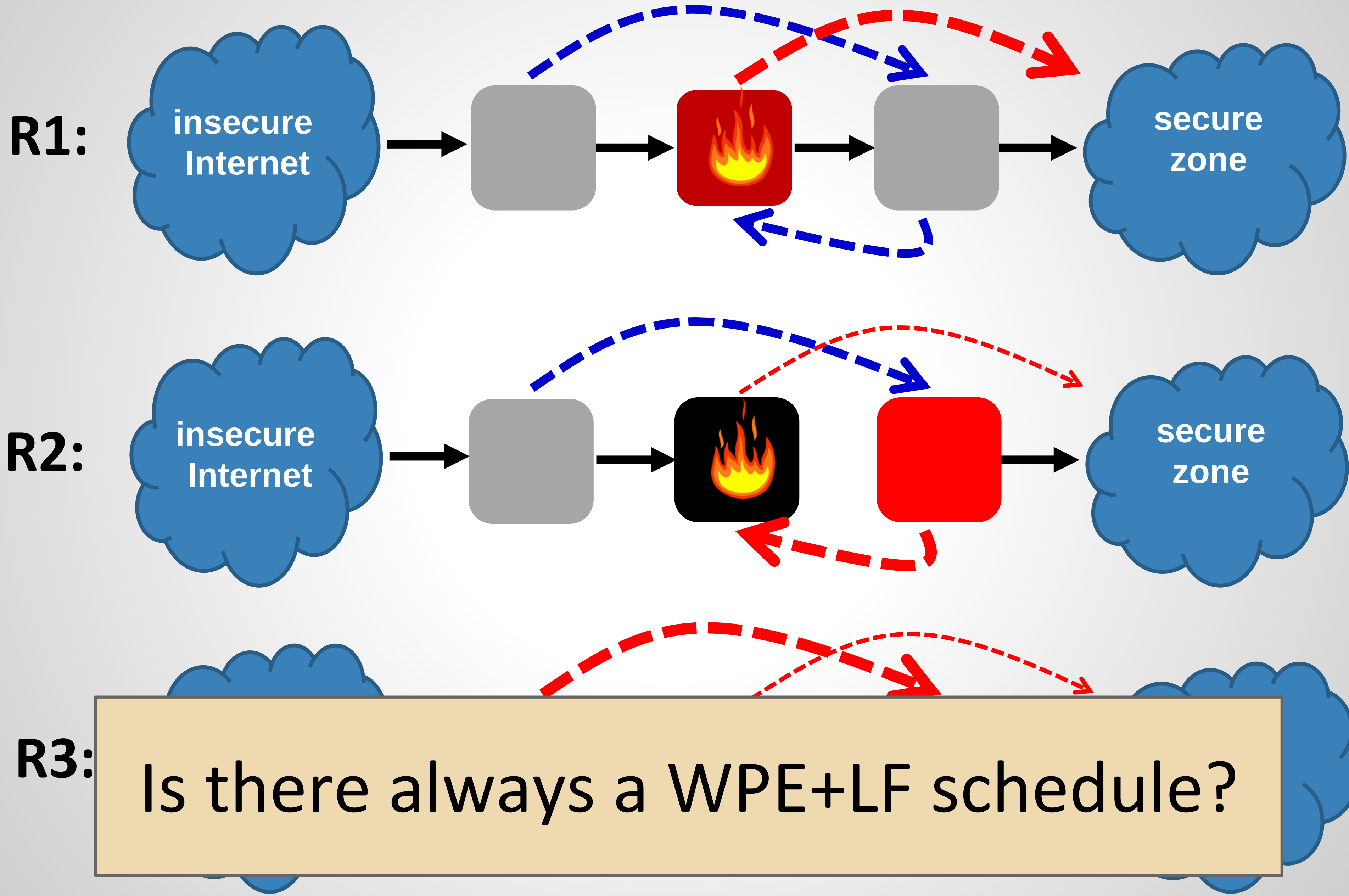
Going Back to Our Examples: Both WPE+LF?



Going Back to Our Examples: WPE+LF!

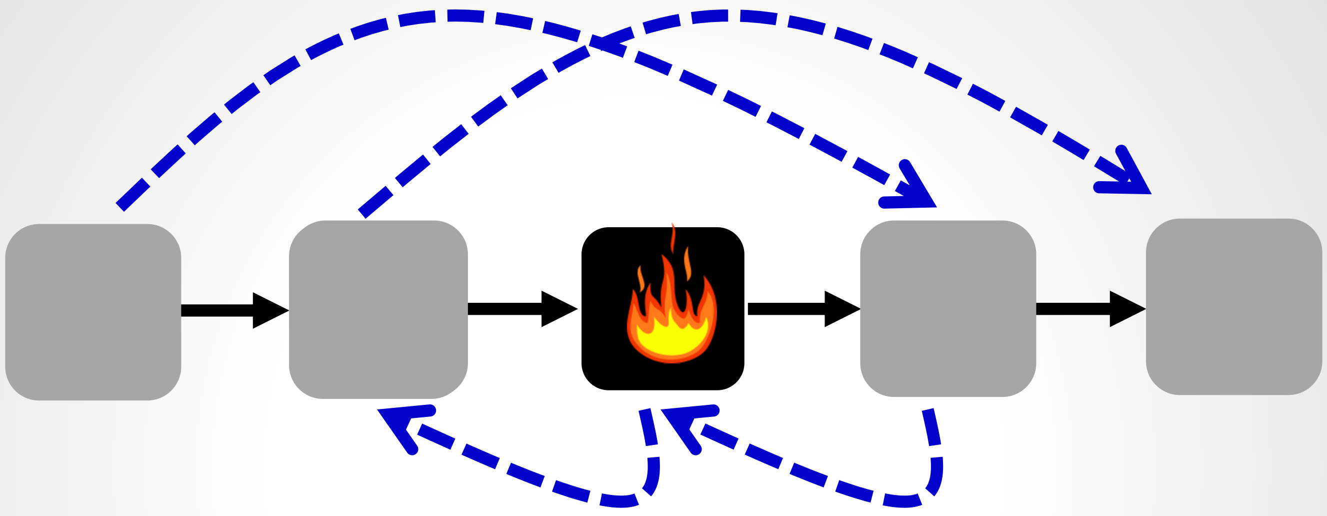


Going Back to Our Examples: WPE+LF!

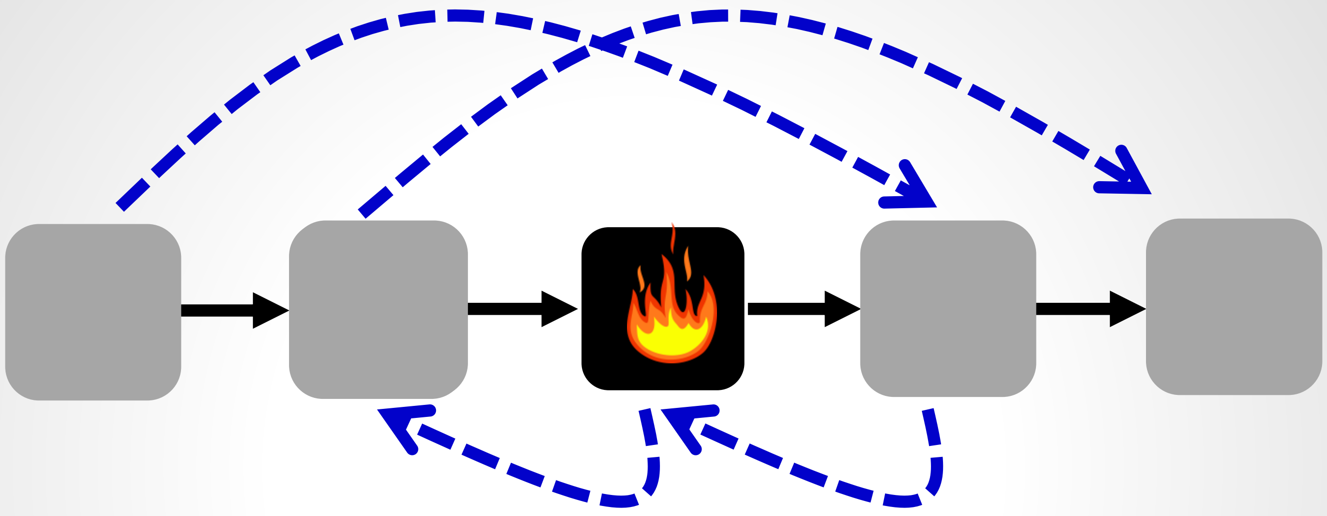


Is there always a WPE+LF schedule?

What about this one?



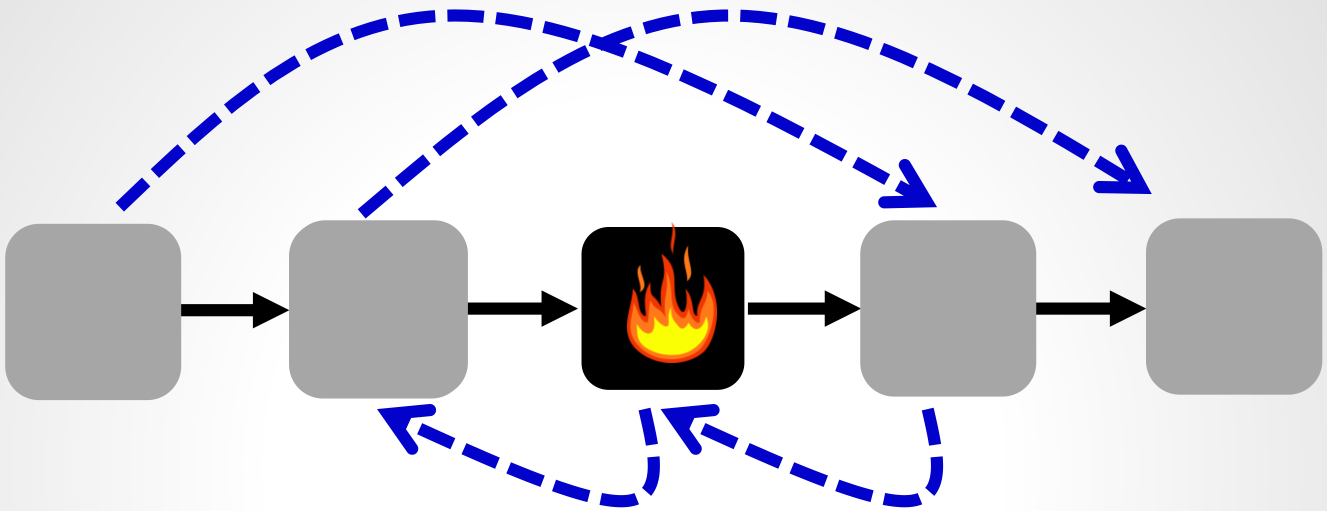
LF and WPE may conflict!



- ❑ Cannot update any forward edge in R1: WP
- ❑ Cannot update any backward edge in R1: LF

No schedule exists!

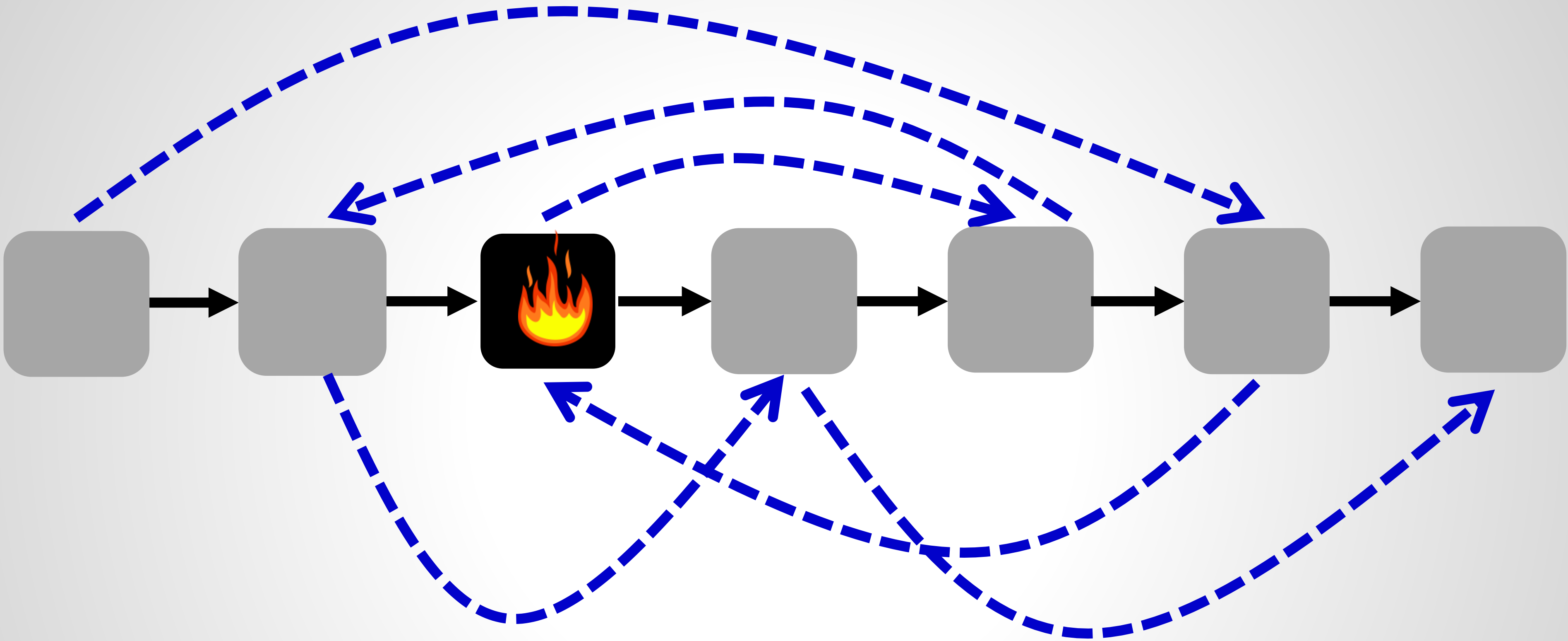
LF and WPE may conflict!



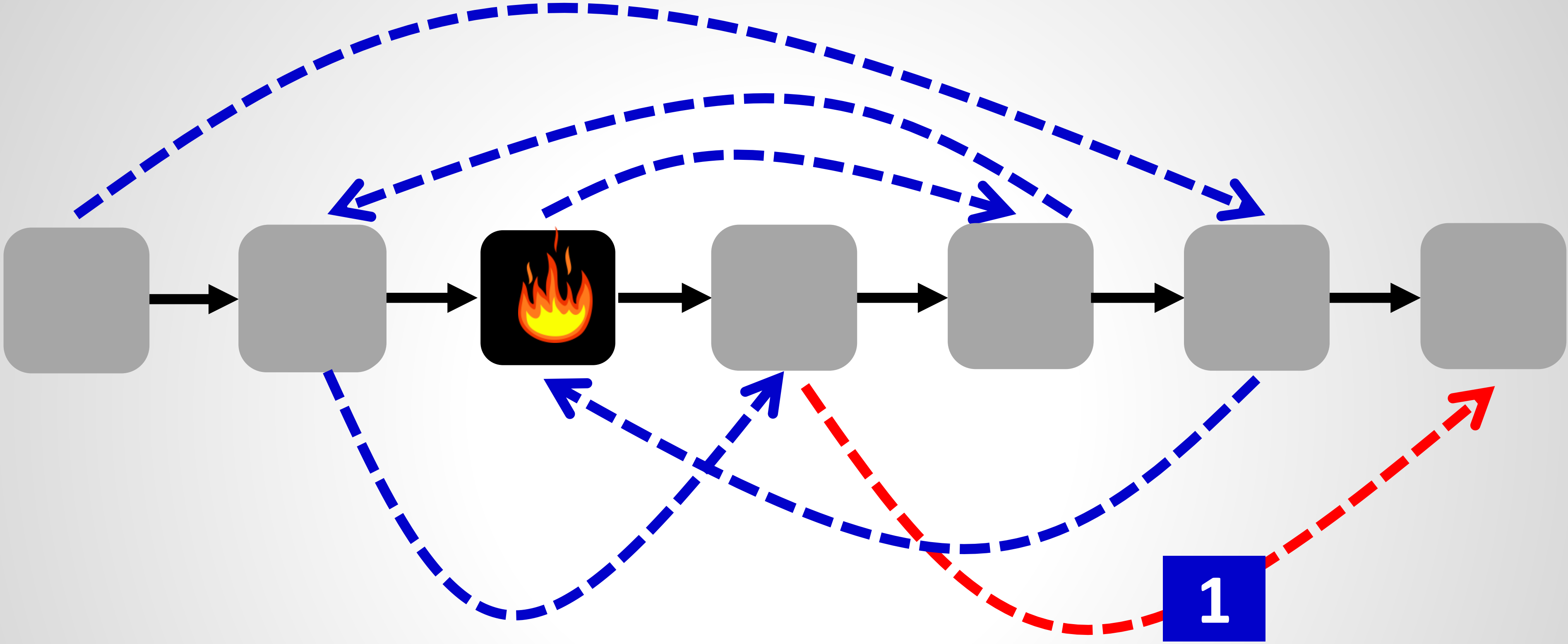
- ❑ Cannot update any forward edge in R1: WP
- ❑ Cannot update any backward edge in R1: LF

Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies
Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid.
13th ACM Workshop on Hot Topics in Networks (**HotNets**), Los Angeles, California, USA, October 2014...

What about this one?

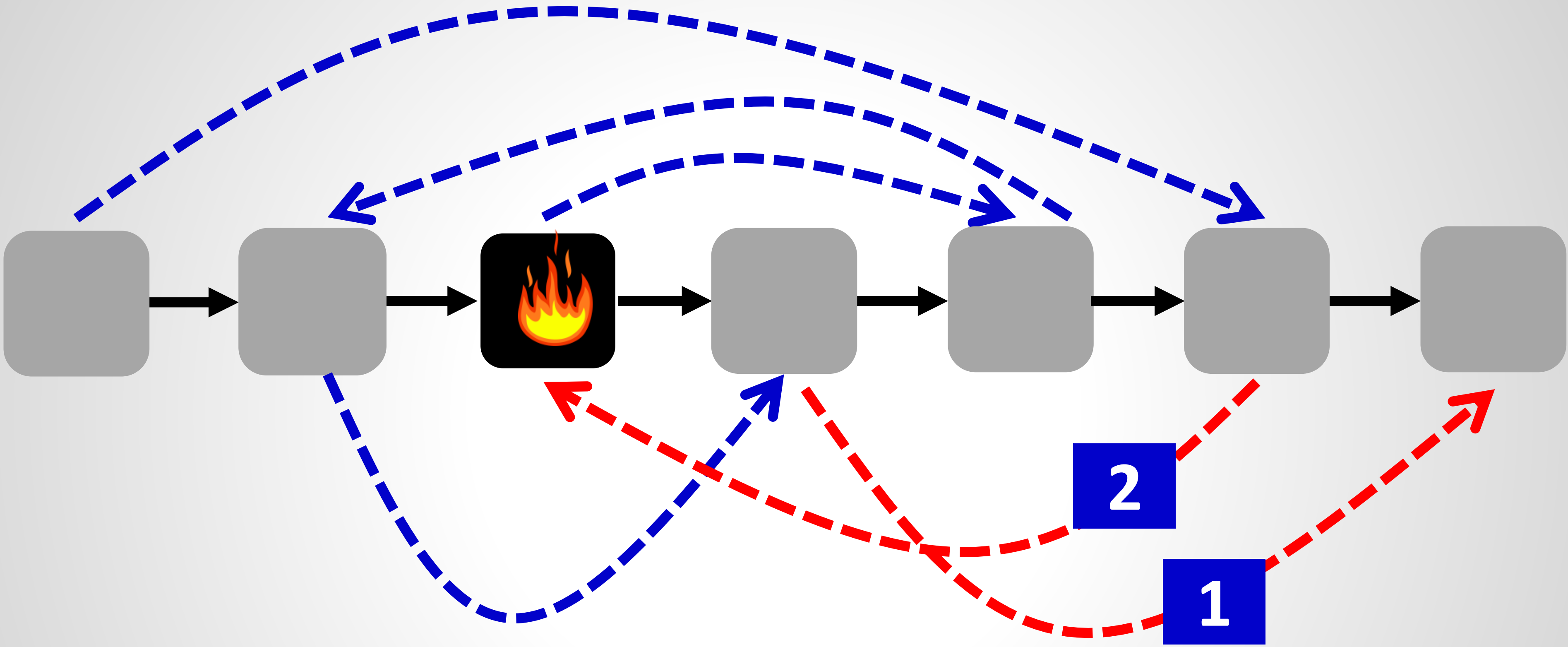


What about this one?



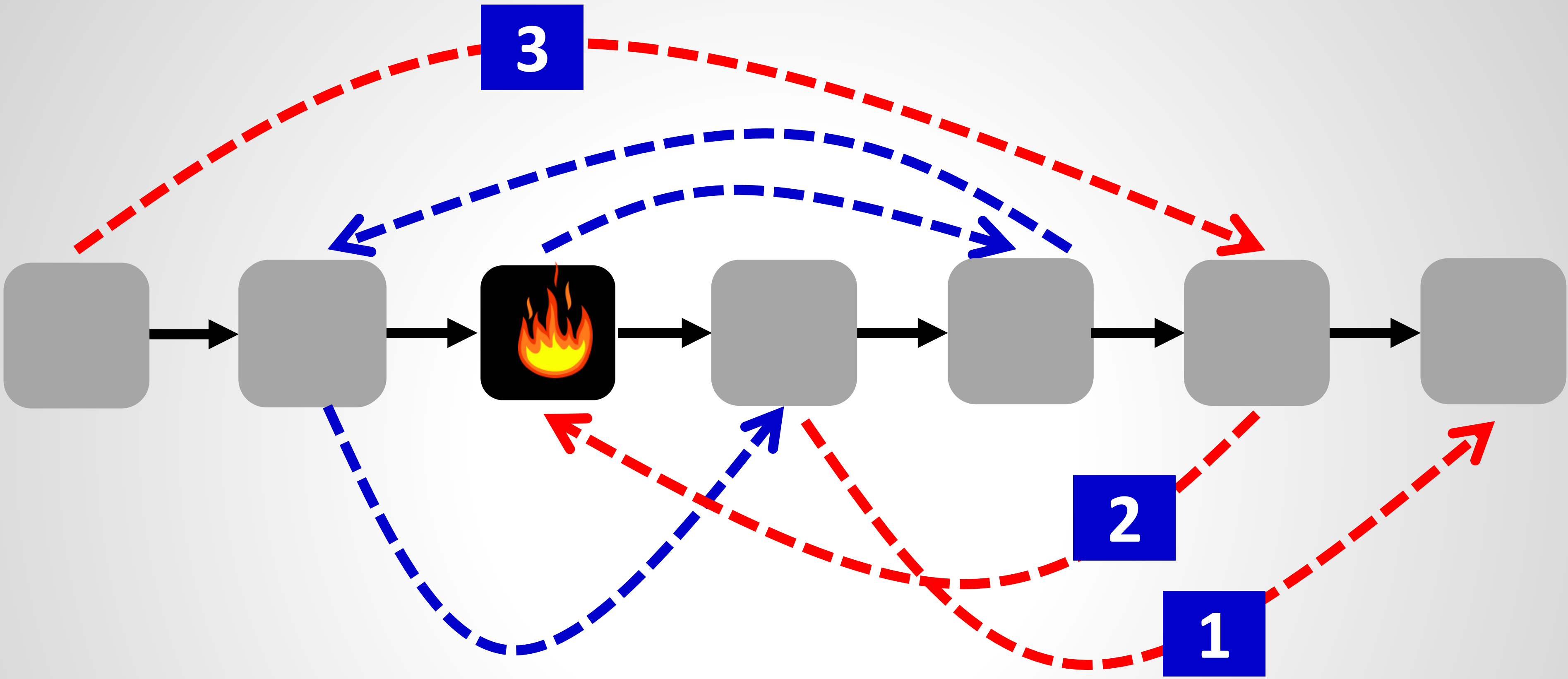
- Forward edge after the waypoint: safe!
  - No loop, no WPE violation

What about this one?



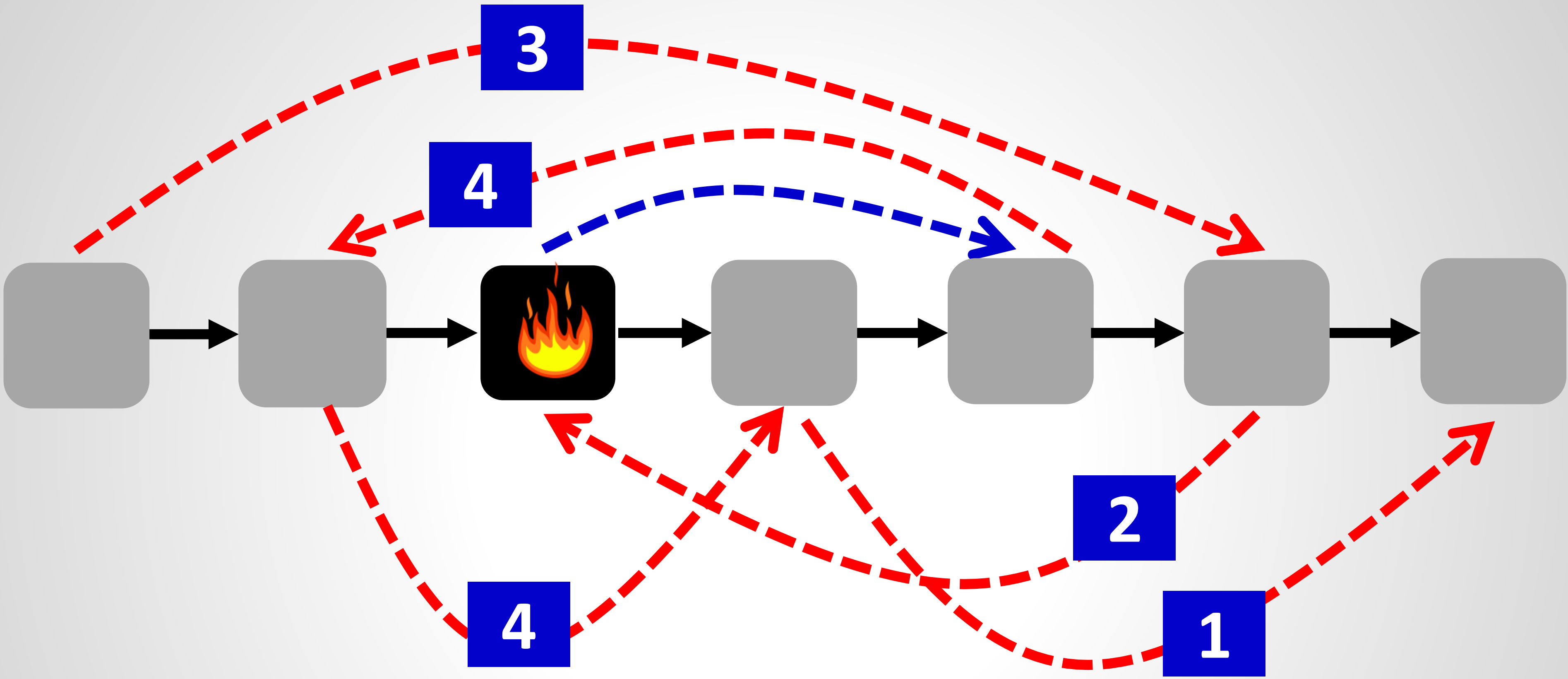
- Now this backward is safe too!
  - No loop because exit through 1

What about this one?



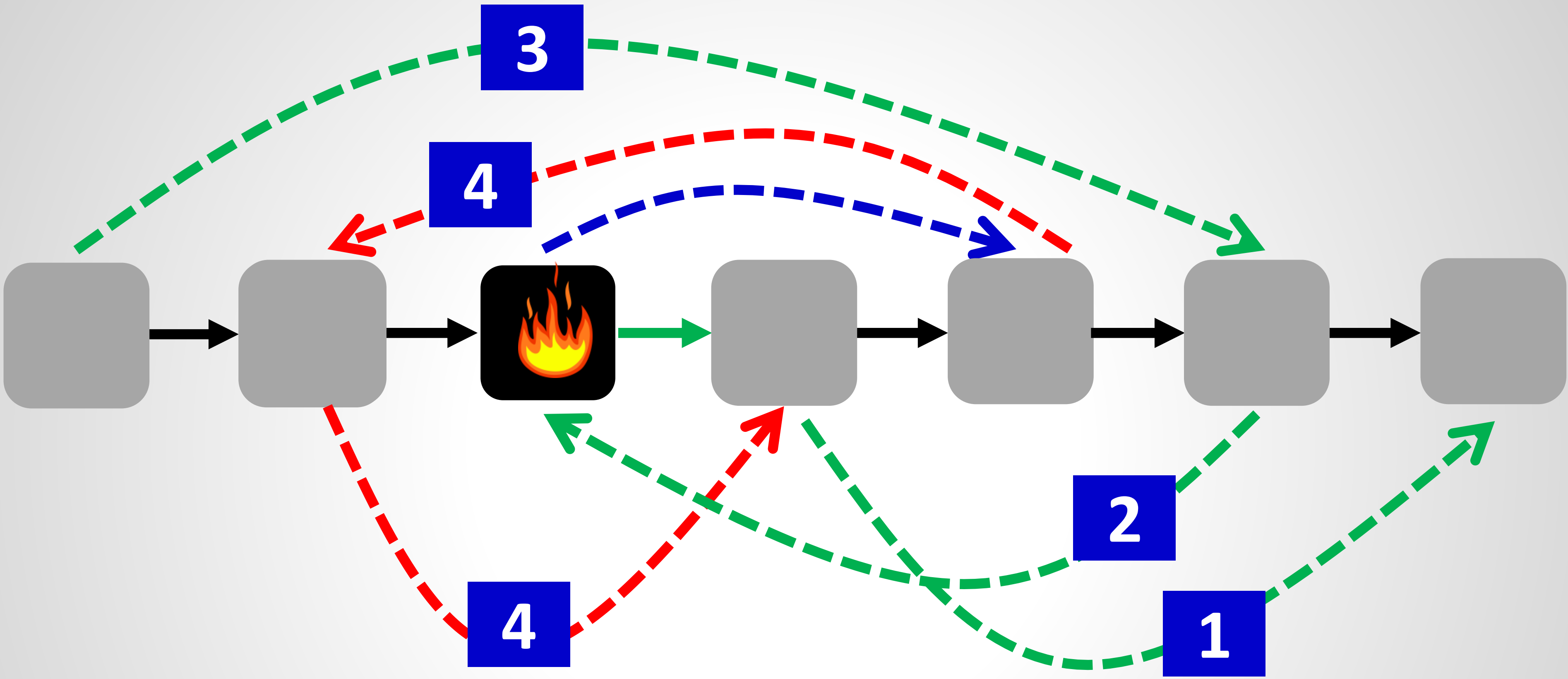
- Now this is safe: 2 ready back to WP!
  - No waypoint violation

What about this one?



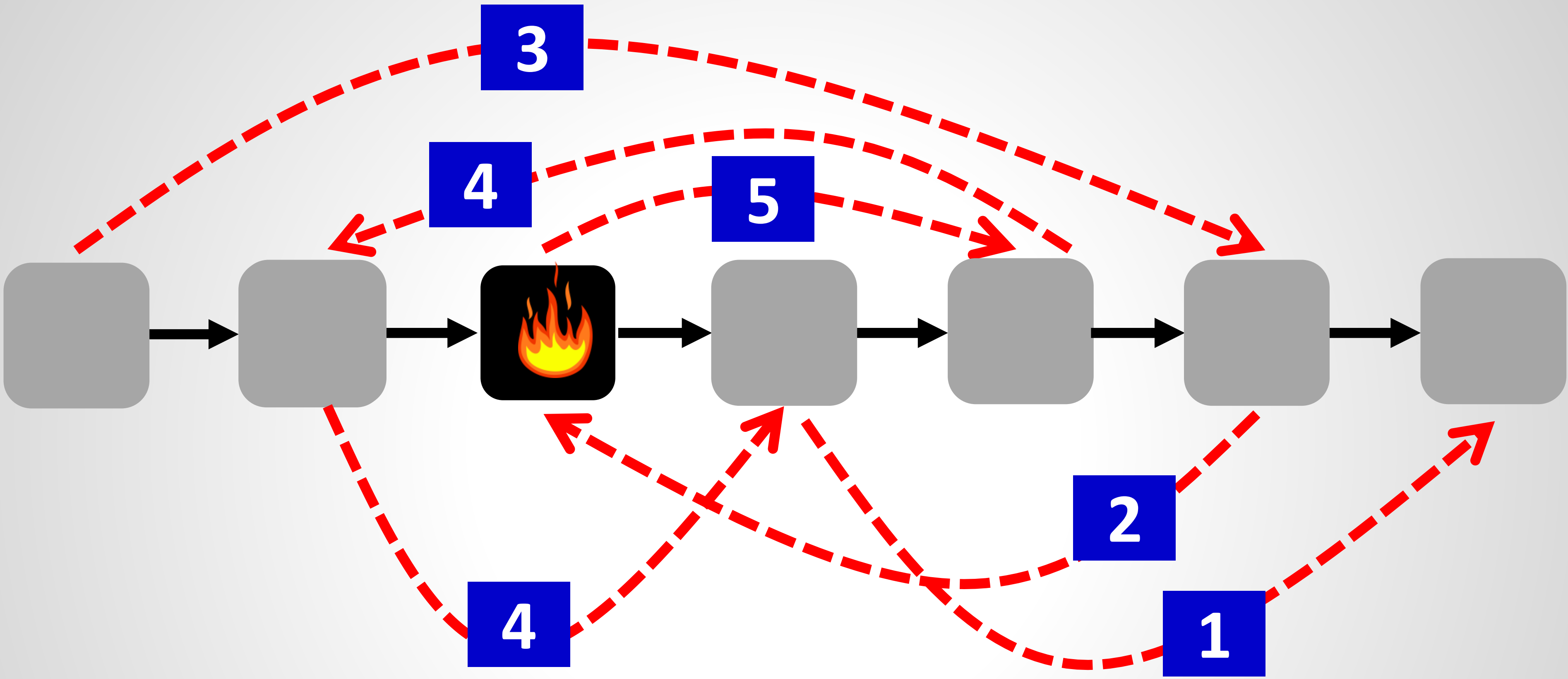
- Ok: loop-free and also not on the path (exit via 1)

What about this one?

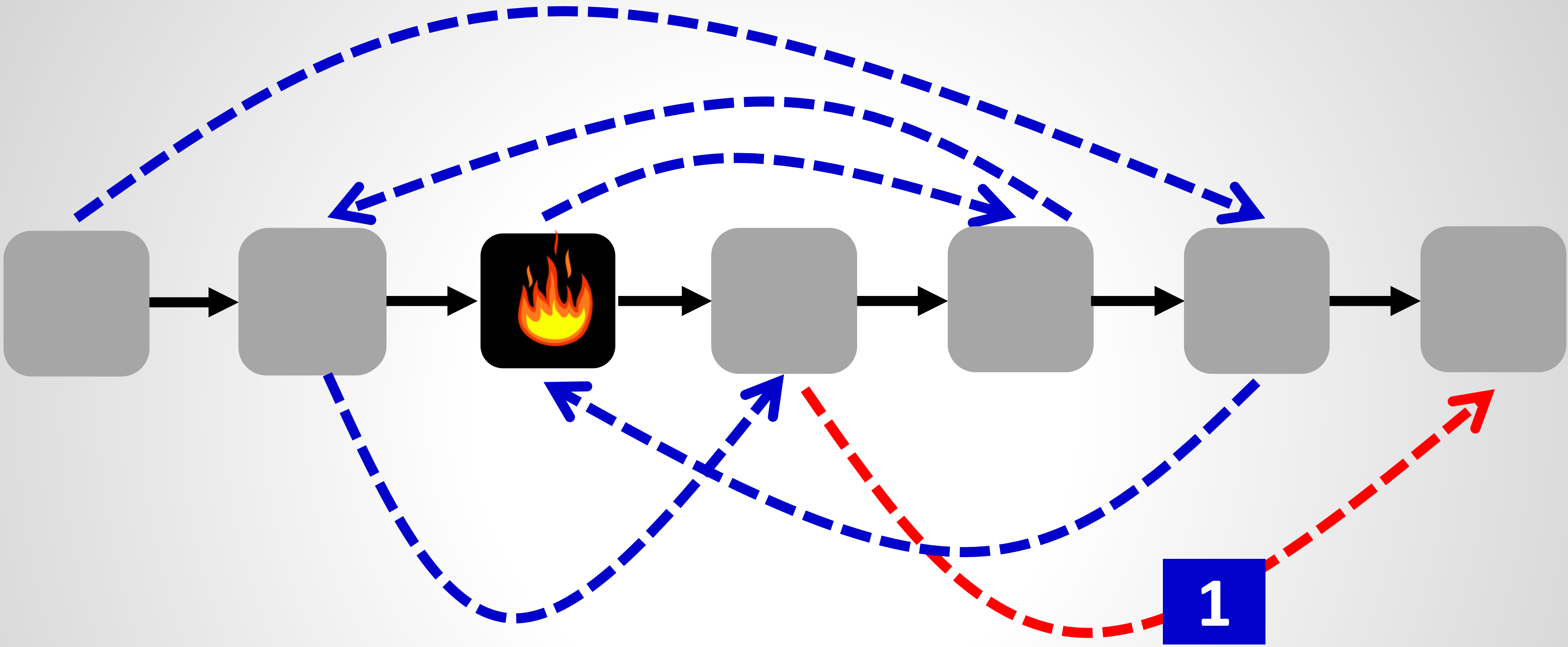


- Ok: loop-free and also not on the path (exit via 1)

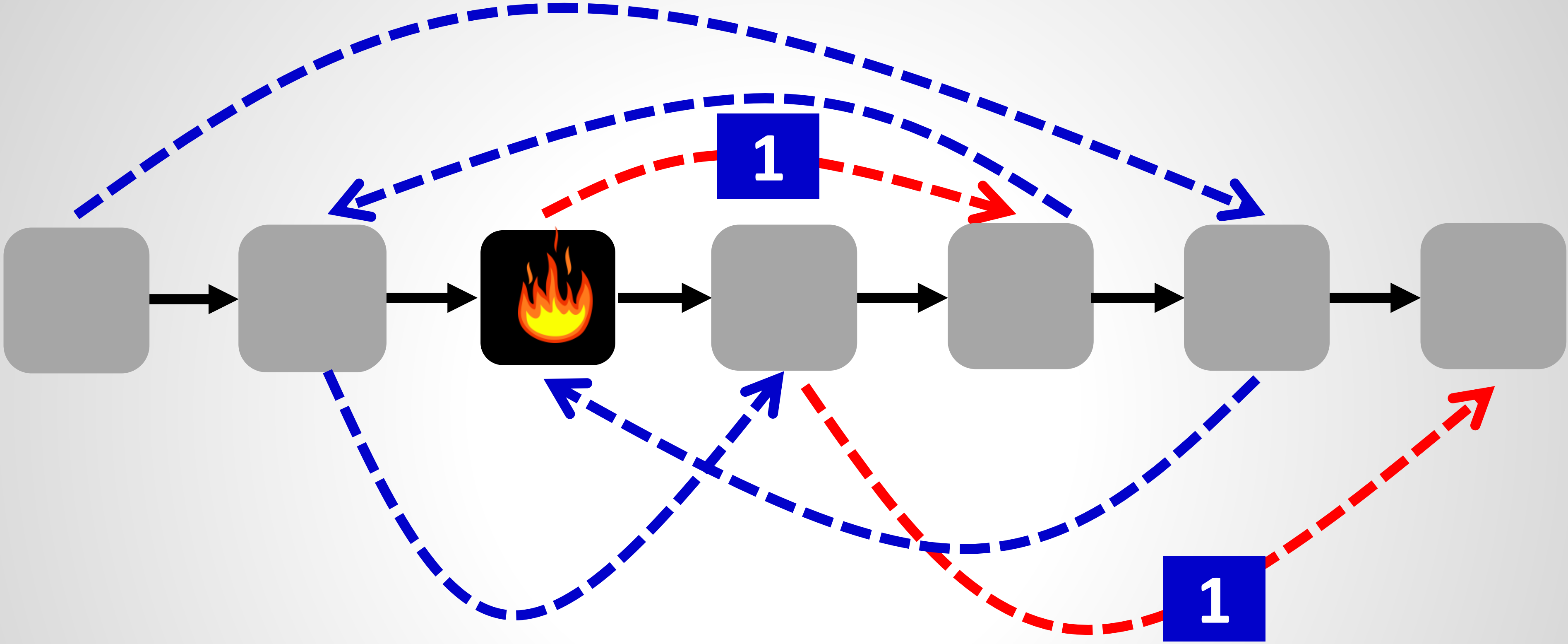
What about this one?



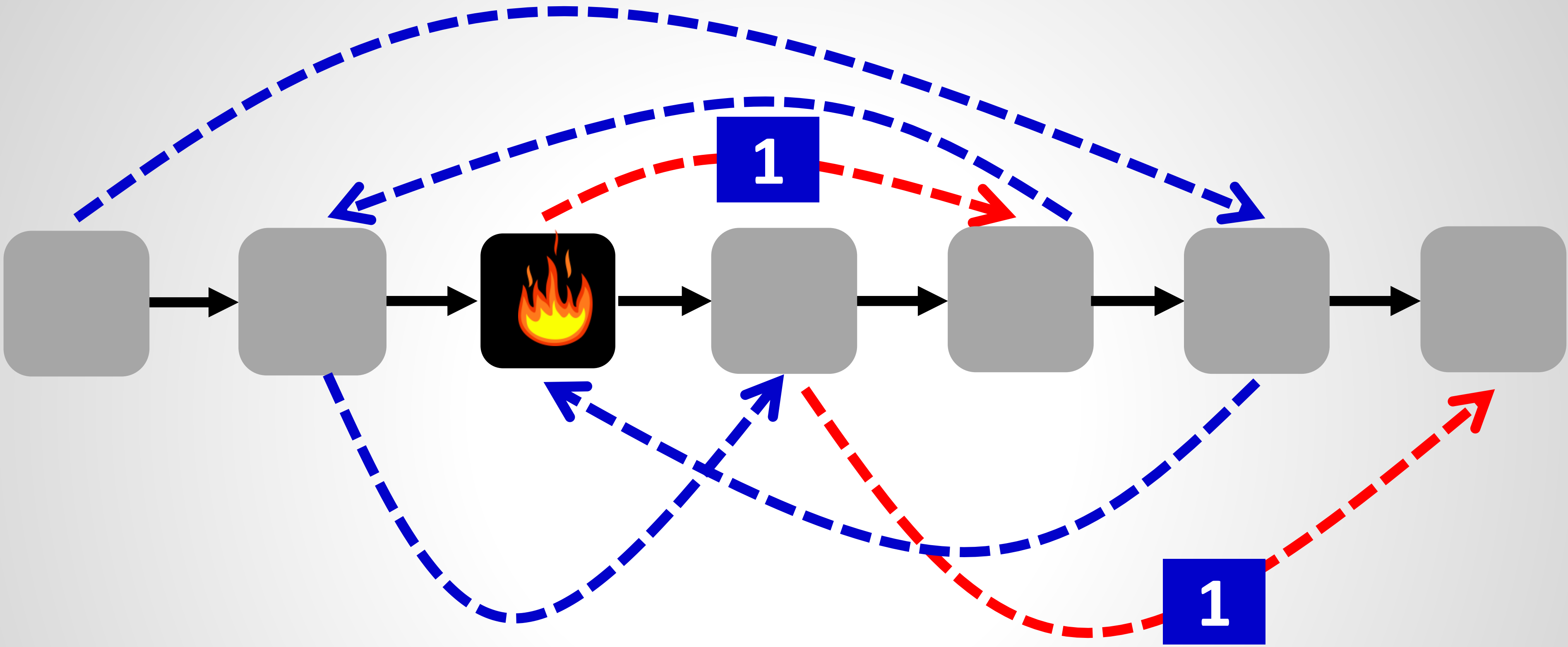
Back to the start: What if….



Back to the start: What if…. also this one?!

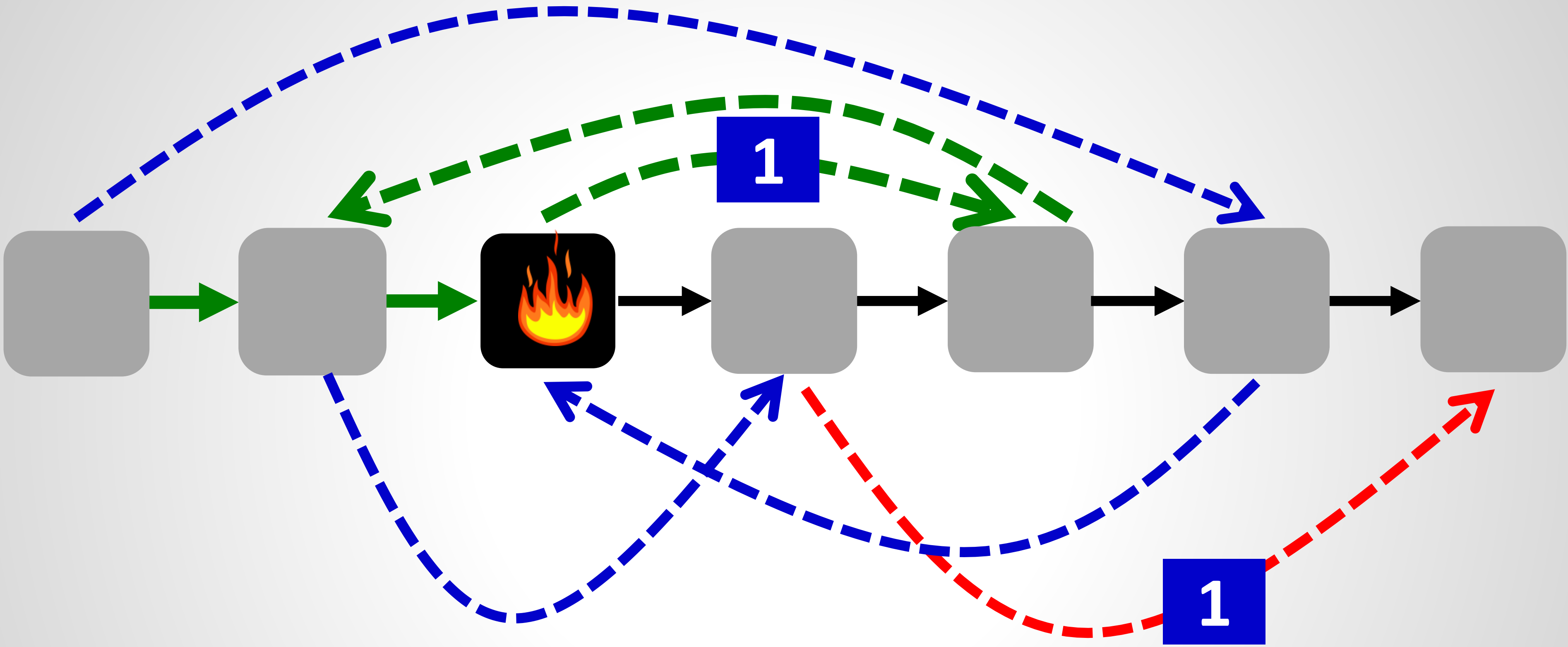


Back to the start: What if…. also this one?!



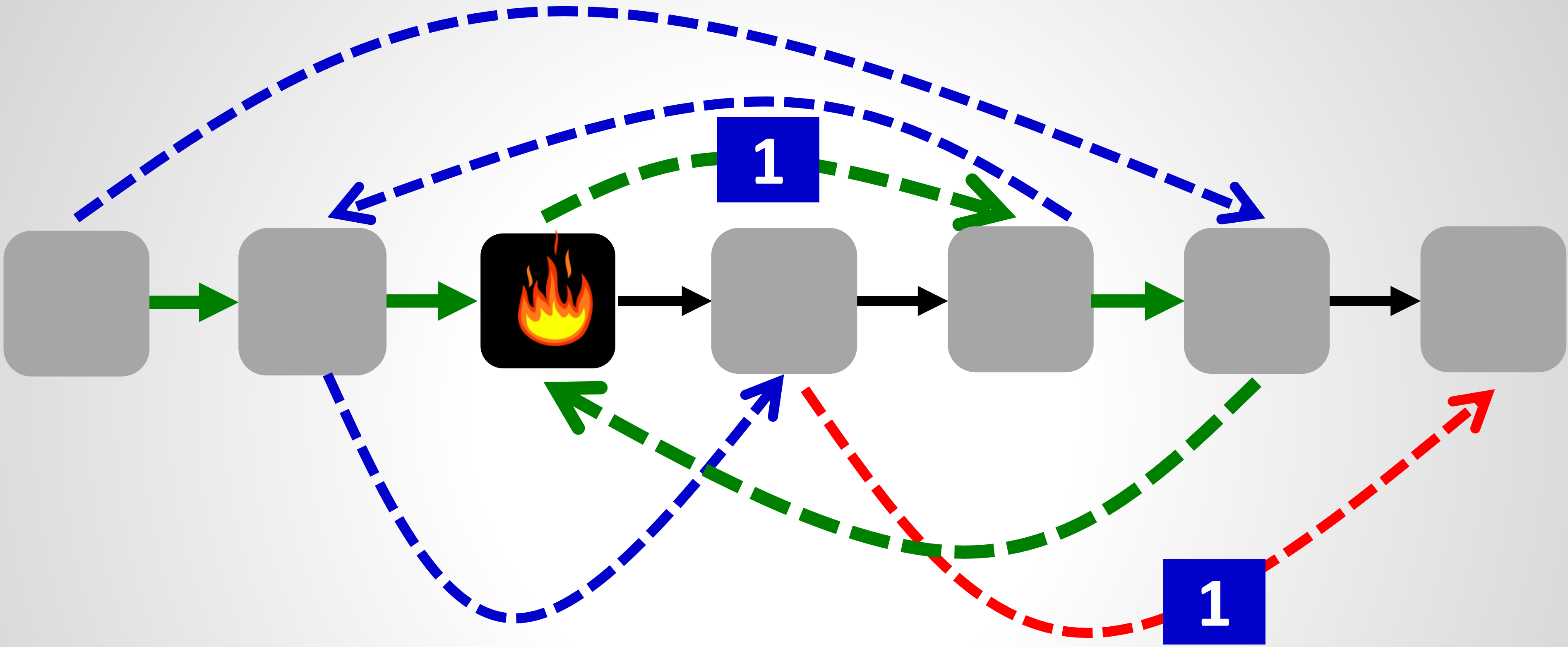
❑ Update any of the 2 backward edges? LF ☹

Back to the start: What if…. also this one?!



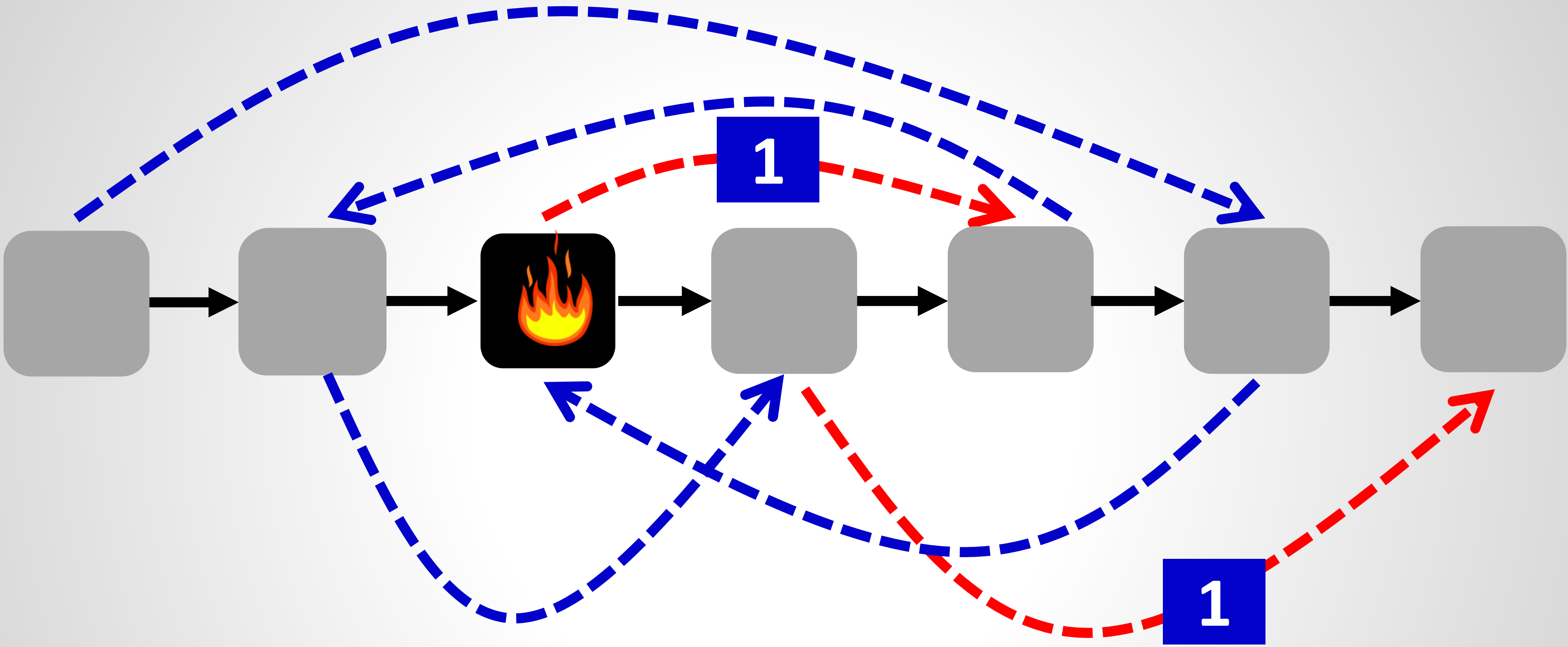
❑ Update any of the 2 backward edges? LF ☹

Back to the start: What if…. also this one?!



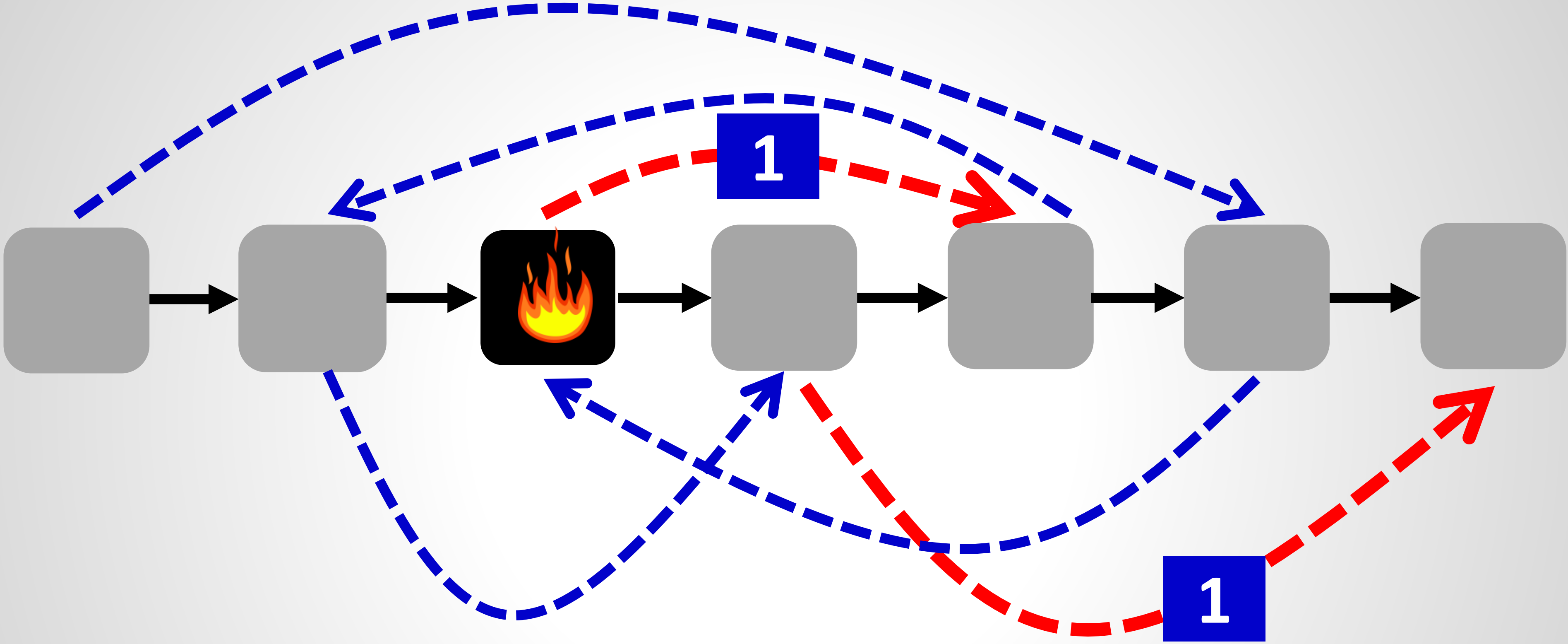
- ❑ Update any of the 2 backward edges? LF ☹

Back to the start: What if…. also this one?!

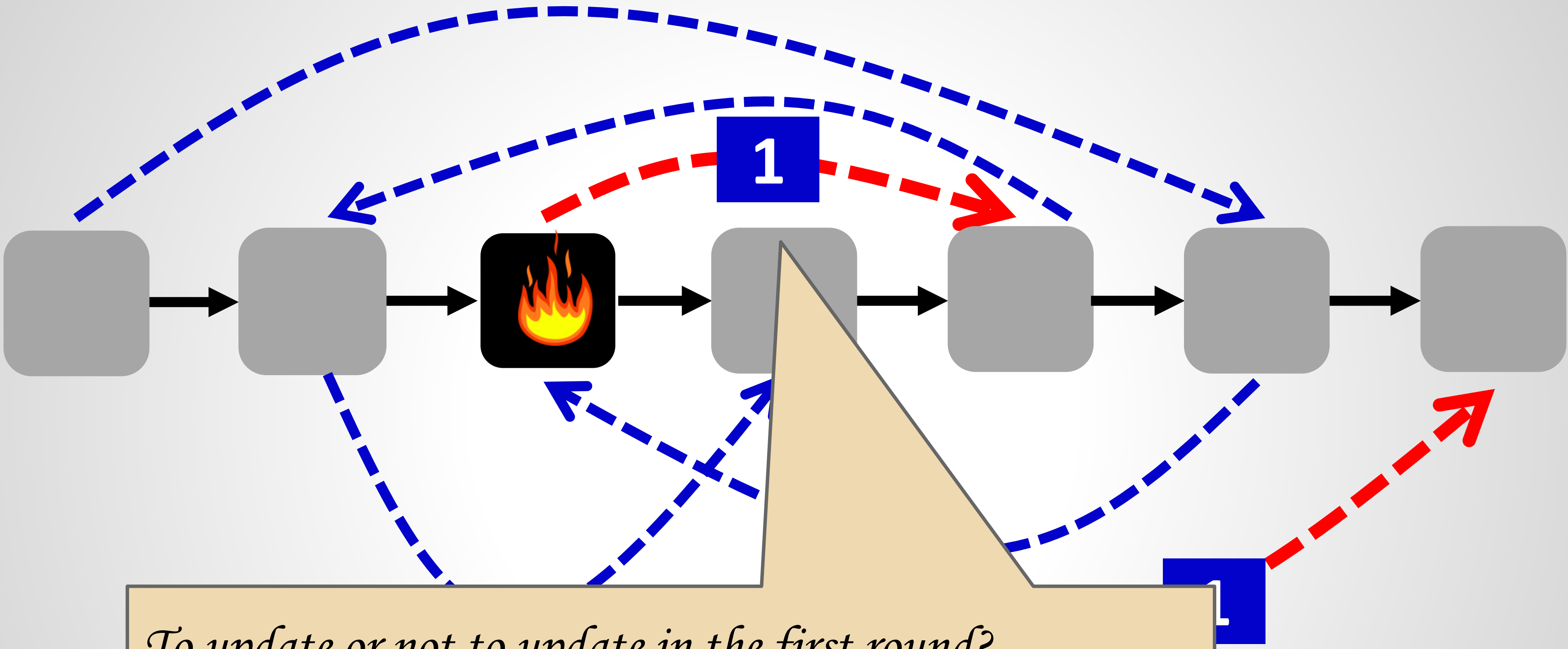


- ❑ Update any of the 2 backward edges? LF ☹
- ❑ Update any of the 2 other forward edges? WPE ☹
- ❑ What about a combination? Nope…

Back to the start: What if…. also this one?!



Back to the start: What if…. also this one?!



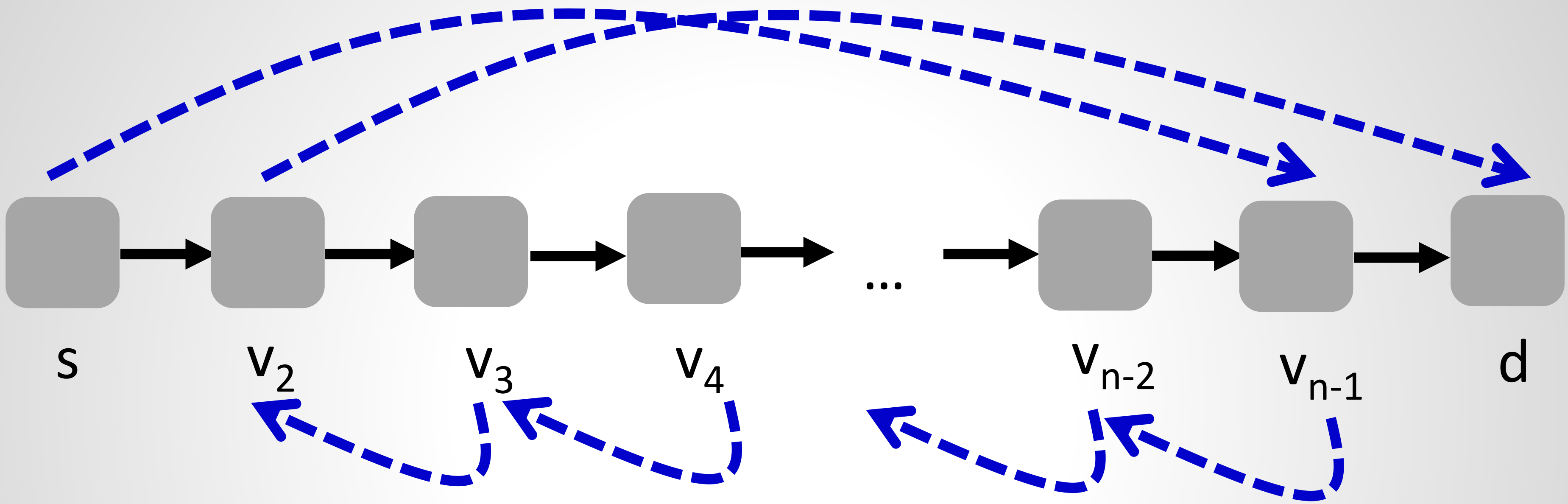
To update or not to update in the first round?
That is the question…
… which leads to NP-hardness!

What about loop-freedom only?

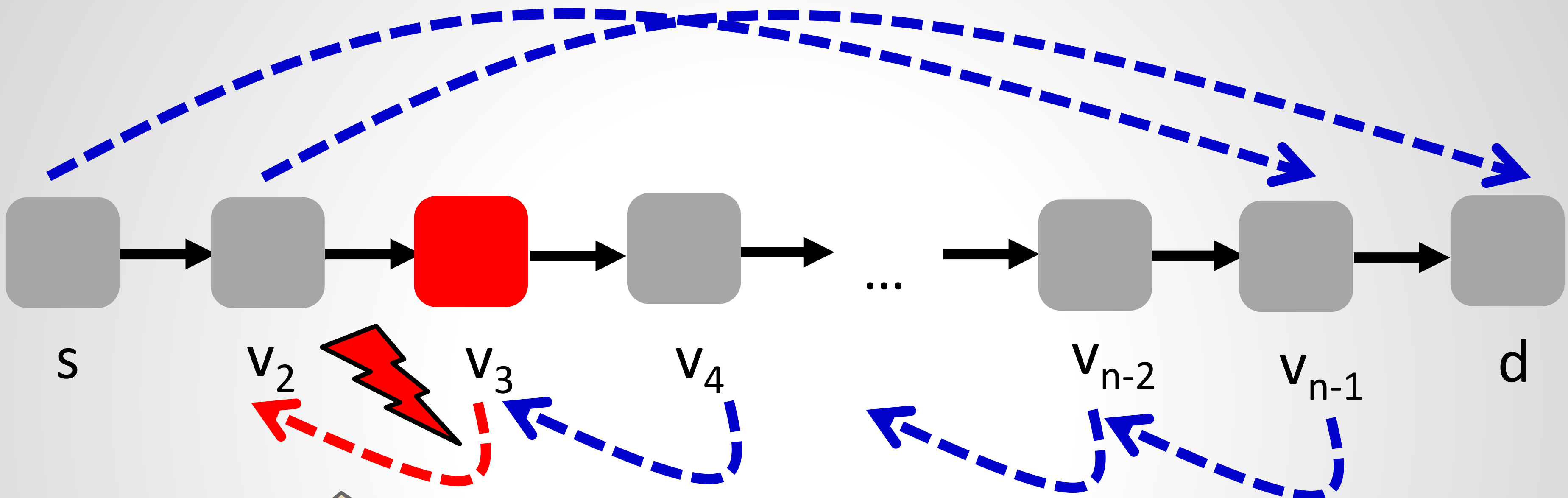
What about loop-freedom only?

Always works! How many rounds?

How to update LF?

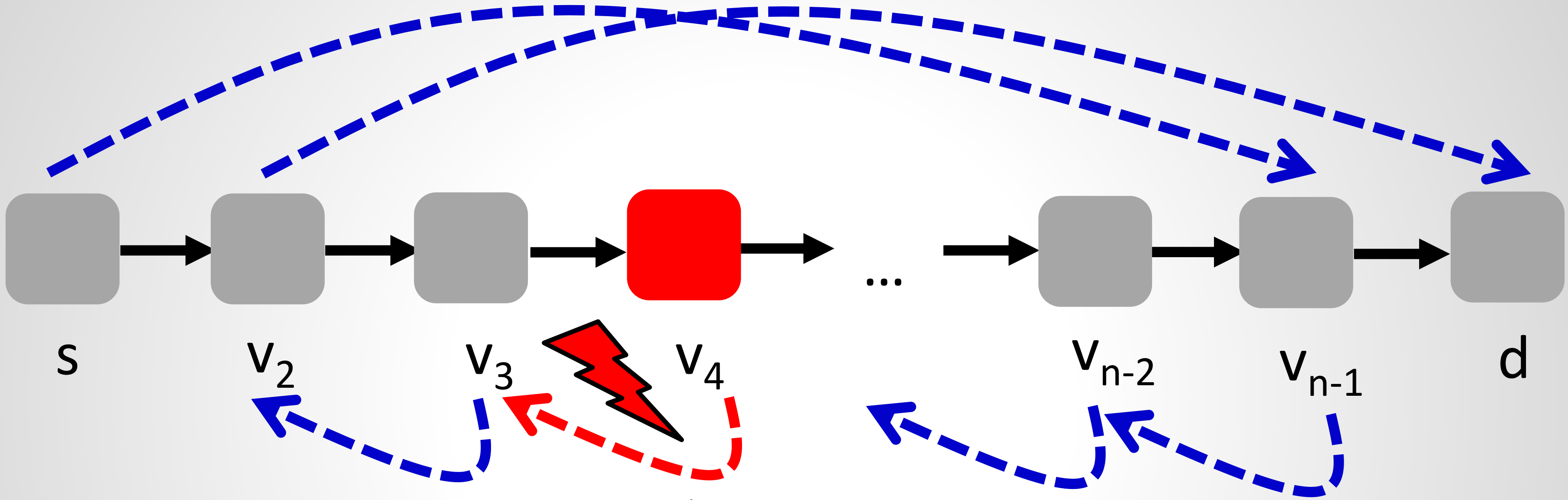


LF Updates Can Take Many Rounds!



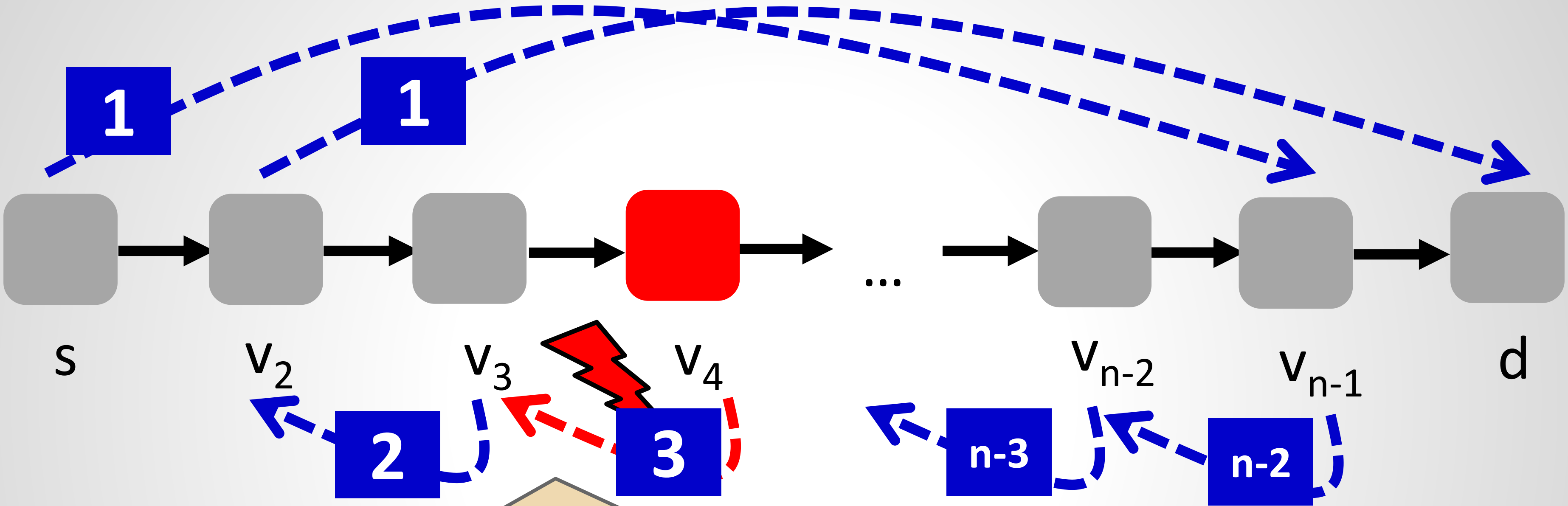
Invariant: need to update v_2 before v_3!

LF Updates Can Take Many Rounds!



Invariant: need to update v_3 before v_4!

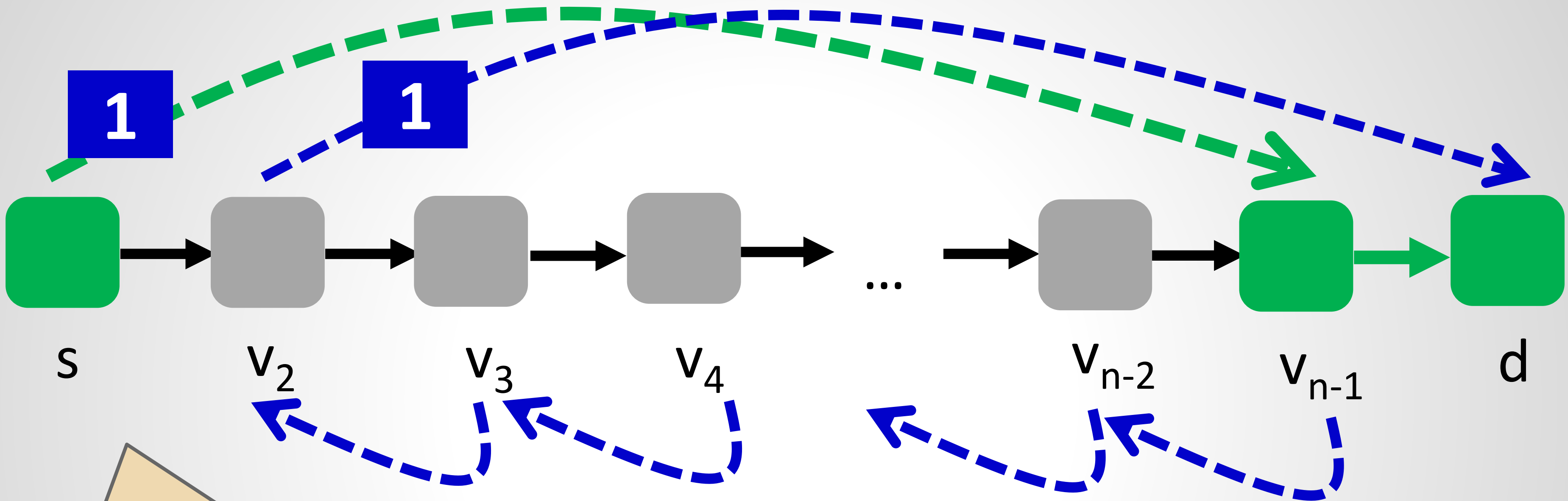
LF Updates Can Take Many Rounds!



Induction: need to update v_{i-1} before v_i (before v_{i+1} etc.)!

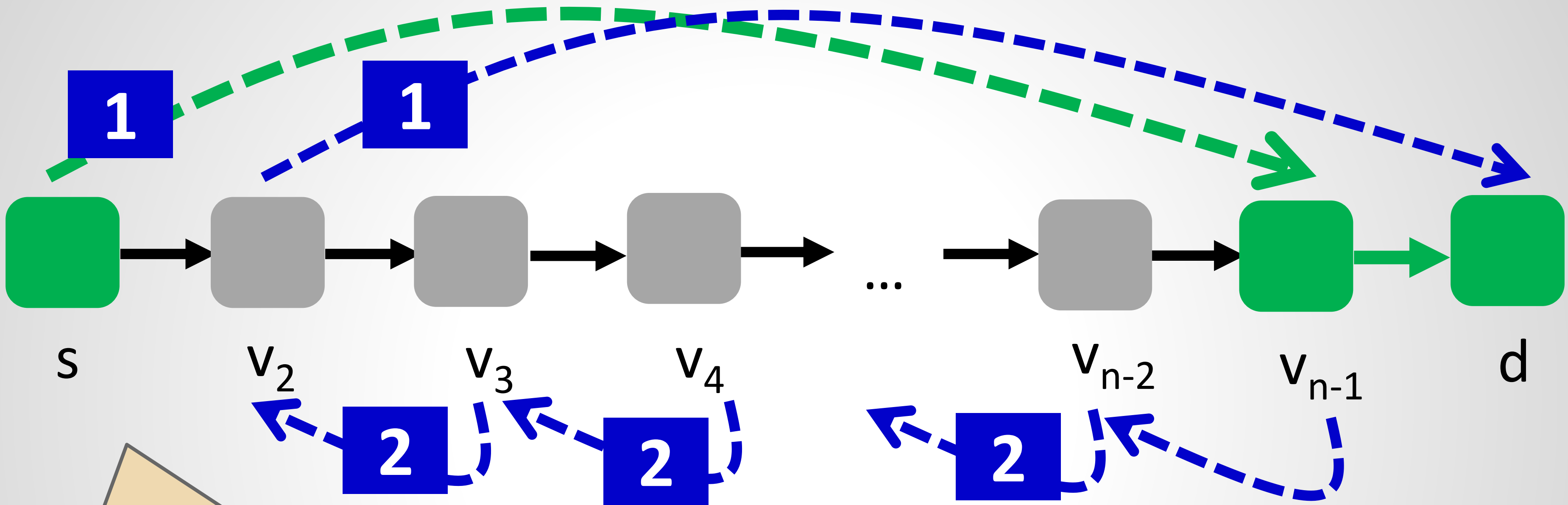
Ω(n) rounds?! In principle, yes…: Need a path back out before updating backward edge!

It is good to relax!



But: If s has been updated, nodes not on (s,d)-path!

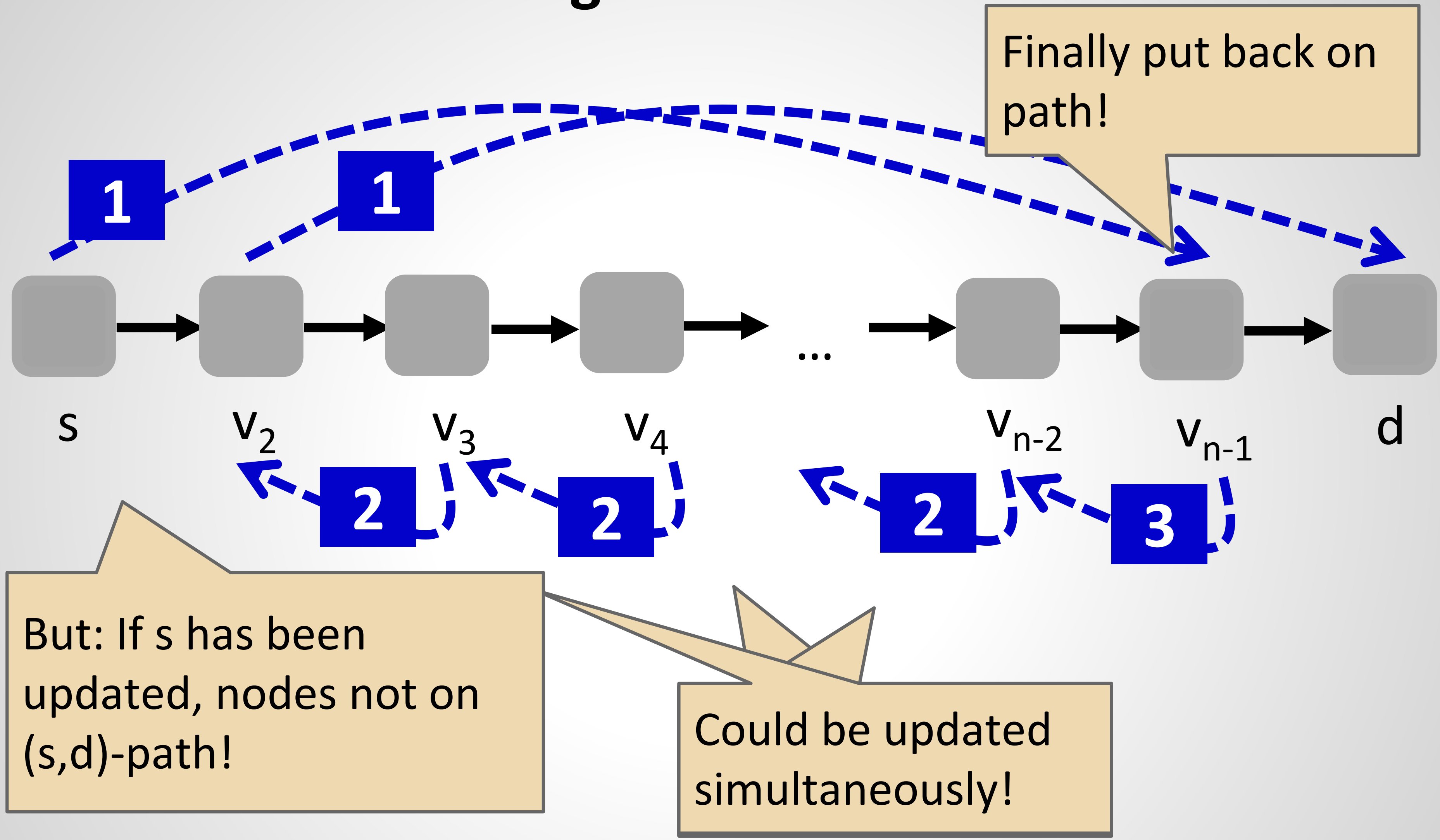
It is good to relax!



But: If s has been updated, nodes not on (s,d)-path!

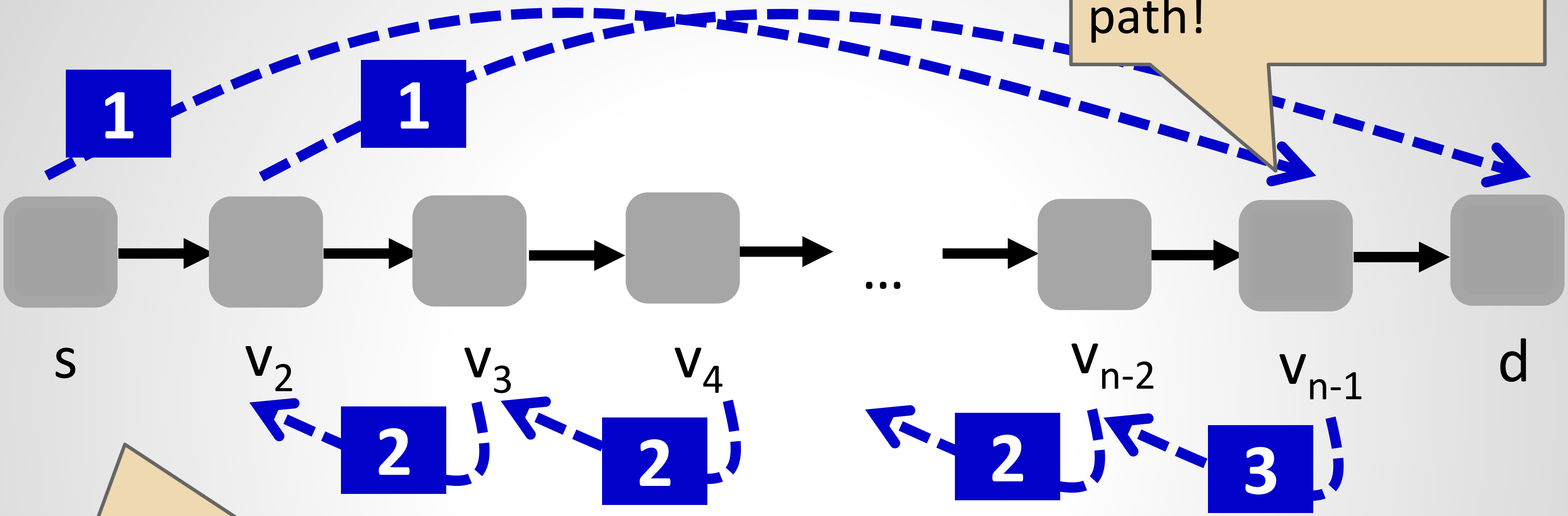
Could be updated simultaneously!

It is good to relax!



It is good to relax!

Finally put back on path!



But: If s has been updated, nodes not on (s,d)-path!

Could be updated simultaneously!

3 rounds only!

Takeaways so far

- ❑ Strong (topological) loop-free update may take Ω (n) rounds
- ❑ Relaxed loop-free schedules may be Ω (n) times faster

Questions

- ❑ Strong loop-freedom: Can we compute optimal schedules?
- ❑ Relaxed loop-freedom: Are O(1) rounds always enough?

Takeaways so far

- Strong (topological) loop-free update may take Ω (n) rounds
- Relaxed loop-free schedules may be Ω (n) times faster

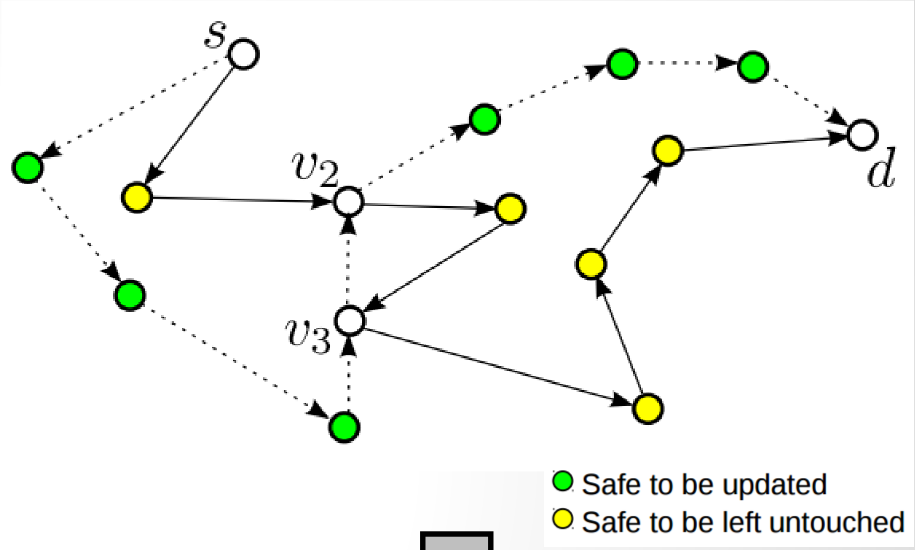
Generally NP-hard and greedy is bad.

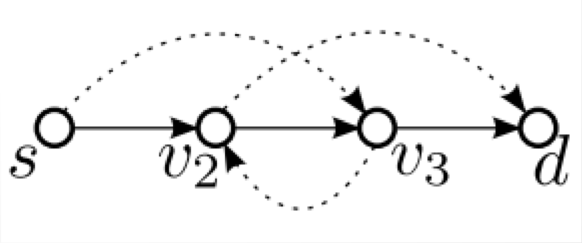
- Strong loop-freedom: Can we compute optimal schedules?
- Relaxed loop-freedom: Are O(1) rounds always enough?

No, but log(n) rounds are sufficient.

Remark on the Model

Easy to update new nodes which do not appear in old policy. And just keep nodes which are not on new path!

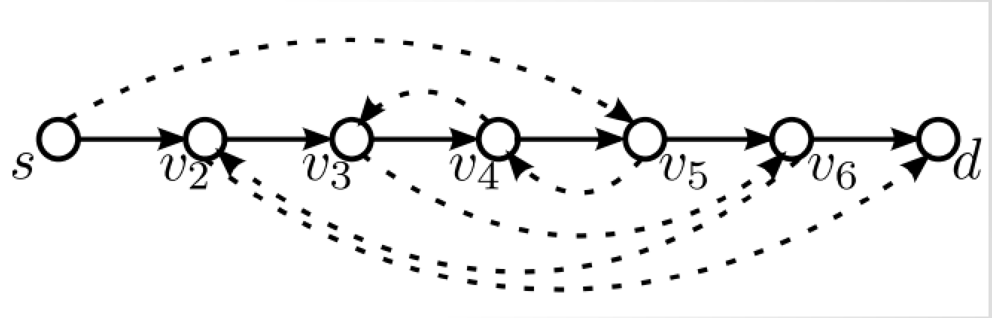




Good Algorithms to Schedule (Strong) LF Updates?

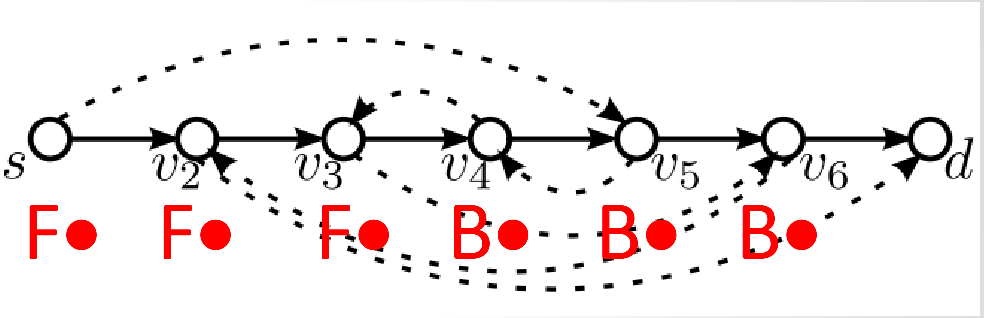
Optimal Algorithm for 2-Round Instances

- Classify nodes/edges with 2-letter code:
  - F•, B•: Does (dashed) new edge point forward or backward wrt (solid) old path?



Optimal Algorithm for 2-Round Instances

- Classify nodes/edges with 2-letter code:
  - F•, B•: Does (dashed) new edge point forward or backward wrt (solid) old path?

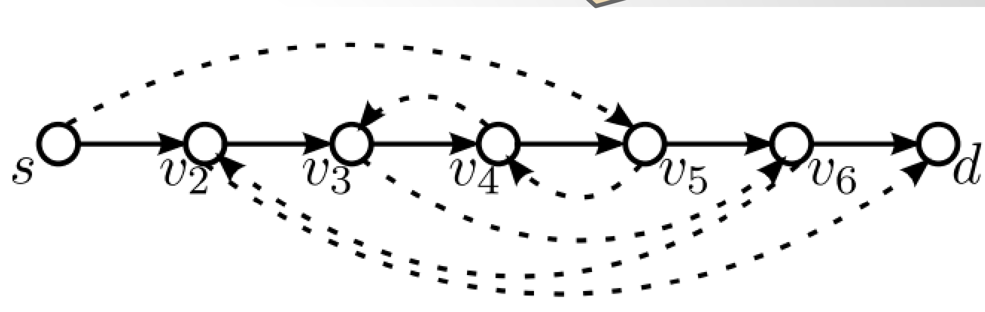


Optimal Algorithm for 2-Round Instances

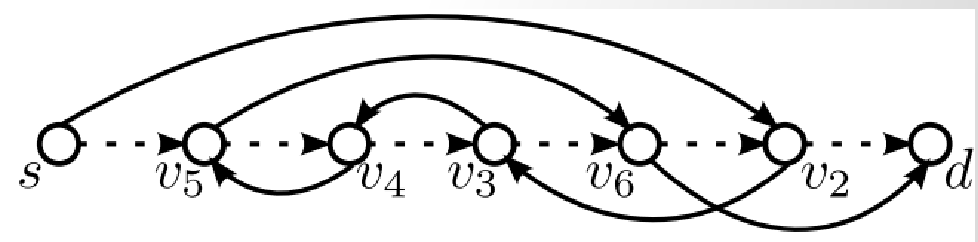
Old policy from left to right!

- Classify nodes/edg

  - F•, B•: Does (dashed) new edge point forward or backward wrt (solid) old path?



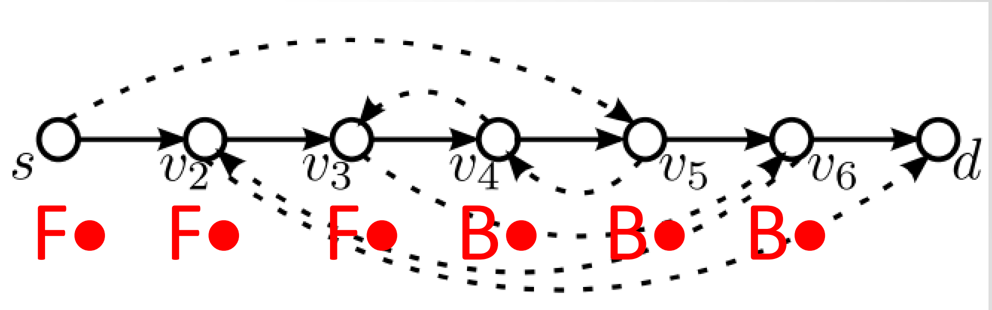
  - •F, •B: Does the (solid) old edge point forward or backwart wrt (dashed) new path?



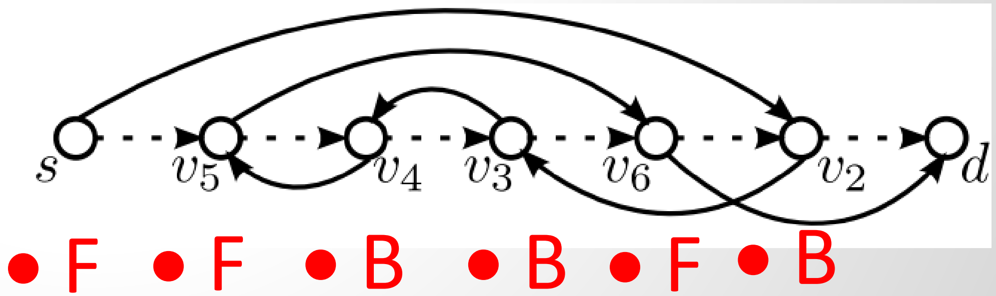
New policy from left to right!

Optimal Algorithm for 2-Round Instances

- Classify nodes/edges with 2-letter code:
  - F•, B•: Does (dashed) new edge point forward or backward wrt (solid) old path?



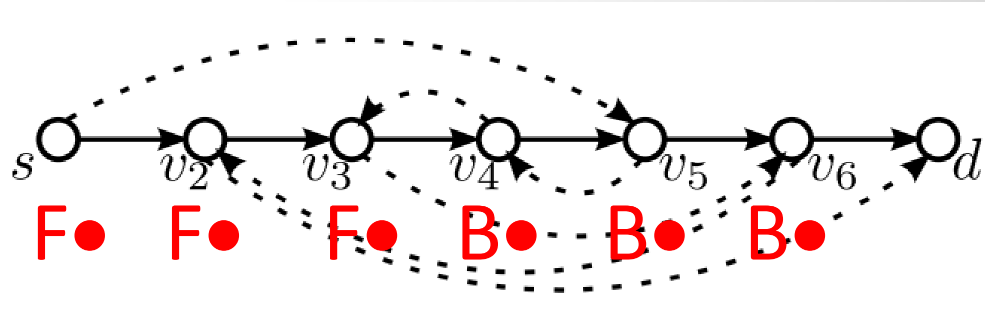
  - •F, •B: Does the (solid) old edge point forward or backwart wrt (dashed) new path?



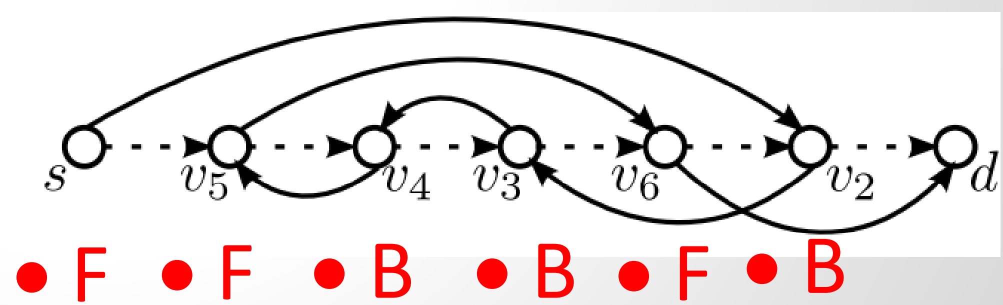
Insight 1: In the 1st round, I can safely update all forwarding (F•) edges! For sure loopfree.

…m for 2-Round Instances

- …es with 2-letter code:
  - F•, B•: Does (dashed) new edge point forward or backward wrt (solid) old path?



  - •F, •B: Does the (solid) old edge point forward or backwart wrt (dashed) new path?



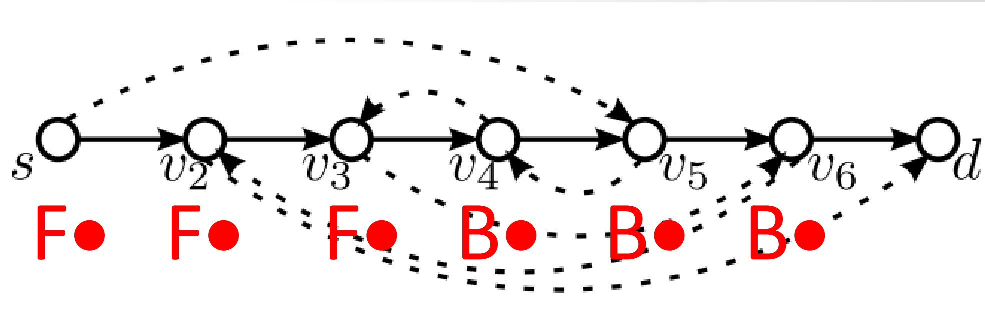
m for 2-Round Instances

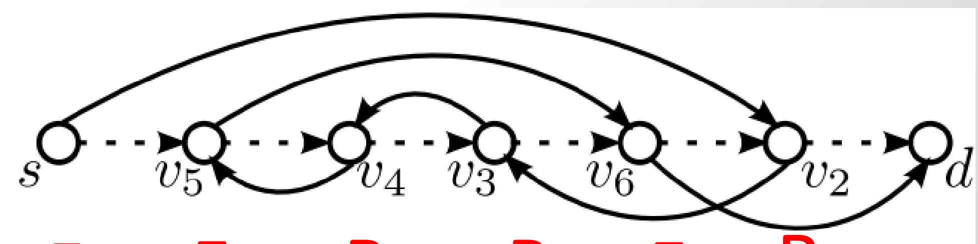
Insight 1: In the 1st round, I can safely update all forwarding (F•) edges! For sure loopfree.

es with 2-letter code:

Insight 2: Valid schedules are reversible! A valid schedule from old to new *read backward* is a valid schedule for new to old!

old edge point forward or backwart wrt (dashed) new path?





• F • F • B • B • F • B

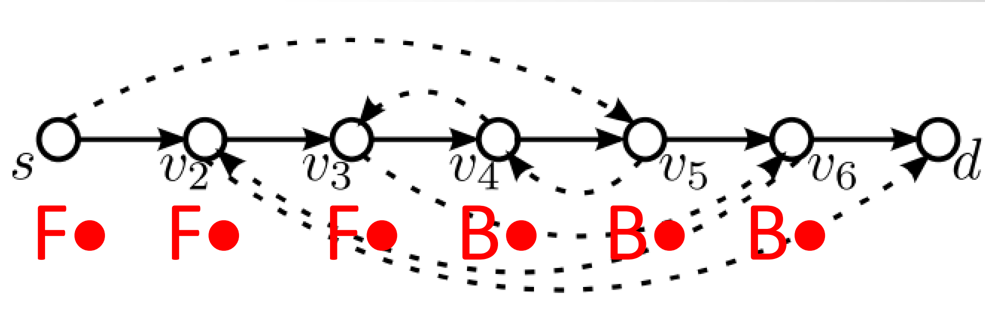
…m for 2-Round Instances

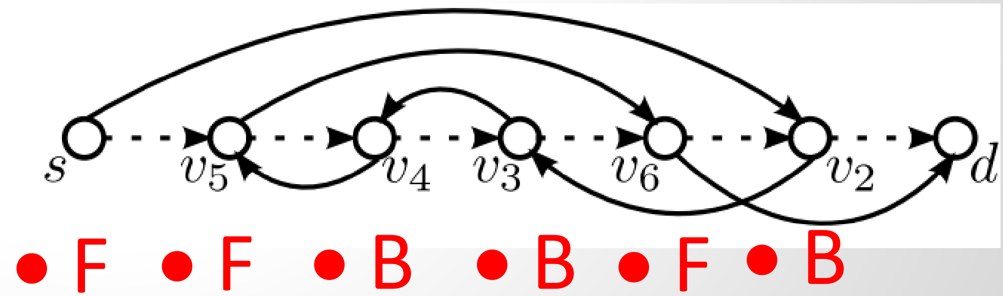
…s with 2-letter code:

Insight 1: In the 1st round, I can safely update all forwarding (F•) edges! For sure loopfree.

Insight 2: Valid schedules are reversible! A valid schedule from old to new *read backward* is a valid schedule for new to old!

Insight 3: Hence in the last round, I can safely update all forwarding (•F) edges! For sure loopfree.





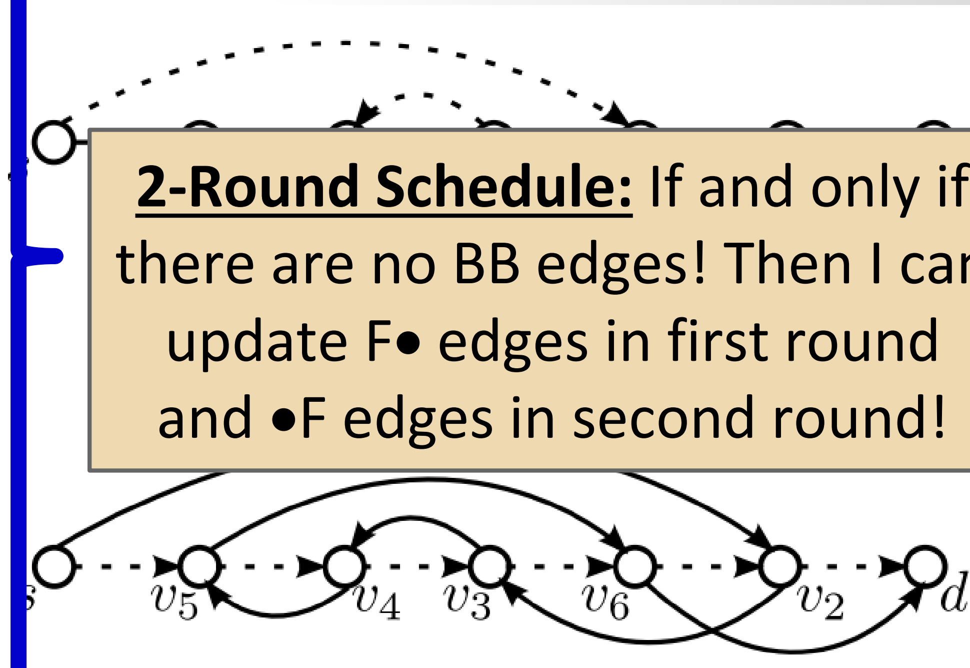
…m for 2-Round Instances

…es with 2-letter code:

<u>Insight 1:</u> In the 1st round, I can safely update all forwarding (F•) edges! For sure loopfree.

<u>Insight 2:</u> Valid schedules are reversible! A valid schedule from old to new *read backward* is a valid schedule for new to old!

<u>Insight 3:</u> Hence in the last round, I can safely update all forwarding (•F) edges! For sure loopfree.



<u>2-Round Schedule:</u> If and only if there are no BB edges! Then I can update F• edges in first round and •F edges in second round!

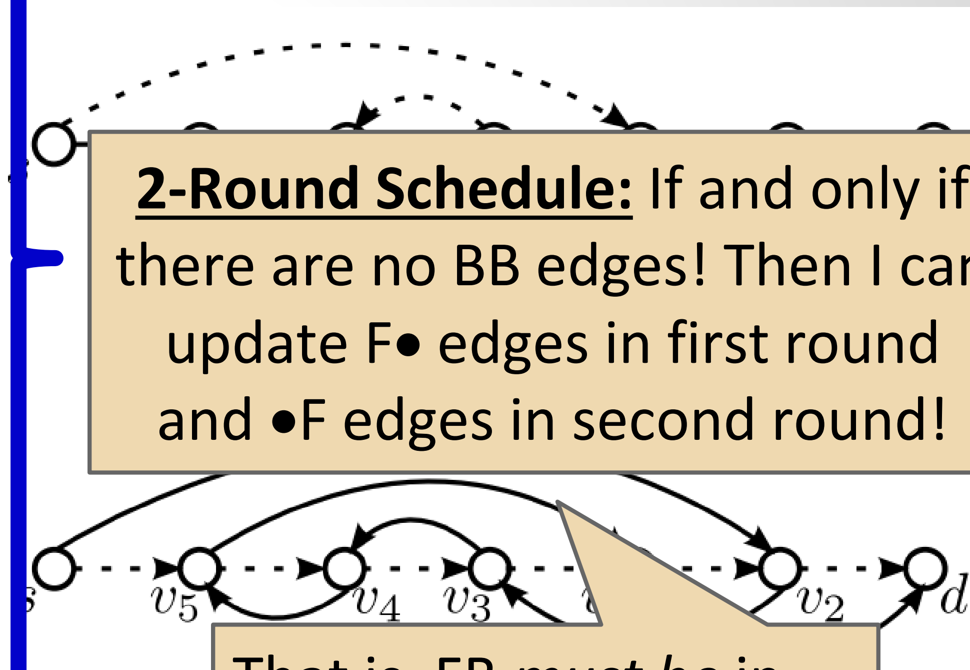
m for 2-Round Instances

es with 2-letter code:

Insight 1: In the 1st round, I can safely update all forwarding (F•) edges! For sure loopfree.

Insight 2: Valid schedules are reversible! A valid schedule from old to new *read backward* is a valid schedule for new to old!

Insight 3: Hence in the last round, I can safely update all forwarding (•F) edges! For sure loopfree.



2-Round Schedule: If and only if there are no BB edges! Then I can update F• edges in first round and •F edges in second round!

That is, FB *must be* in first round, BF *must be* in second round, and FF are *flexible*!

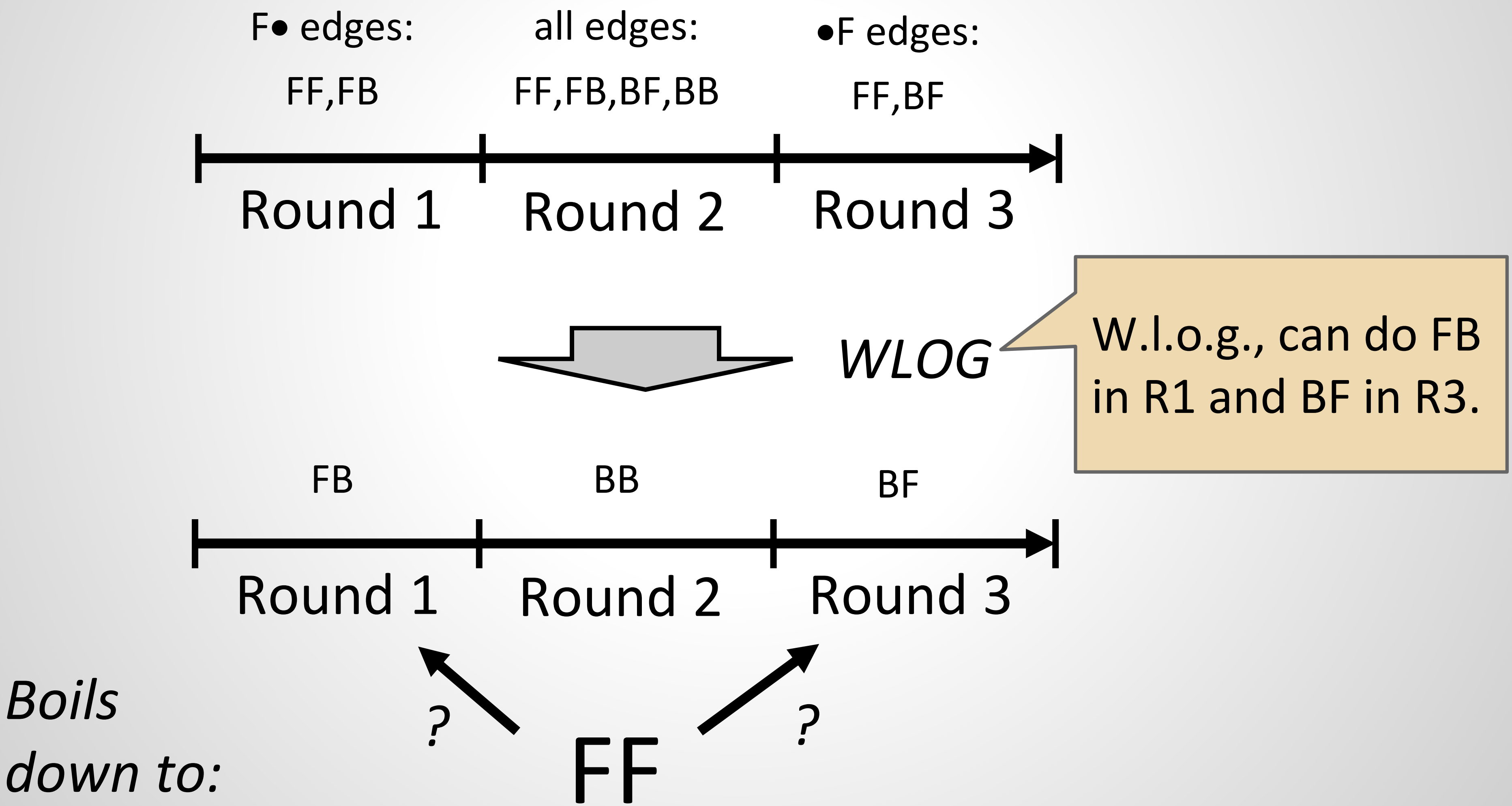
What about 3 rounds?

What about 3 rounds?

- Structure of a 3-round schedule:

What about 3 rounds?

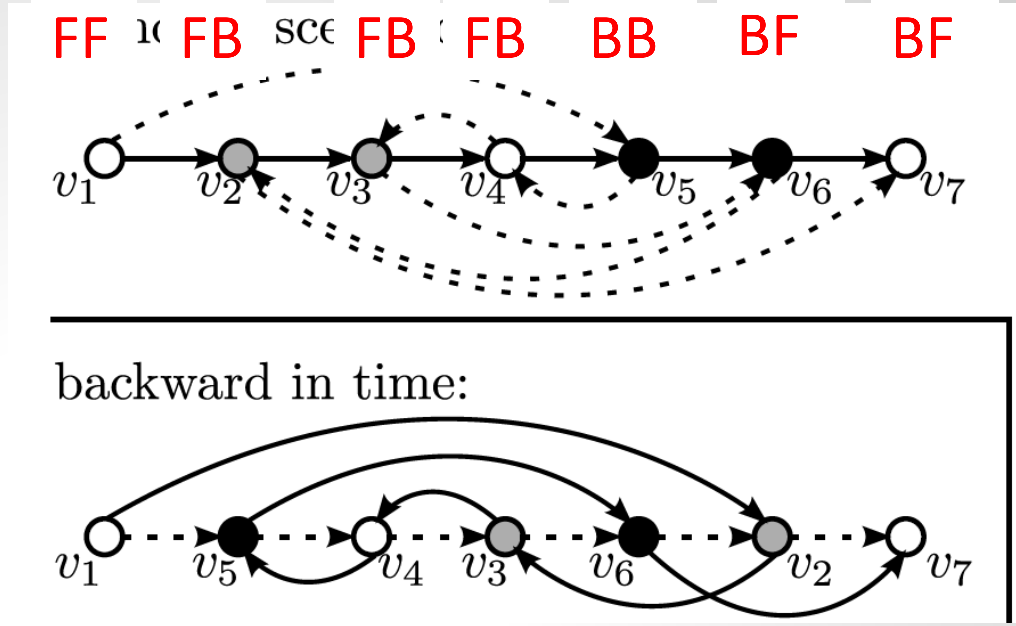
- Structure of a 3-round schedule:



Proof

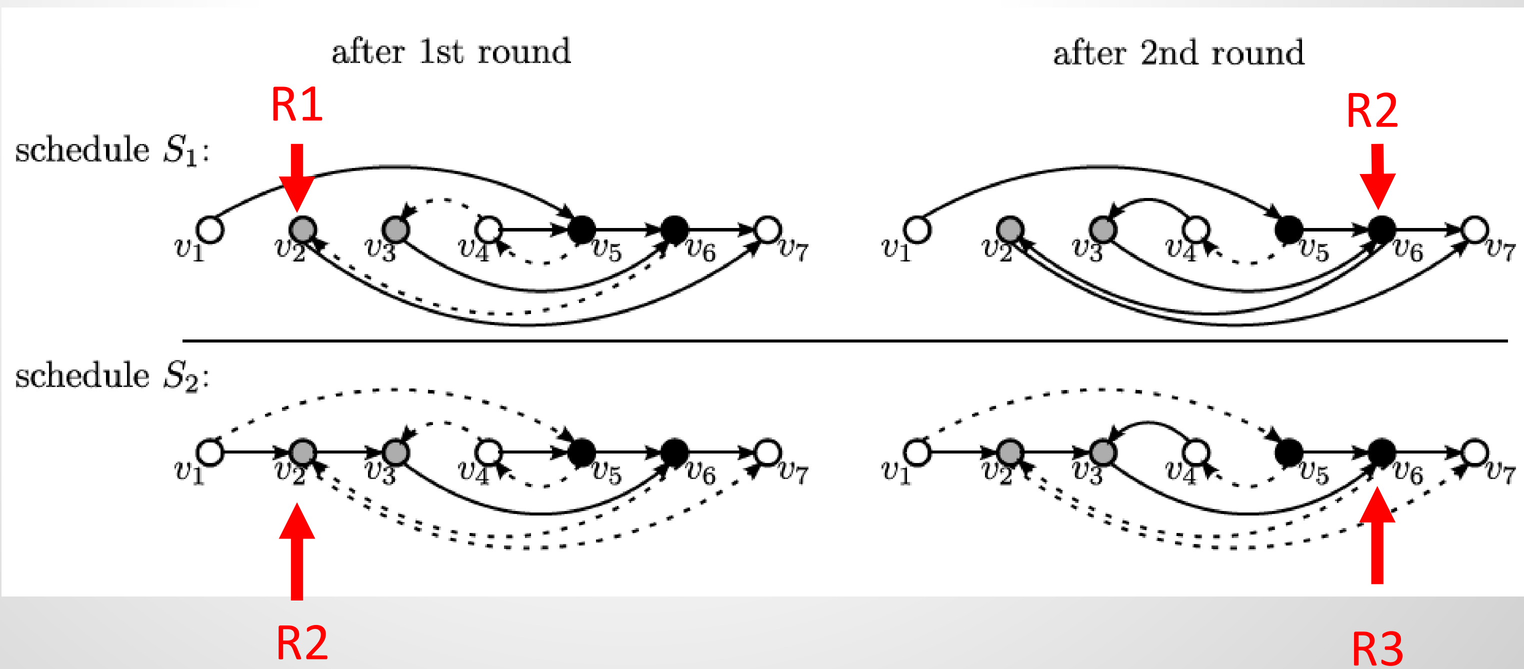
Claim: If there exists 3-round schedule, then also one where FB are only updated in Round 1.

Reason: Can move FB to first round!



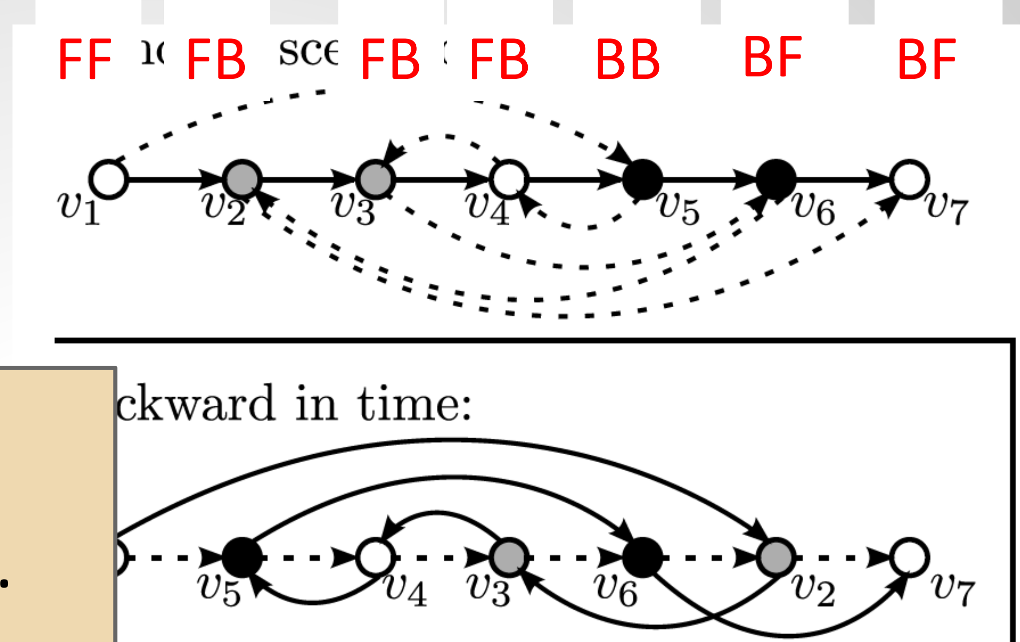
S1: as early as possible

S2: as late as possible



Proof

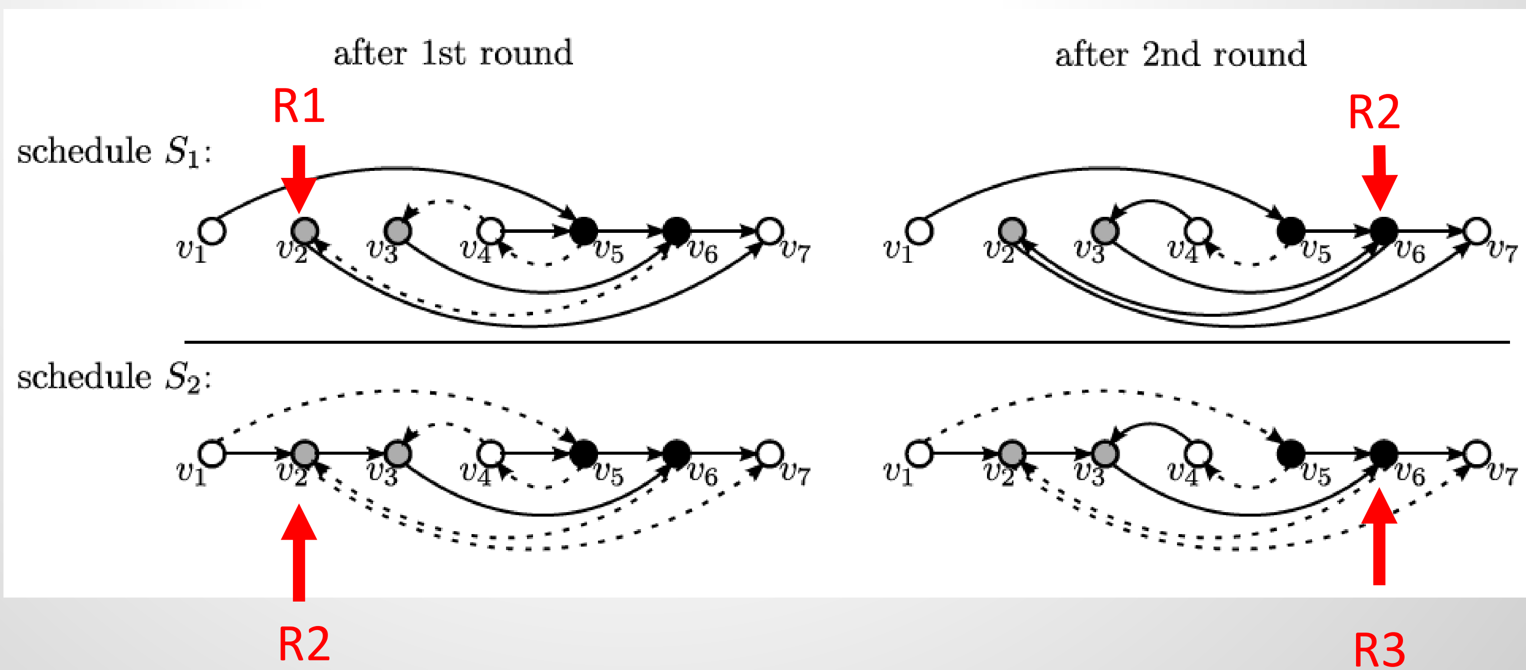
Claim: If there exists 3-round schedule, then also



Fowarding edges do not introduce loops in G(t=1).

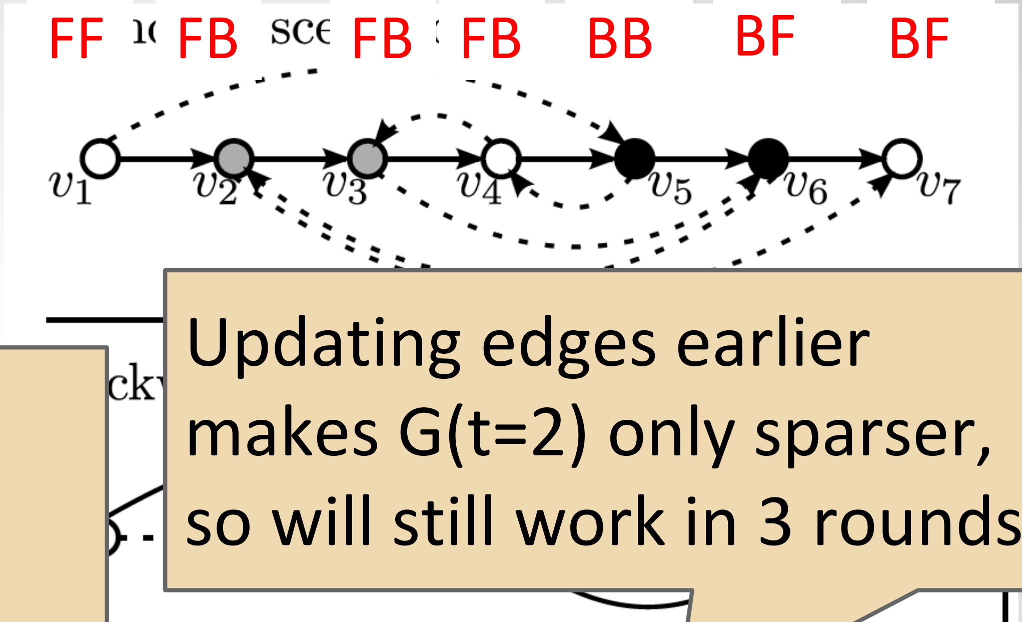
S1: as early as possible

S2: as late as possible



Proof

Claim: If there exists 3-round schedule, then also

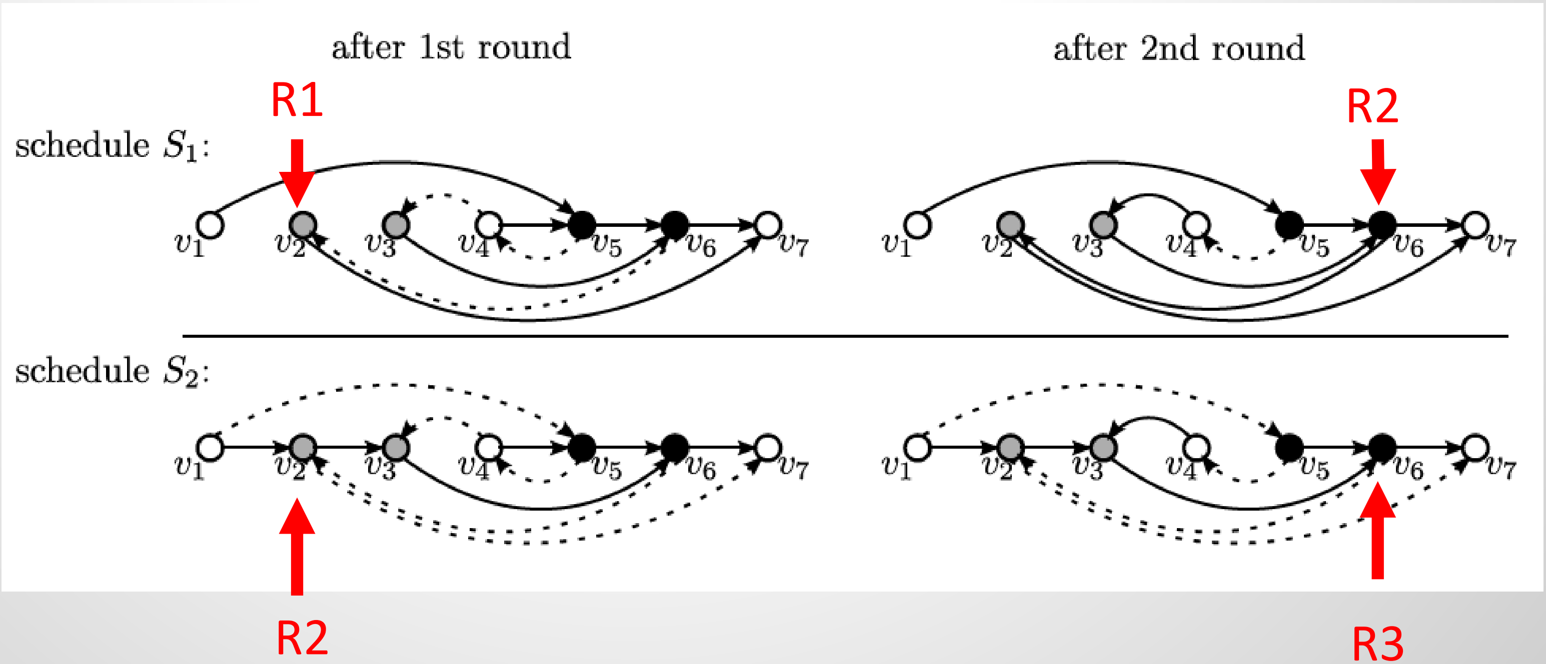


Updating edges earlier makes G(t=2) only sparser, so will still work in 3 rounds.

Fowarding edges do not introduce loops in G(t=1).

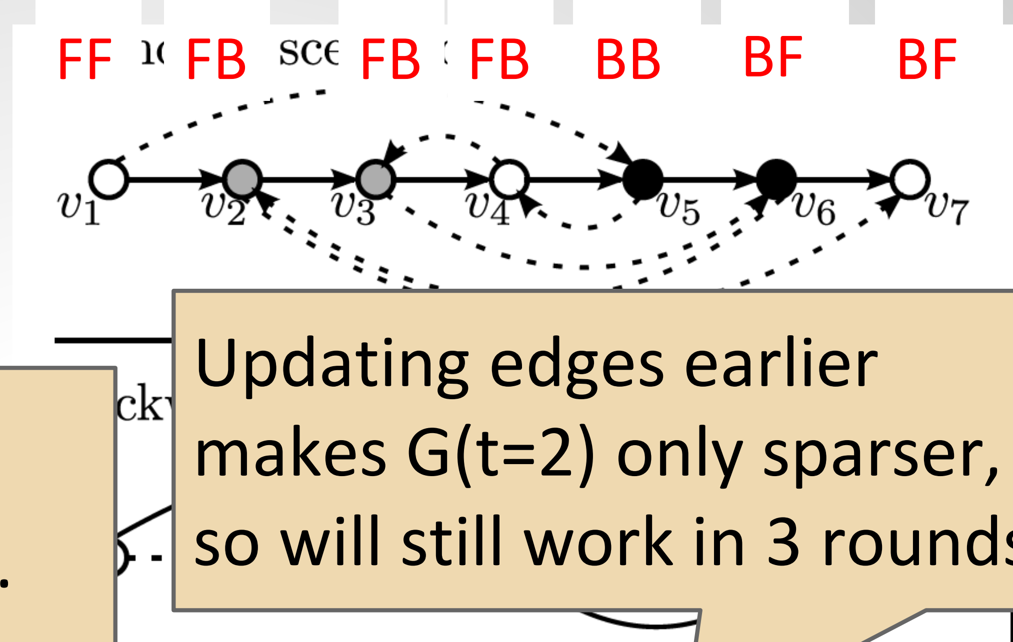
S1: as early as possible

S2: as late as possible



Proof

Claim: If there exists 3-round schedule, then also



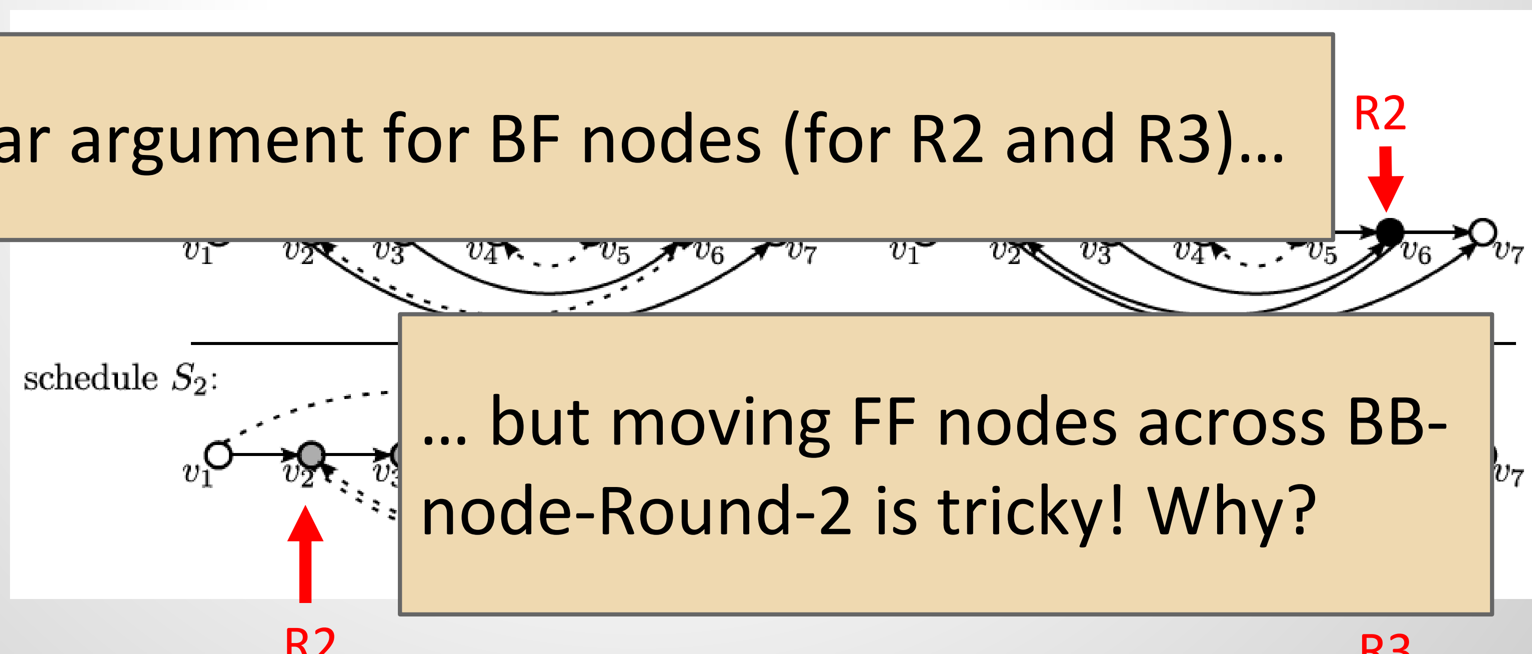
Fowarding edges do not introduce loops in G(t=1).

Updating edges earlier makes G(t=2) only sparser, so will still work in 3 rounds.

S1: a

as po

S2: as late as possible

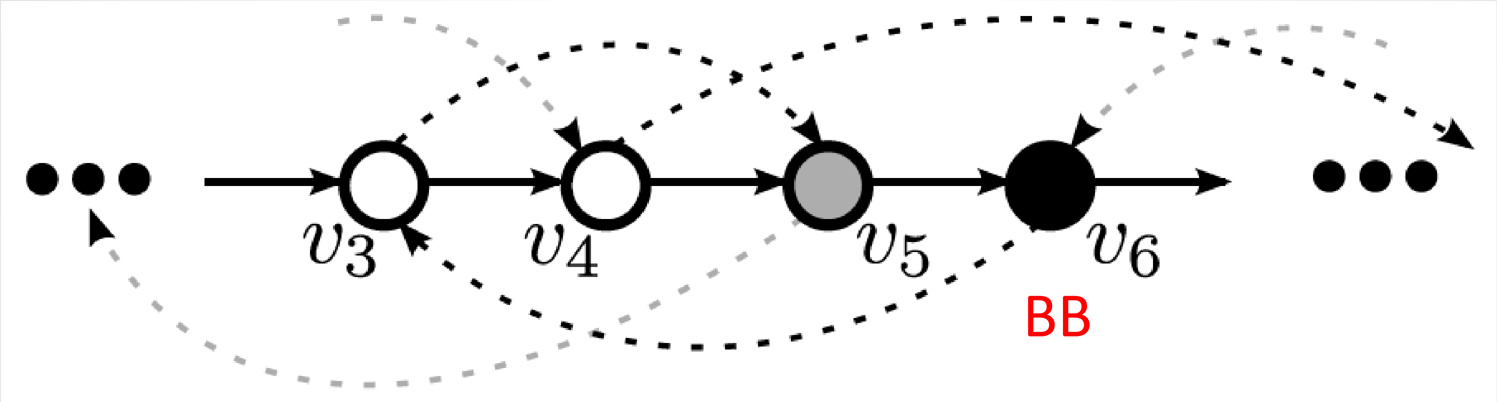


Similar argument for BF nodes (for R2 and R3)…

… but moving FF nodes across BB-node-Round-2 is tricky! Why?

NP-hardness

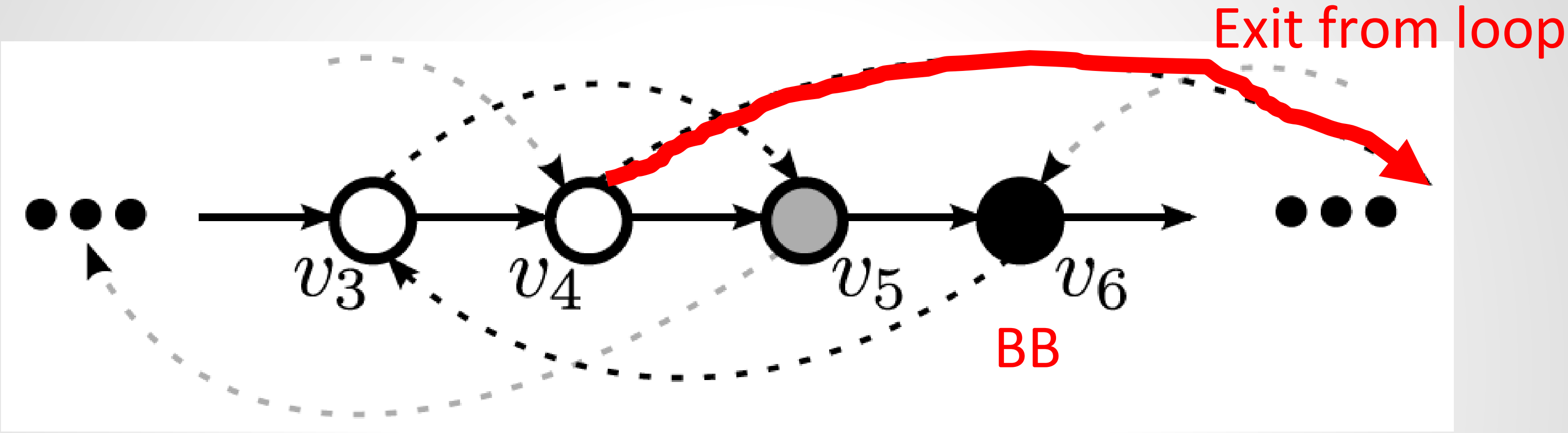
A hard decision problem: when to update FF?



- We know: BB node v_6 can only be updated in R2

NP-hardness

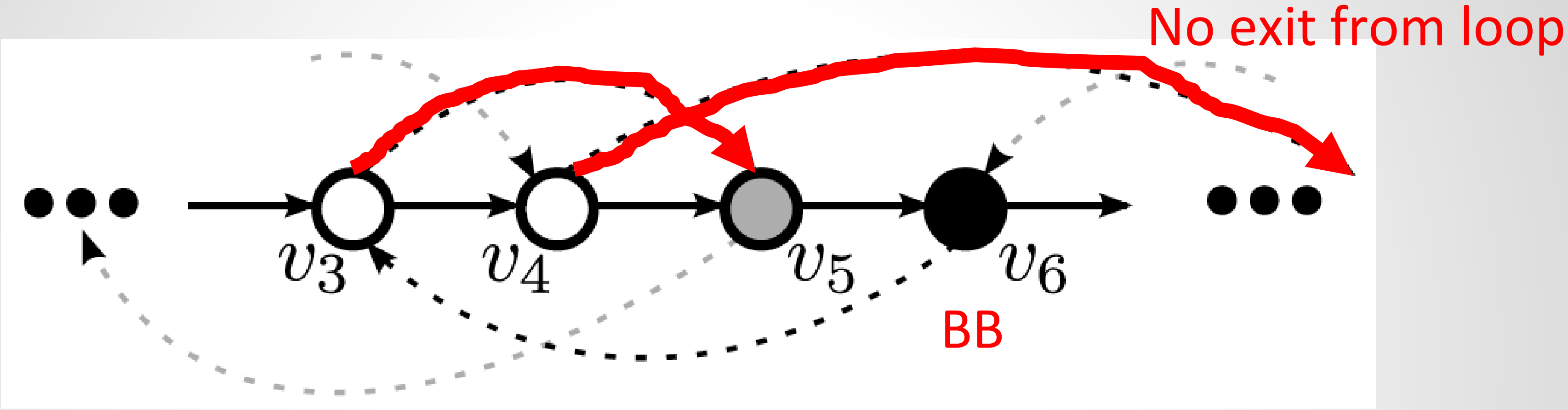
A hard decision problem: when to update FF?



- We know: BB node v_6 can only be updated in R2
- Updating FF-node v_4 in R1 allows to update BB node v_6 in R2

NP-hardness

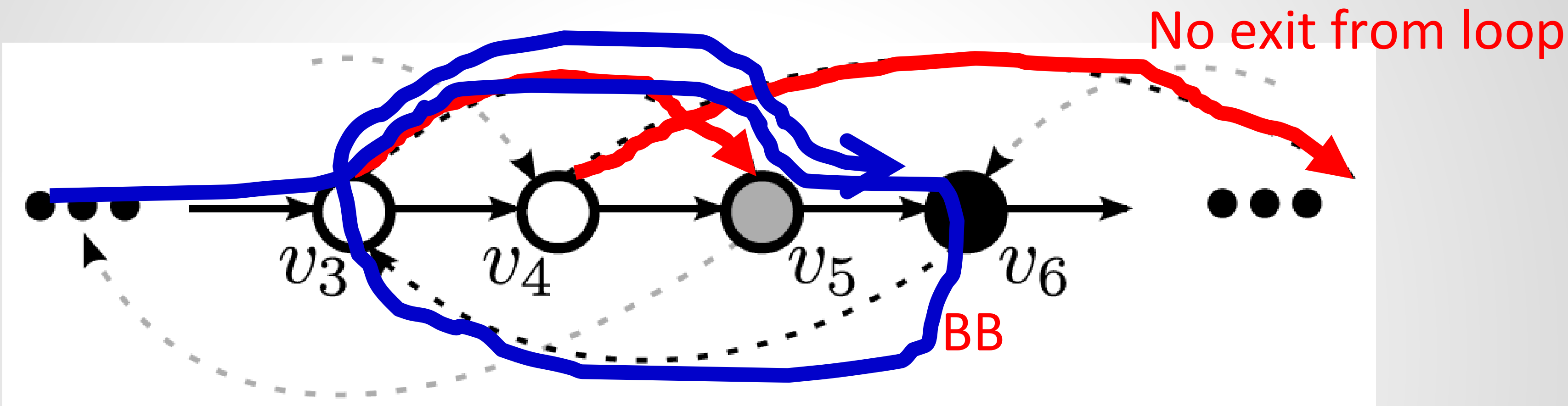
A hard decision problem: when to update FF?



- We know: BB node v_6 can only be updated in R2
- Updating FF-node v_4 in R1 allows to update BB node v_6 in R2
- Updating FF-node v_3 as well in R1 would be bad: cannot update v_6 in next round: potential loop

NP-hardness

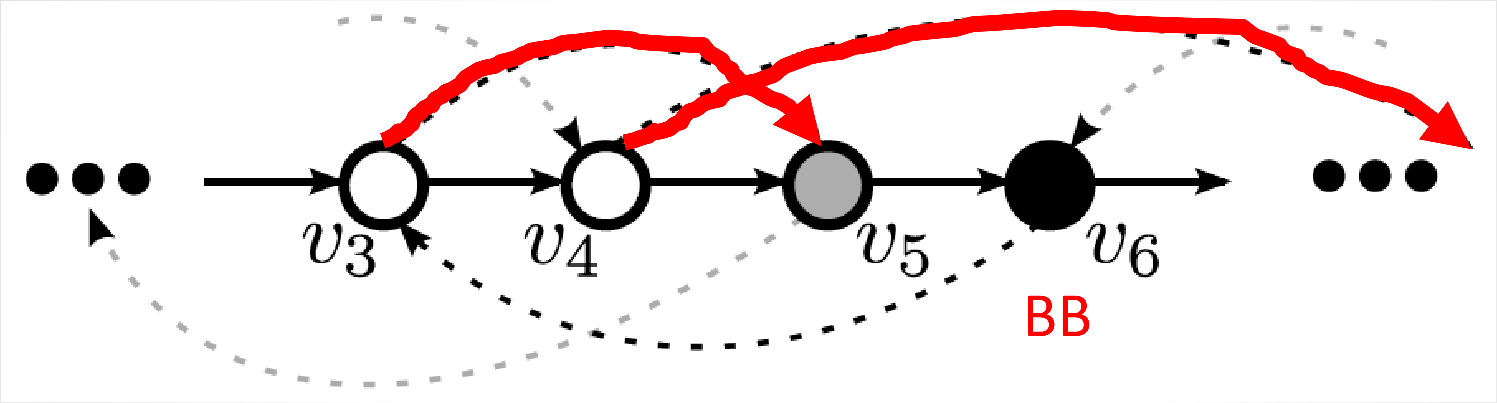
A hard decision problem: when to update FF?



- We know: BB node v_6 can only be updated in R2
- Updating FF-node v_4 in R1 allows to update BB node v_6 in R2
- Updating FF-node v_3 as well in R1 would be bad: cannot update v_6 in next round: potential loop

NP-hardness

A hard decision problem: when to update FF?



- We know: BB node v_6 can only be updated in R2
- Updating FF-node v_4 in R1 allows to update BB node v6 in R2
- Updating FF-node v_3 as well in R1 would be bad: cannot update v_6 in next round: potential loop
- Node v_5 is B• and cannot be updated in R1

NP-hardness

- Reduction from a 3-SAT version where variables appear only a small number of times
  - Variable x appearing p_x times positively and n_x times negatively is replaced by:

    $$x_0, x_1, \ldots, x_{p_x}, x_l, \overline{x}_0, \overline{x}_1, \ldots, \overline{x}_{n_x}$$
  - Gives low-degree requirements!

- Types of clauses
  - Assignment clause: $(x_0 \vee \overline{x}_0)$
  - Implication clause: $(x_i \rightarrow x_{i+1})$
  - Exclusive Clause: $(\neg x_l \vee \neg \overline{x}_l)$

NP-hardness

We need a low degree…

- Reduction from [obscured] where variables appear only a small number of times
  - Variable x appearing p_x times positively and n_x times negatively is replaced by:

    $$x_0, x_1, \ldots, x_{p_x}, x_l, \overline{x}_0, \overline{x}_1, \ldots, \overline{x}_{n_x}$$

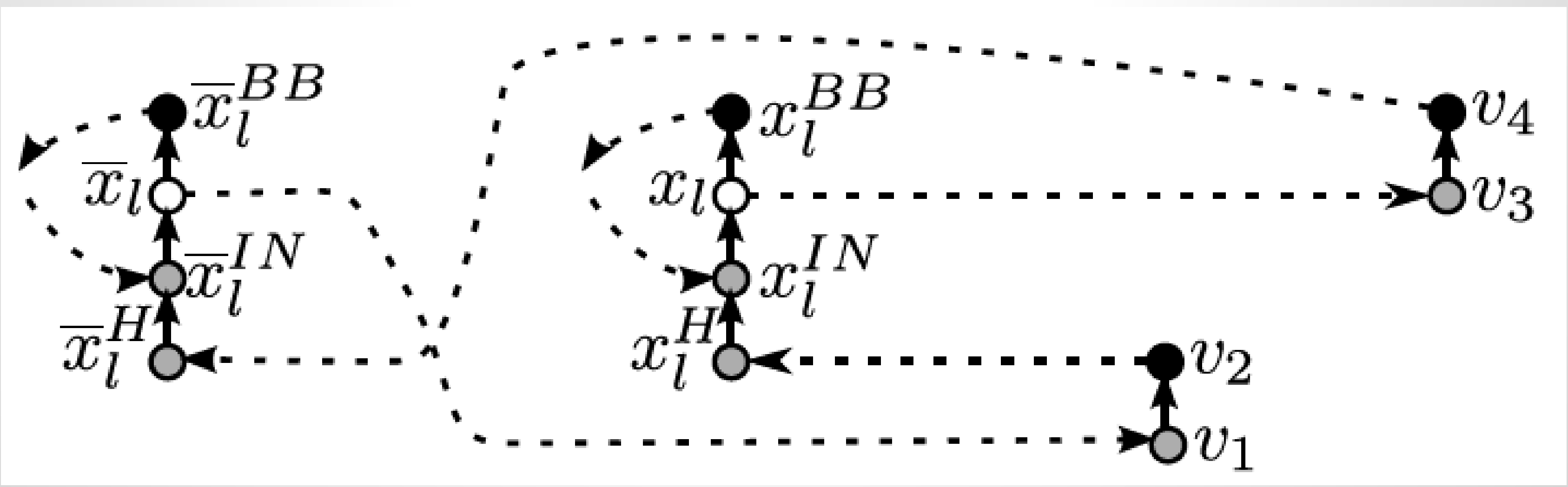
  - Gives low-degree requirements!

- Types of clauses
  - Assignment clause: $(x_0 \vee \overline{x}_0)$
  - Implication clause: $(x_i \rightarrow x_{i+1})$
  - Exclusive Clause: $(\neg x_l \vee \neg \overline{x}_l)$

Connecting clones: consistent value for original variable.

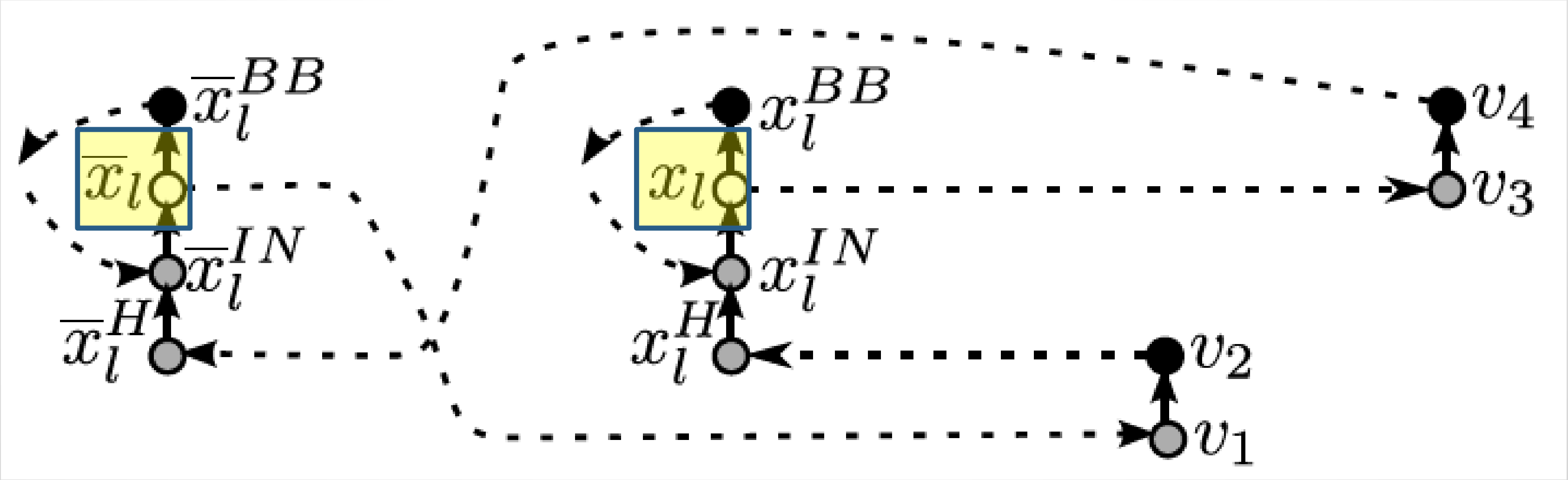
Example: Gadget for Exclusive Clause $(\neg x_l \vee \neg \overline{x}_l)$

- Updating x_l prevents $\overline{x}_l$ update and vice versa
- BB nodes v_2 and v_4 need to be updated in R2 and will introduce a cycle otherwise
- So only one of the two can be updated in R1



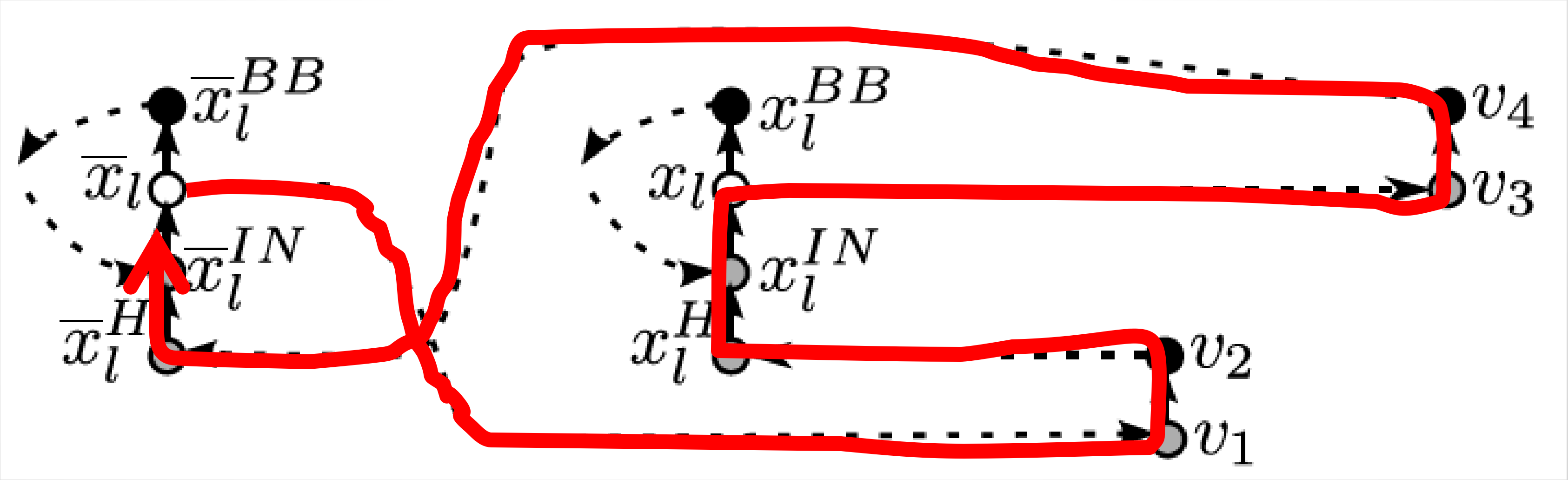
Example: Gadget for Exclusive Clause $(\neg x_l \vee \neg \overline{x}_l)$

- Updating x_l prevents $\overline{x}_l$ update and vice versa
- BB nodes v_2 and v_4 need to be updated in R2 and will introduce a cycle otherwise
- So only one of the two can be updated in R1

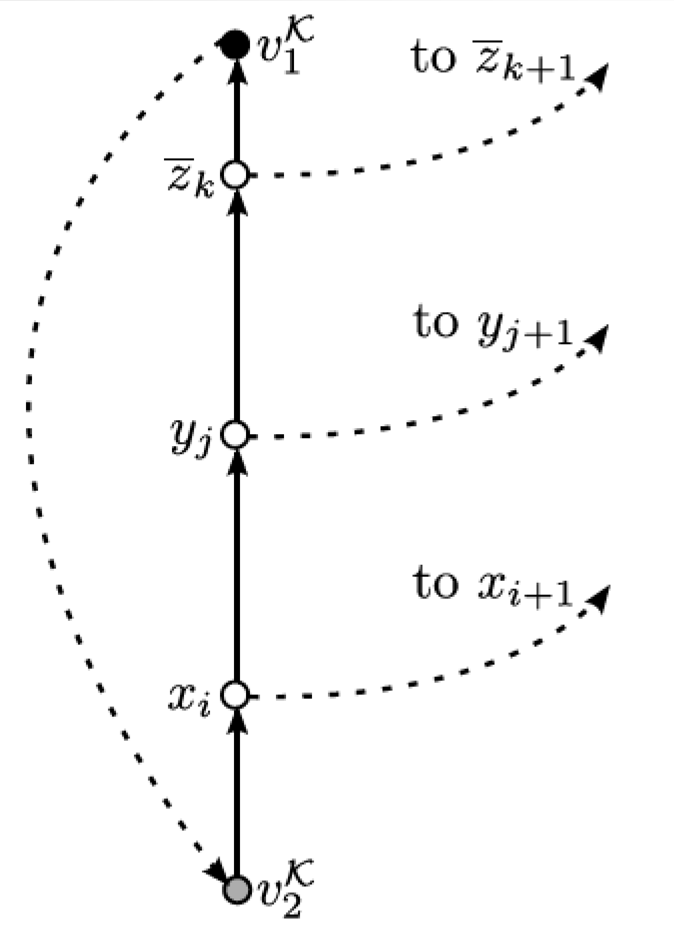


Example: Gadget for Exclusive Clause $(\neg x_l \vee \neg \overline{x}_l)$

- Updating x_l prevents $\overline{x}_l$ update and vice versa
- BB nodes v_2 and v_4 need to be updated in R2 and will introduce a cycle otherwise
- So only one of the two can be updated in R1

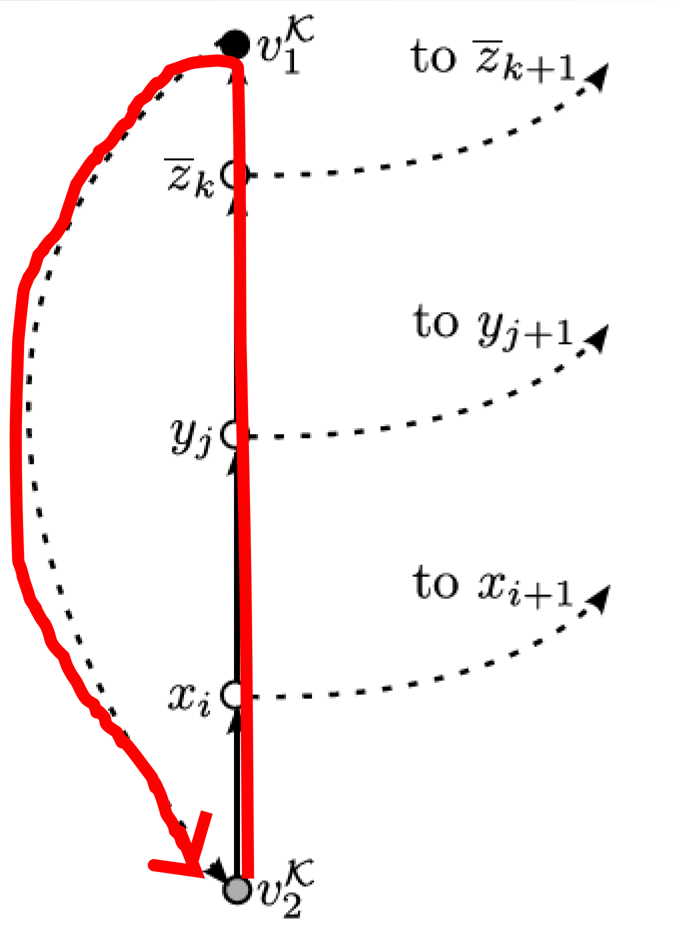


Example: Gadget for Clause $x_i \vee y_j \vee \overline{z}_k$



- Need to update (satisfy) at least one of the literals in the clause…
- … so to escape the potential loop

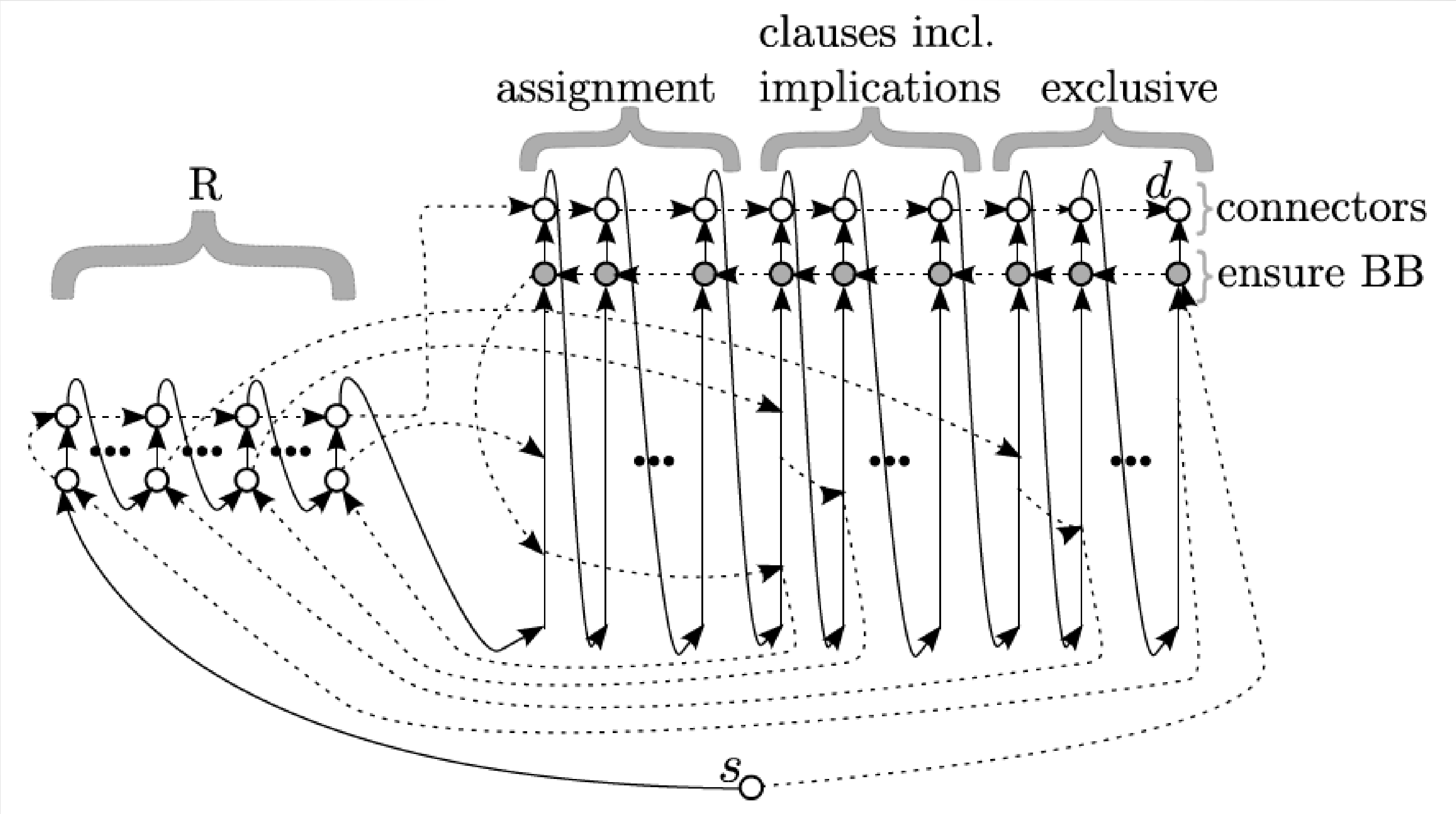
Example: Gadget for Clause $x_i \vee y_j \vee \overline{z}_k$



- ❑ Need to update (satisfy) at least one of the literals in the clause…
- ❑ … so to escape the potential loop

NP-hardness

- Eventually everything has to be connected…
- … to form a valid path



Relaxed Loopfreedom

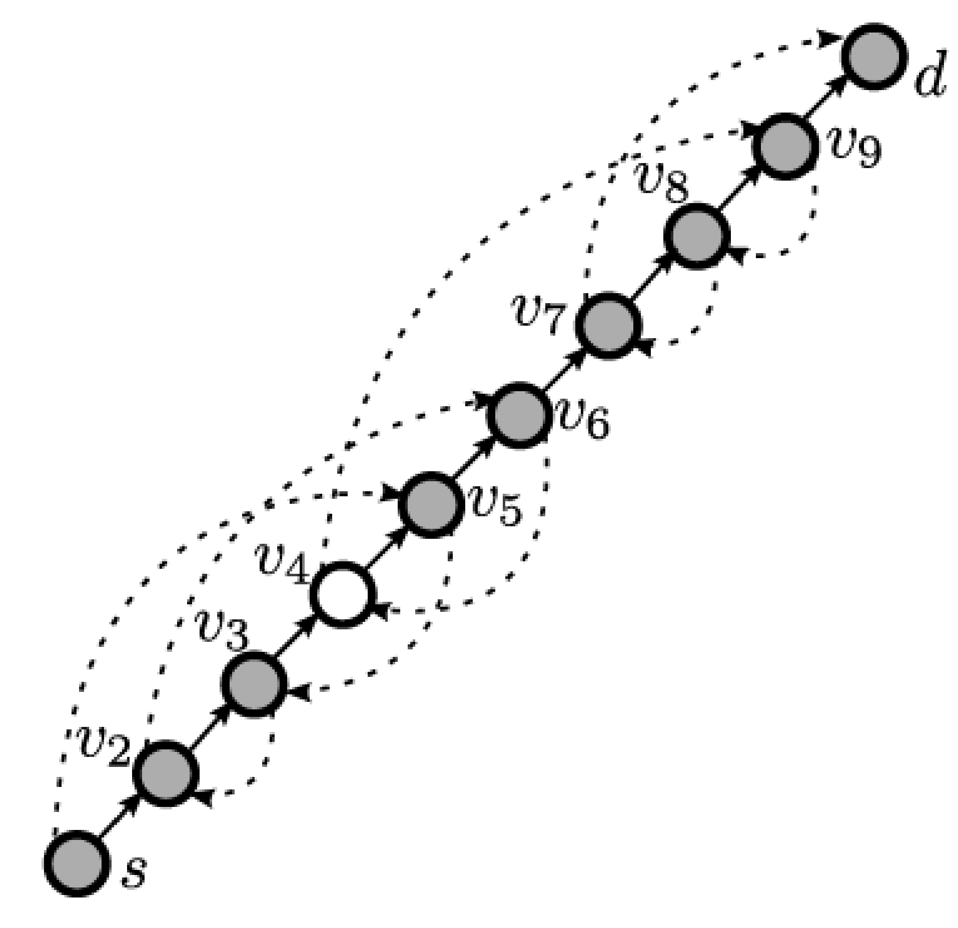
- Recall: relaxed loop-freedom can reduce number of rounds by a factor O(n)
- But how many rounds are needed for relaxed loop-free update in the worst case?
- We don't know…
- … what we do know: next slide ☺

Peacock: Relaxed Updates in O(log n) Rounds

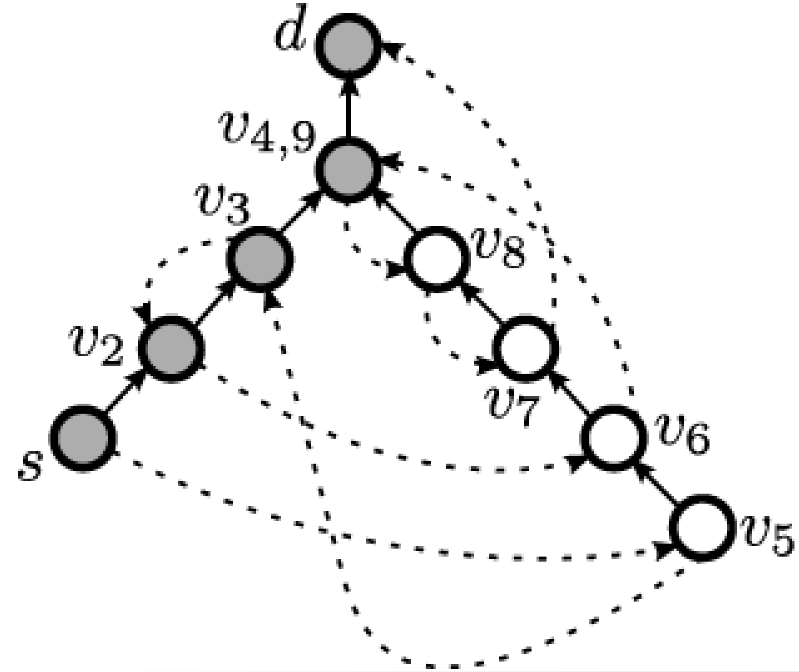
Two observations / principles:

- **Node merging:** a node which is updated is *irrelevant for the future*, so merge it with subsequent one
- **Directed tree:** while initial network consists of two directed paths (in-degree=out-degree=2), during update rounds, situation can become a directed tree
  - in-degree can *increase* due to merging
  - dashed in- and out-degree however stays one

Example

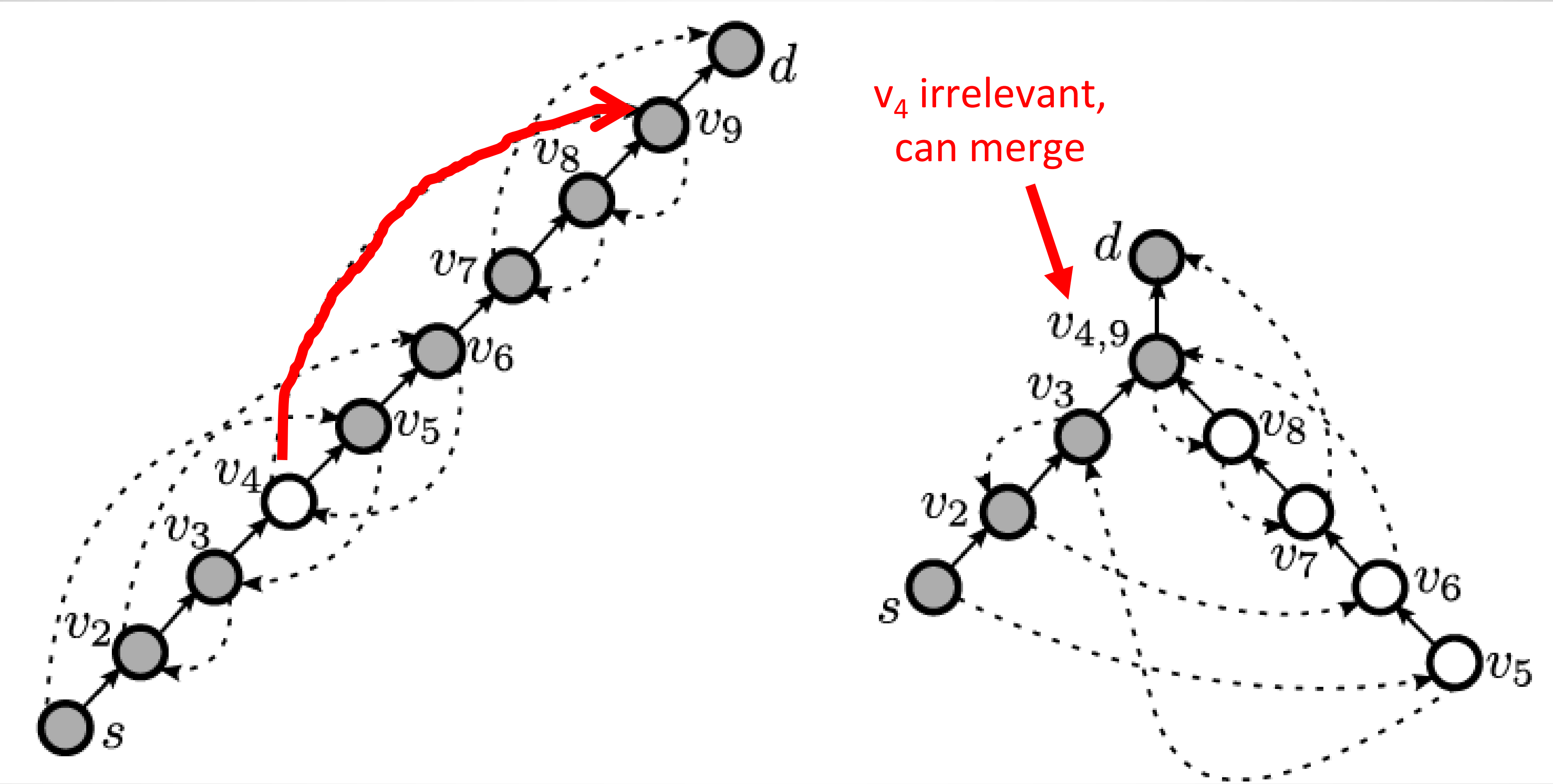


Initially: Two valid paths!



After updating v_4.

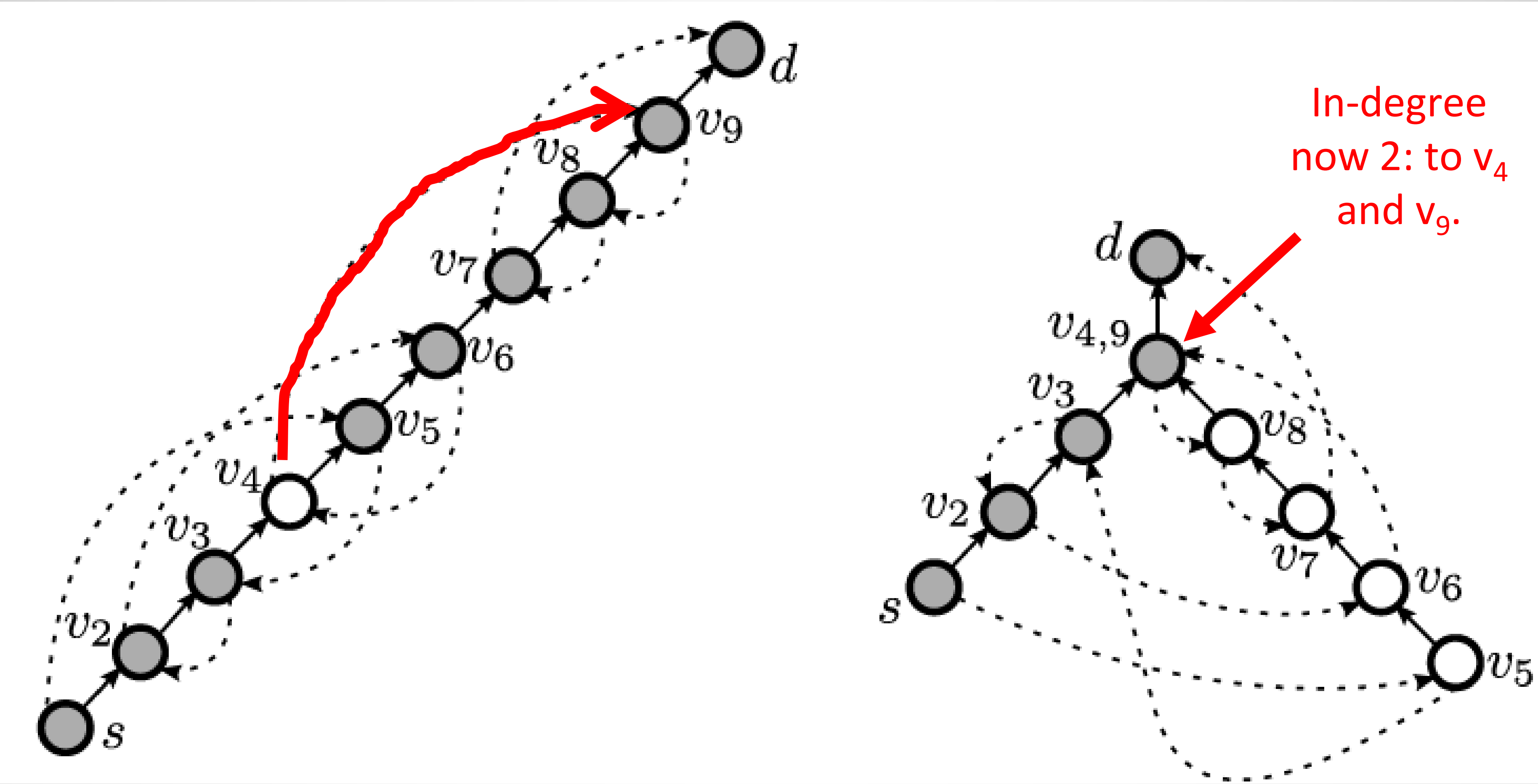
Example



Initially: Two valid paths!

After updating v_4.

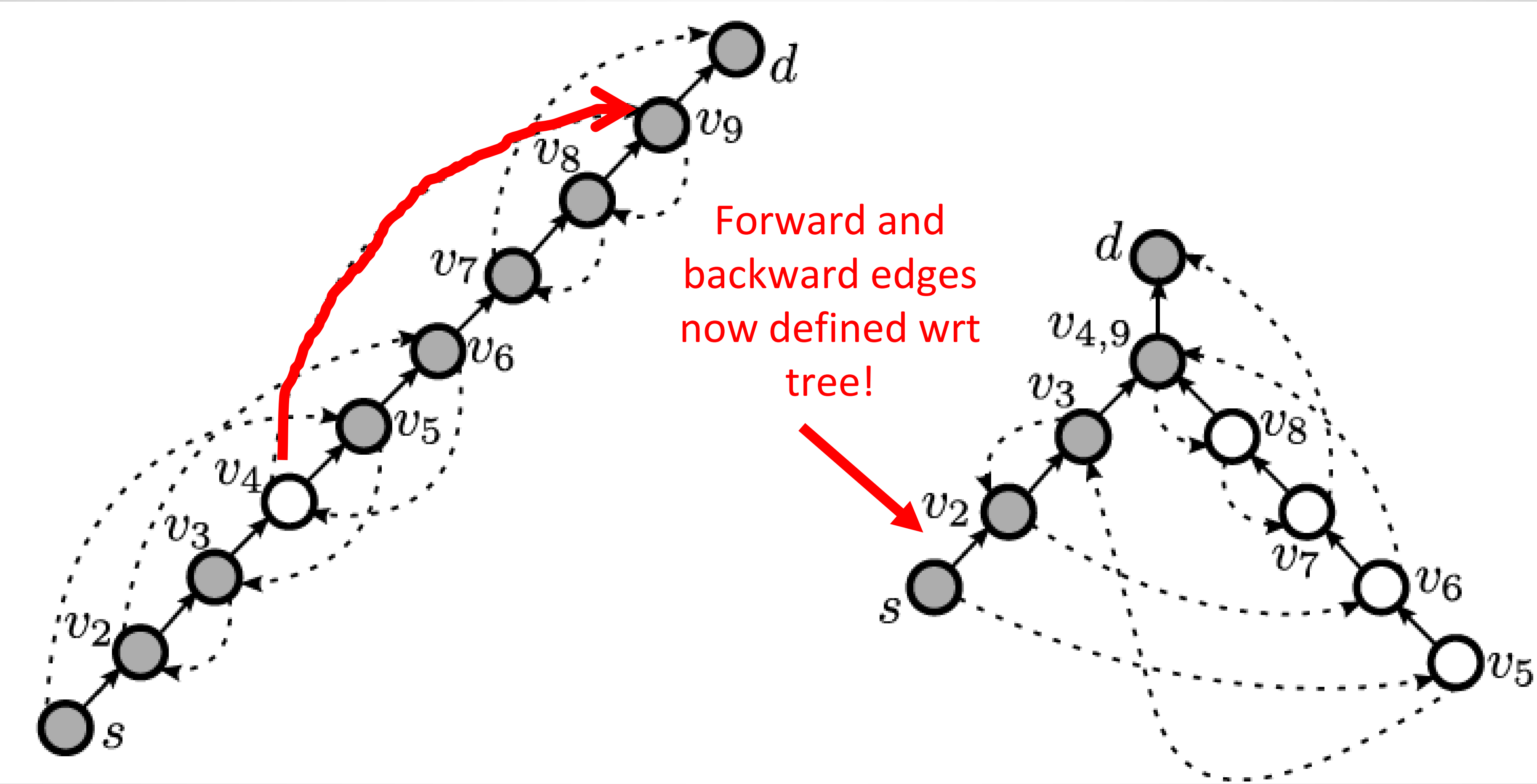
Example



Initially: Two valid paths!

After updating v_4.

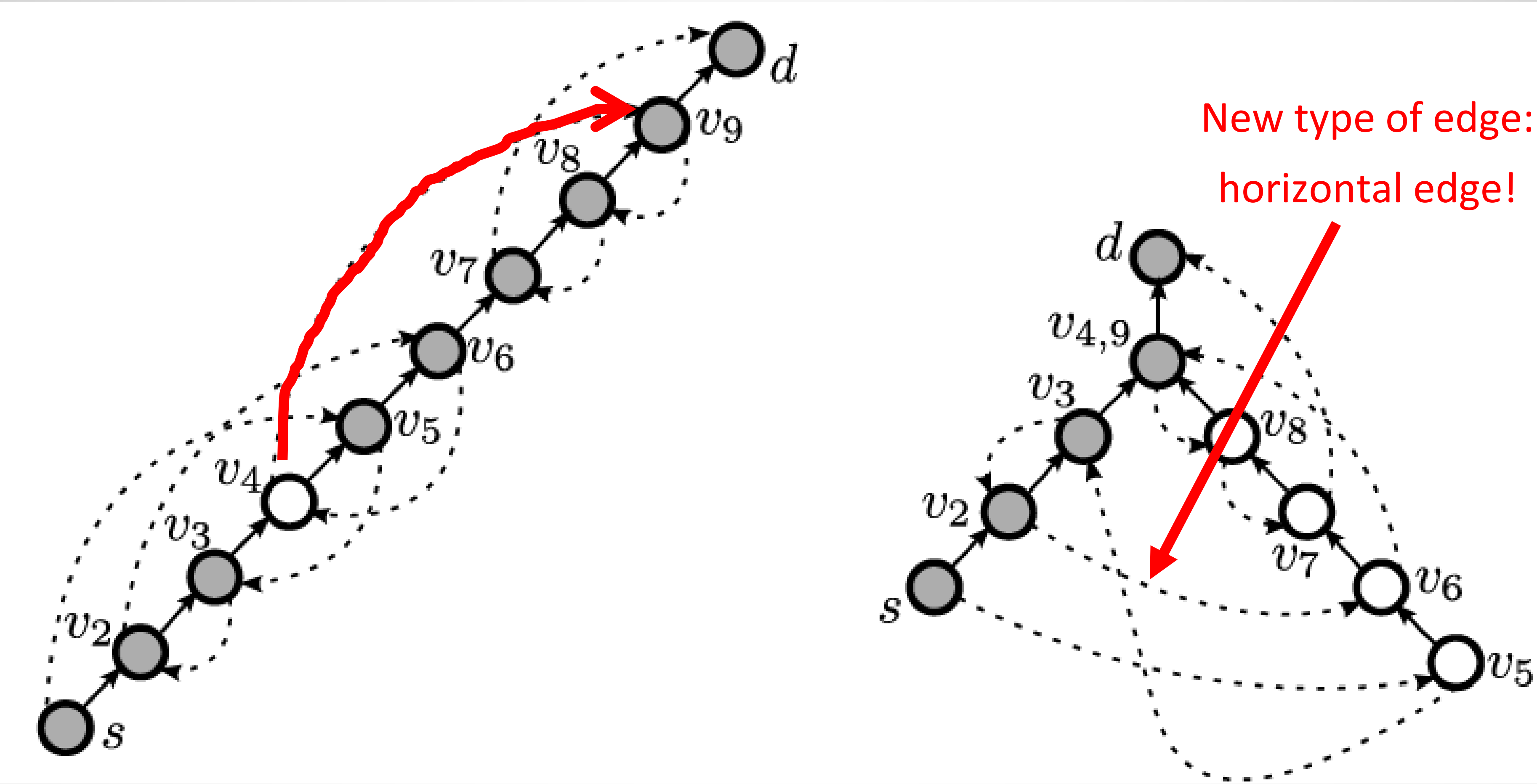
Example



Initially: Two valid paths!

After updating v_4.

Example



Initially: Two valid paths!

After updating v_4.

Ideas of Peacock Algorithm

- **Rounds come in pairs:** Try to update (and hence merge) as much as possible in every other round

- **Round 1 (odd rounds): Shortcut**
  - Move source close to destination
  - Generate many «independent subtrees» which are easy to update!

- **Round 2 (even rounds): Prune**
  - Update independent subtrees
  - Brings us back to a chain!

Ideas of Peacock Algorithm

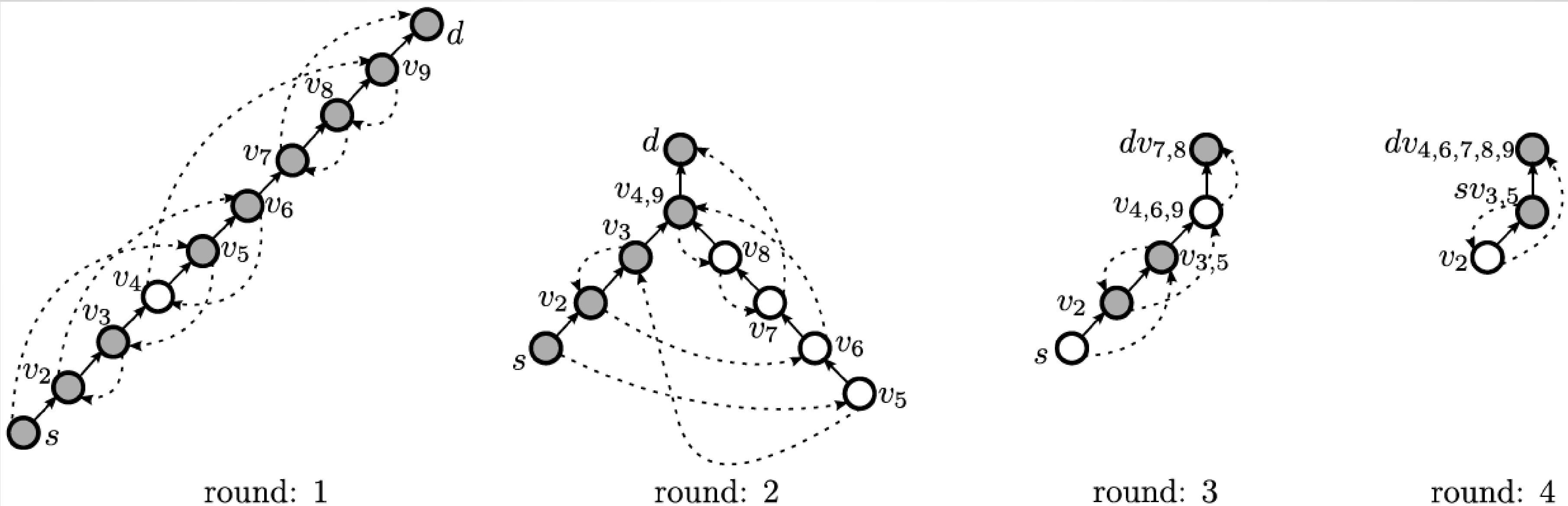
- **Rounds come in pairs:** Try to update (and hence merge) as much as possible in every other round

- **Round 1 (odd rounds): Shortcut**
  - Move source close to destination
  - Generate many «independent subtrees» which are easy to update!

Don‘t be greedy!
Don‘t update all FF edges!

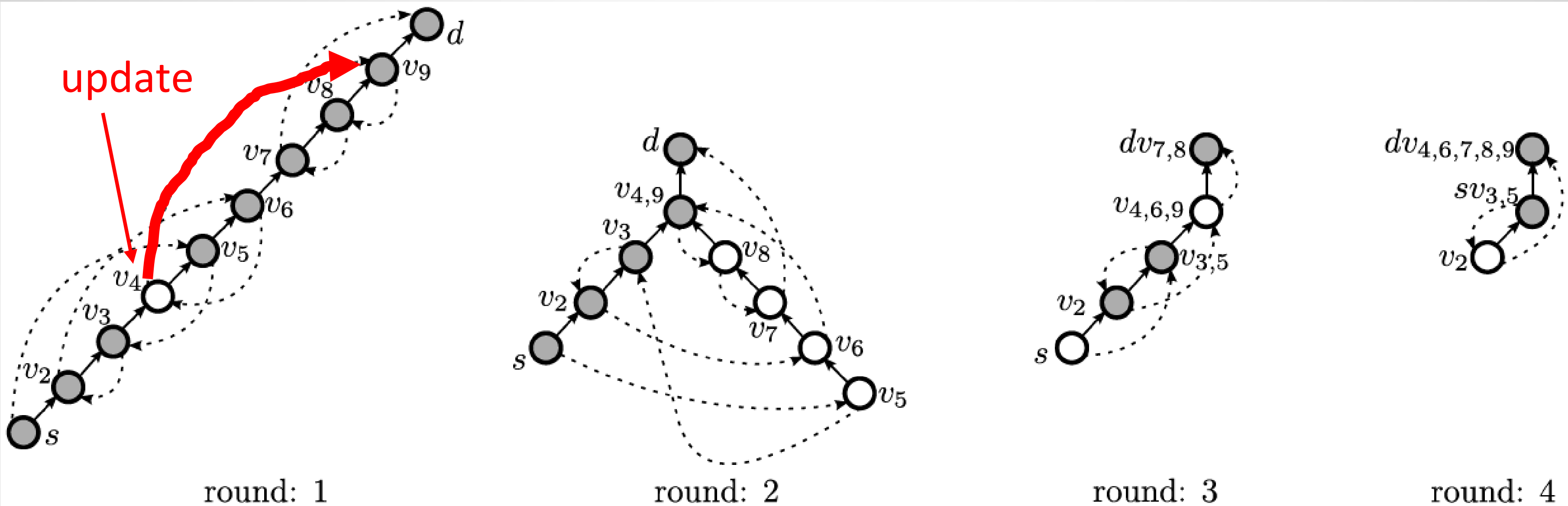
- **Round 2 (even rounds): Prune**
  - Update independent subtrees
  - Brings us back to a chain!

Peacock in Action



Shortcut Prune Shortcut Prune

Peacock in Action



Shortcut

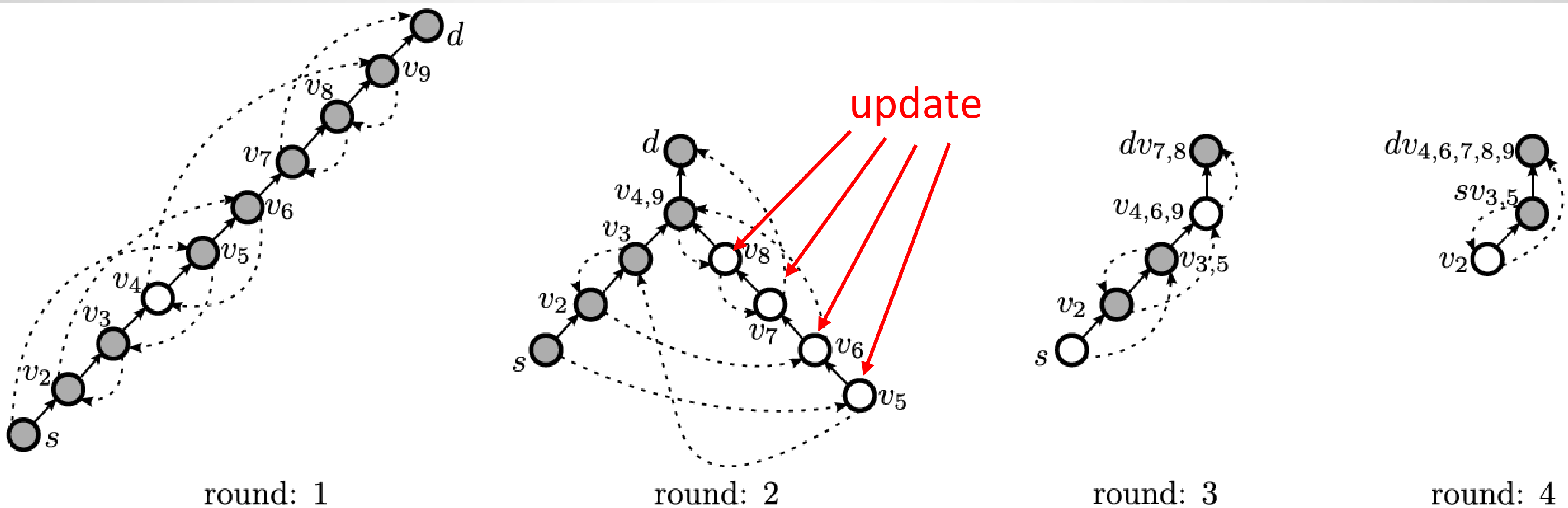
Prune

Shortcut

Prune

Greedily choose far-reaching (independent) forward edges.

Peacock in Action



Shortcut

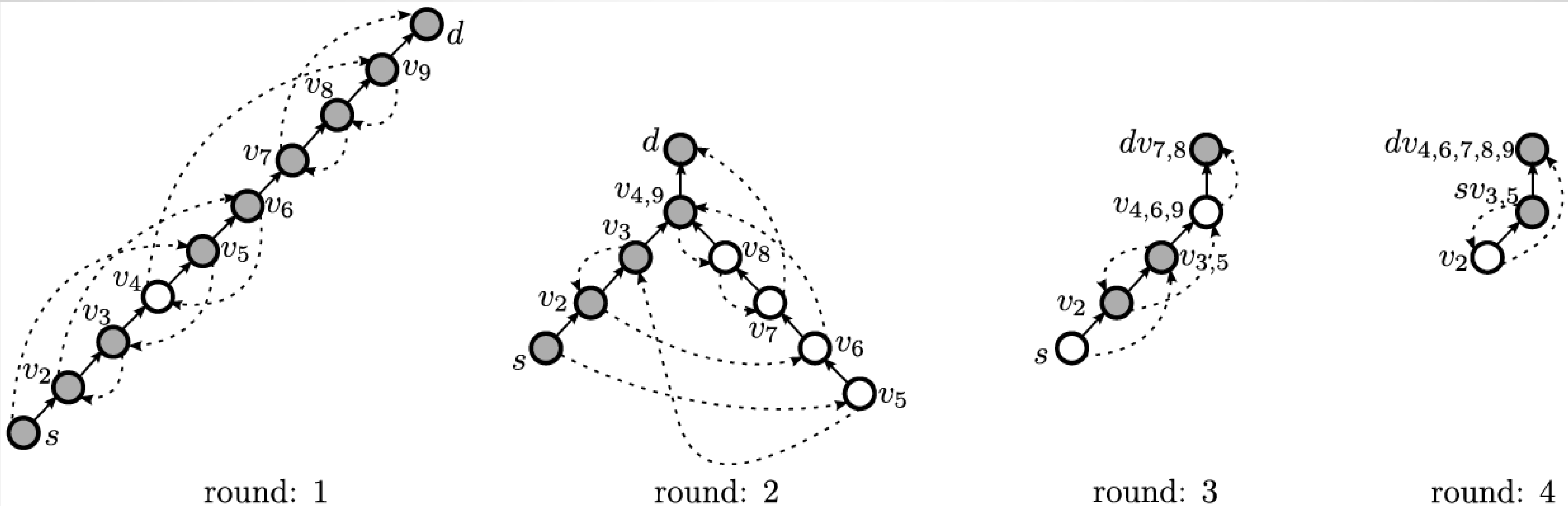
Prune

Shortcut

Prune

R1 generated many nodes in branches which can be updated simultaneously!

Peacock in Action



Shortcut

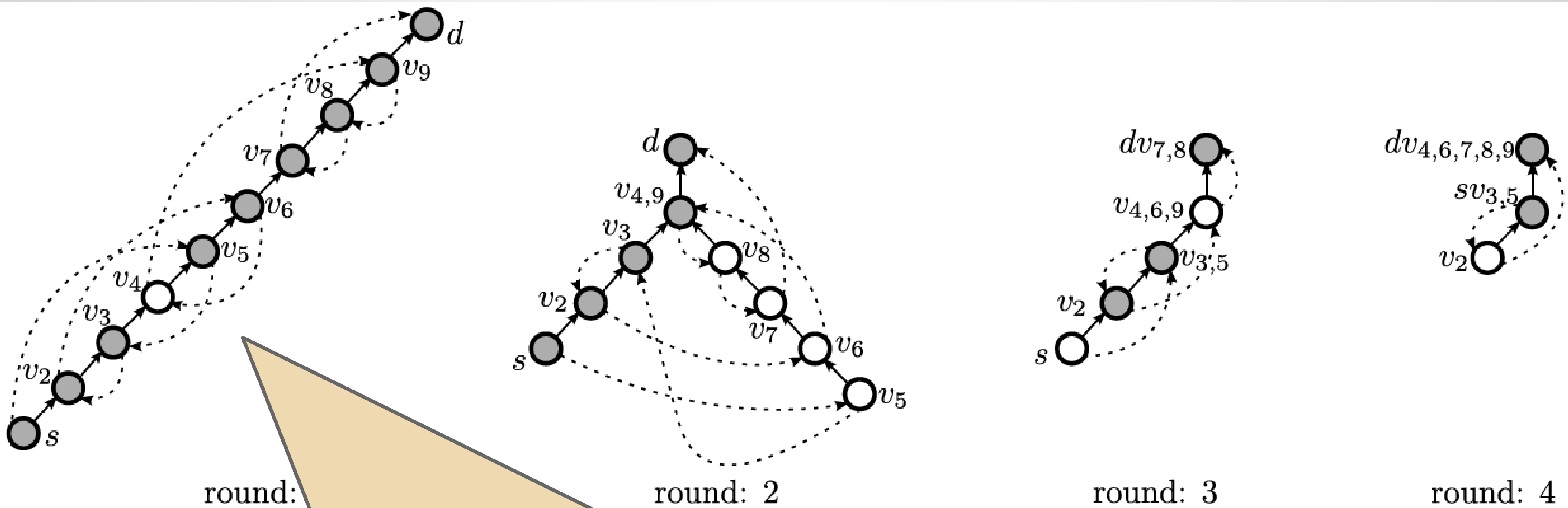
Prune

Shortcut

Prune

Line re-established!
(all merged with a
node on the s-d-path)

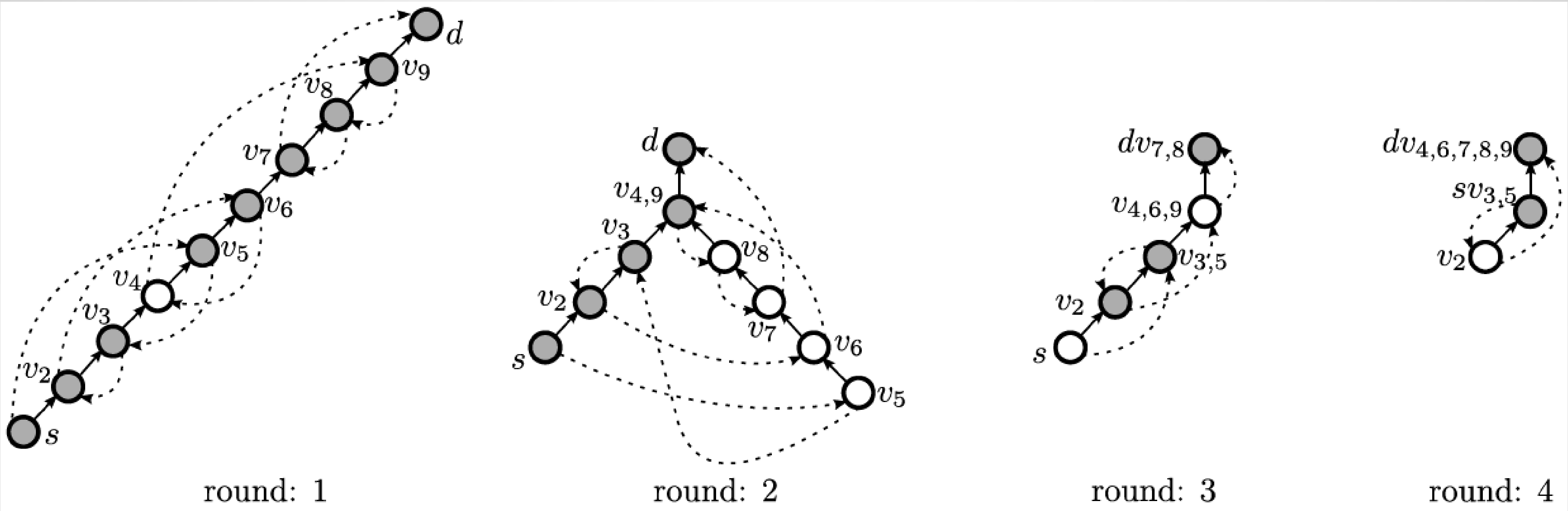
Peacock in Action



Peacock orders nodes wrt to distance: edge of length x can block at most 2 edges of length x, so distance 2x.

Prune

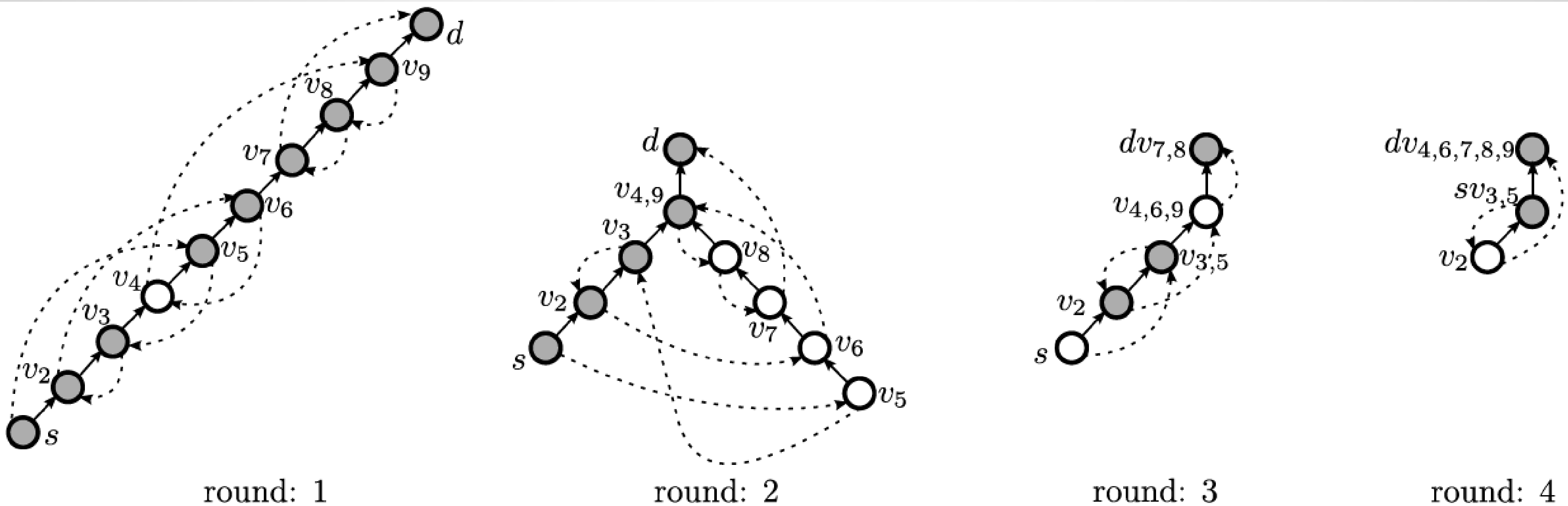
Peacock in Action



Shortcut Prune Shortcut Prune

At least 1/3 of nodes merged in each round pair (shorter s-d path): logarithmic runtime!

Peacock in Action

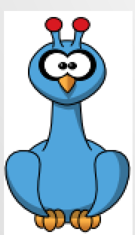


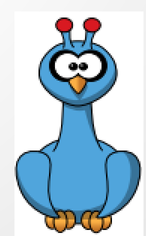
Shortcut

Prune

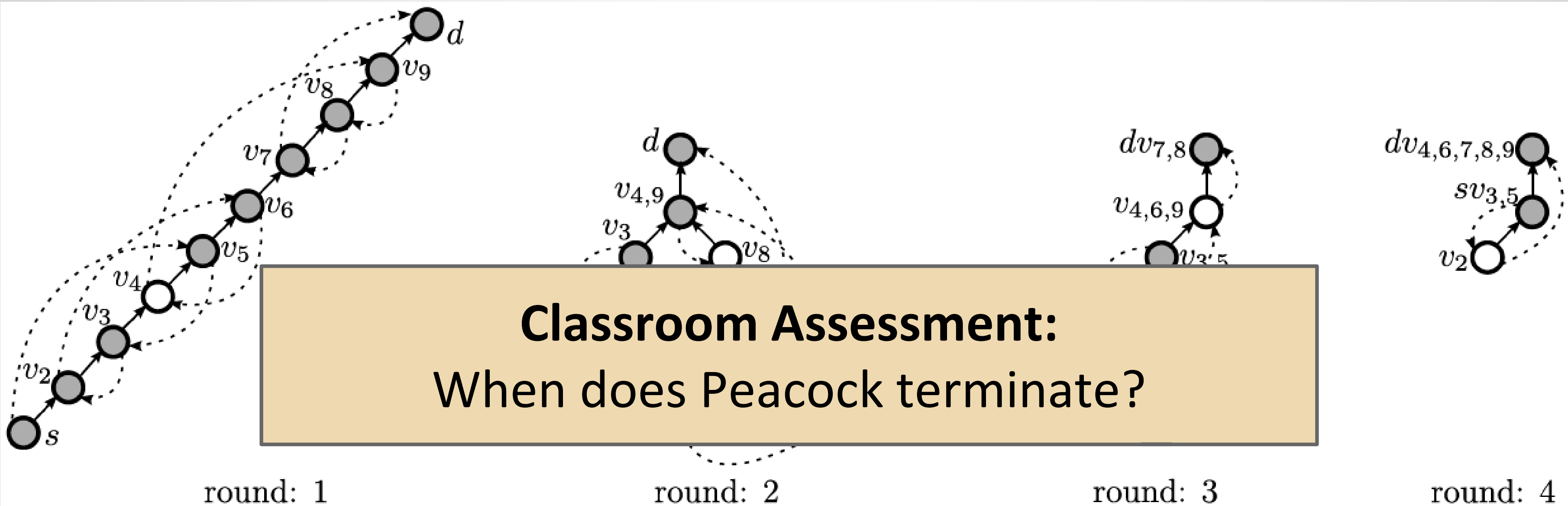
Shortcut

Prune





Peacock in Action



Classroom Assessment:

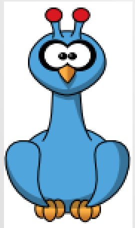
When does Peacock terminate?

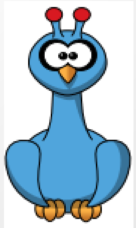
Shortcut

Prune

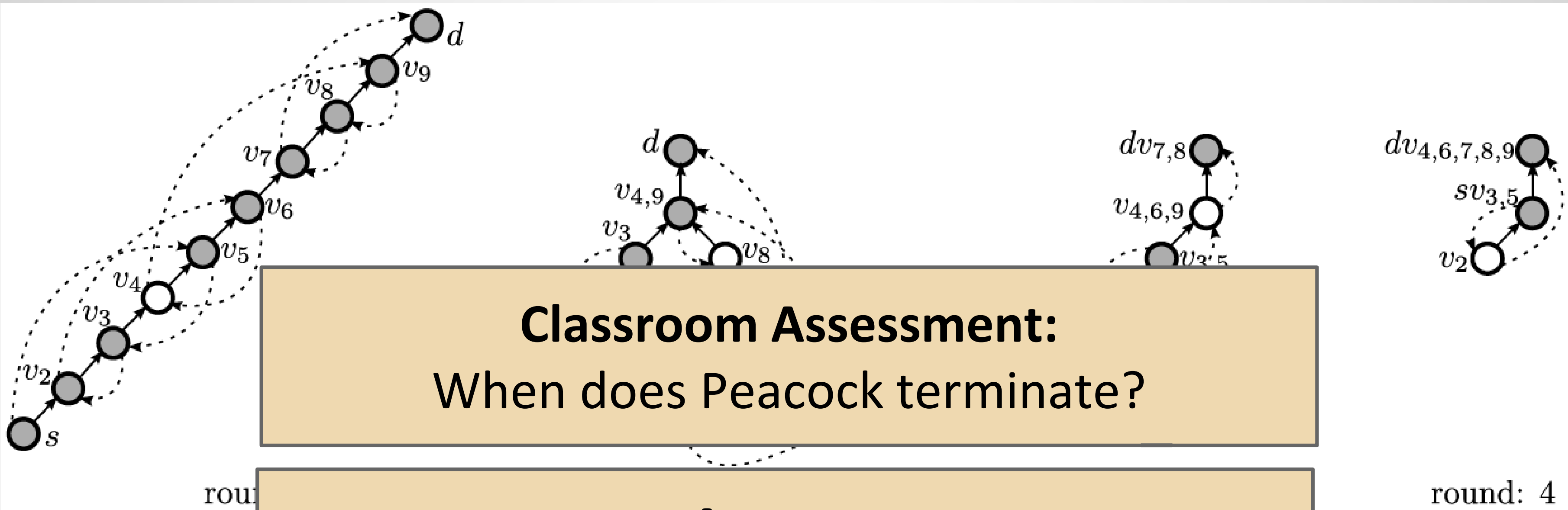
Shortcut

Prune





Peacock in Action

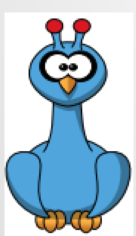


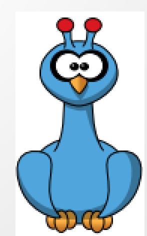
Classroom Assessment:
When does Peacock terminate?

Answer:
Only in odd rounds: then s-d merged

Shor

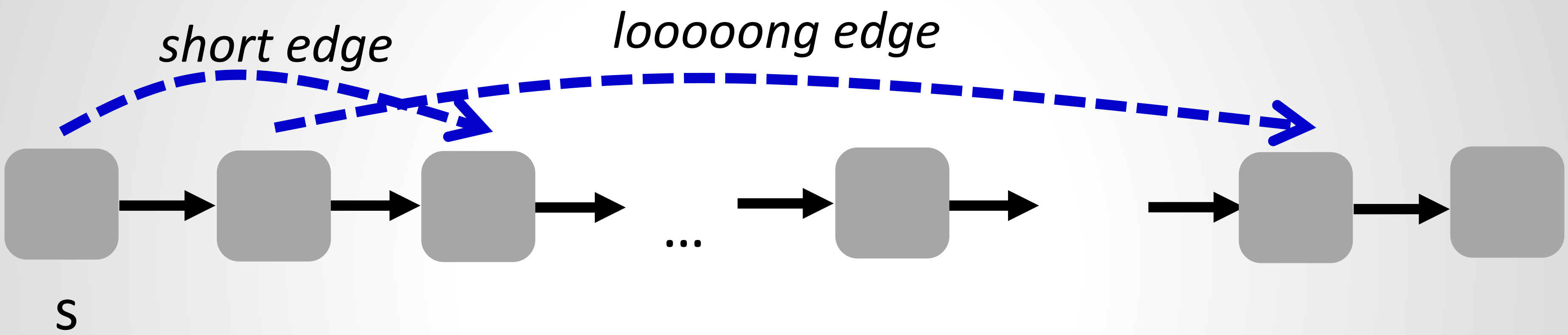
Prune



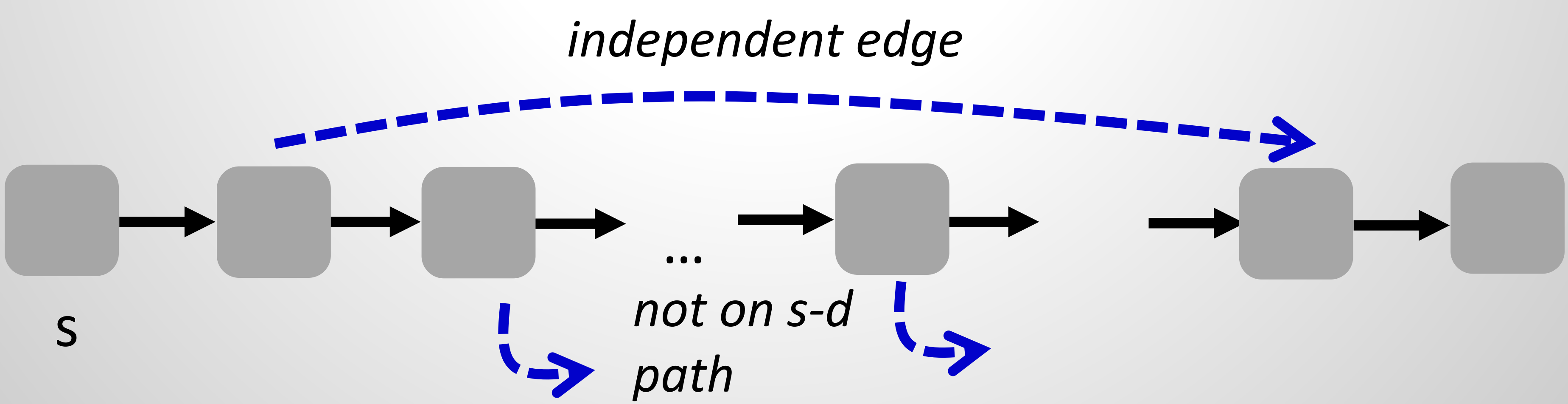



Why not update two non-independent edges?

- Don't update all FF edges: A short edge may not reduce distance to source if it jumps over a long edge



- Can update all fwd edges starting in interval



Conclusion

- Programmable and virtualized networks offer fundamental algorithmic problems

- Regarding network updates, so far we know:
  - Strong LF:
    - Greedy arbitrarily bad (up to n rounds) and NP-hard
    - 2 rounds easy
    - 3 rounds hard
  - Relaxed LF:
    - Peacock solves any scenario in O(log n) rounds
    - Computational results indicate that # rounds grows
  - LF and WPE may conflict

Thank you!

And thanks to co-authors: Arne Ludwig, Jan Marcinkowski

as well as Marco Canini, Damien Foucard, Petr Kuznetsov, Dan Levin, Matthias Rost, Jukka Suomela

and more recently Saeed Amiri, Szymon Dudycz, Felix Widmaier

Own References

Scheduling Loop-free Network Updates: It's Good to Relax!
Arne Ludwig, Jan Marcinkowski, and Stefan Schmid.
ACM Symposium on Principles of Distributed Computing (**PODC**), Donostia-San Sebastian, Spain, July 2015.

Medieval: Towards A Self-Stabilizing, Plug & Play, In-Band SDN Control Network (Demo Paper)
Liron Schiff, Stefan Schmid, and Marco Canini.
ACM Sigcomm Symposium on SDN Research (**SOSR**), Santa Clara, California, USA, June 2015.

A Distributed and Robust SDN Control Plane for Transactional Network Updates
Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid.
34th IEEE Conference on Computer Communications (**INFOCOM**), Hong Kong, April 2015.

Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies
Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid.
13th ACM Workshop on Hot Topics in Networks (**HotNets**), Los Angeles, California, USA, October 2014.

Provable Data Plane Connectivity with Local Fast Failover: Introducing OpenFlow Graph Algorithms
Michael Borokhovich, Liron Schiff, and Stefan Schmid.
ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (**HotSDN**), Chicago, Illinois, USA, August 2014.

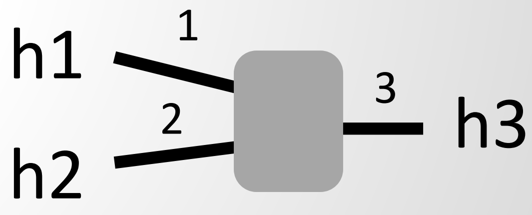
Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks
Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, and Anja Feldmann.
USENIX Annual Technical Conference (**ATC**), Philadelphia, Pennsylvania, USA, June 2014.

The SDN *Hello World*:
MAC Learning
(Even a single switch scenario is non-trivial!)

Distributed Computing Fail: Updating a Single Switch

Already updating a single switch from a single controller is non-trivial!

- Fundamental networking task: MAC learning
  - Flood packets sent to unknown destinations
  - Learn host's location when it sends packets
- Example
  - h1 sends to h2:
  - h3 sends to h1:
  - h1 sends to h3:



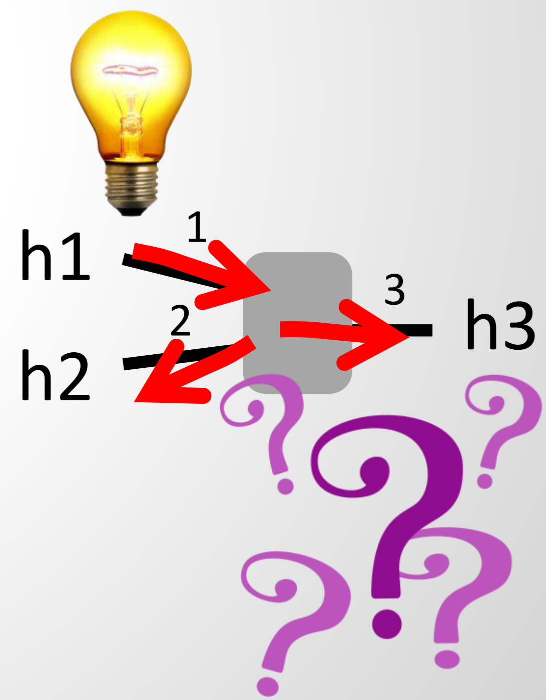
Thanks to Jennifer Rexford for example!

Distributed Computing Fail: Updating a Single Switch

Already updating a single switch from a single controller is non-trivial!

- Fundamental networking task: MAC learning
  - Flood packets sent to unknown destinations
  - Learn host’s location when it sends packets
- Example
  - h1 sends to h2:

    flood, learn (h1,p1)
  - h3 sends to h1:
  - h1 sends to h3:



Thanks to Jennifer Rexford for example!

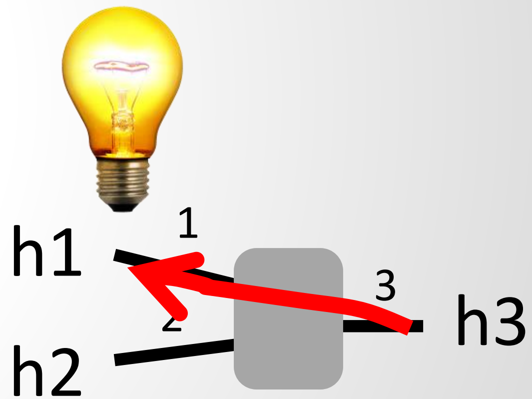
Distributed Computing Fail: Updating a Single Switch

Already updating a single switch from a single controller is non-trivial!

- Fundamental networking task: MAC learning
  - Flood packets sent to unknown destinations
  - Learn host's location when it sends packets
- Example
  - h1 sends to h2:

    flood, learn (h1,p1)
  - h3 sends to h1:

    forward to p1, learn (h3,p3)
  - h1 sends to h3:



Thanks to Jennifer Rexford for example!

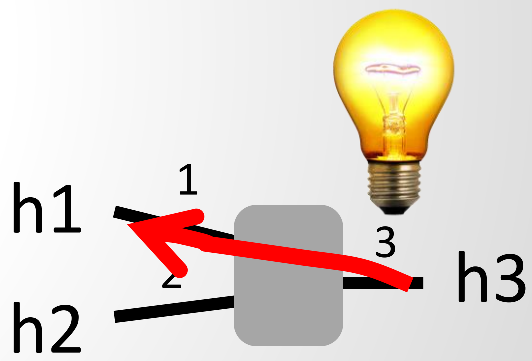
Distributed Computing Fail: Updating a Single Switch

Already updating a single switch from a single controller is non-trivial!

- Fundamental networking task: MAC learning
  - Flood packets sent to unknown destinations
  - Learn host's location when it sends packets
- Example
  - h1 sends to h2:

    flood, learn (h1,p1)
  - h3 sends to h1:

    forward to p1, learn (h3,p3)
  - h1 sends to h3:



Thanks to Jennifer Rexford for example!

Distributed Computing Fail: Updating a Single Switch

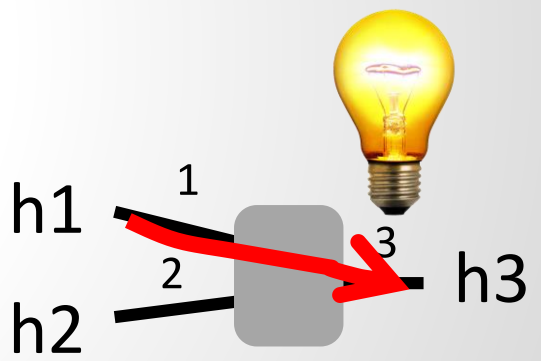
Already updating a single switch from a single controller is non-trivial!

- Fundamental networking task: MAC learning
  - Flood packets sent to unknown destinations
  - Learn host's location when it sends packets
- Example
  - h1 sends to h2:

    flood, learn (h1,p1)
  - h3 sends to h1:

    forward to p1, learn (h3,p3)
  - h1 sends to h3:

    forward to p3



Thanks to Jennifer Rexford for example!

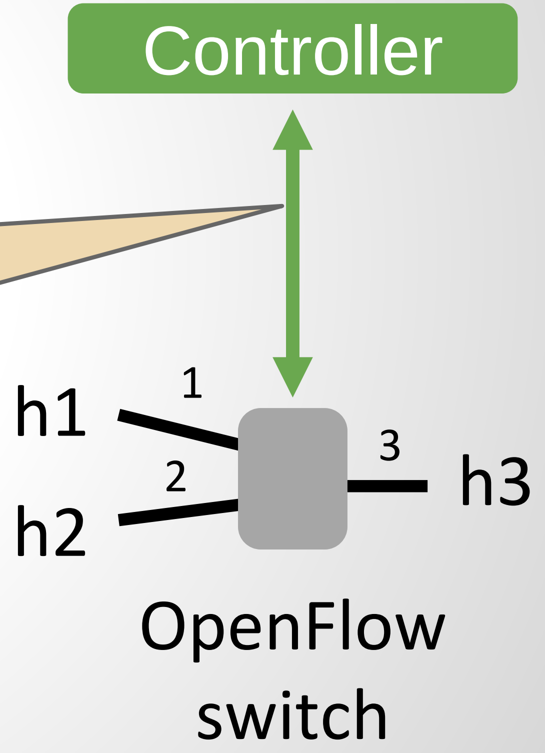
Distributed Computing Fail: Updating a Single Switch

Already updating a single switch from a single controller is non-trivial!

- Fundamental task: MAC learning
  - Flood packets sent to unknown destinations

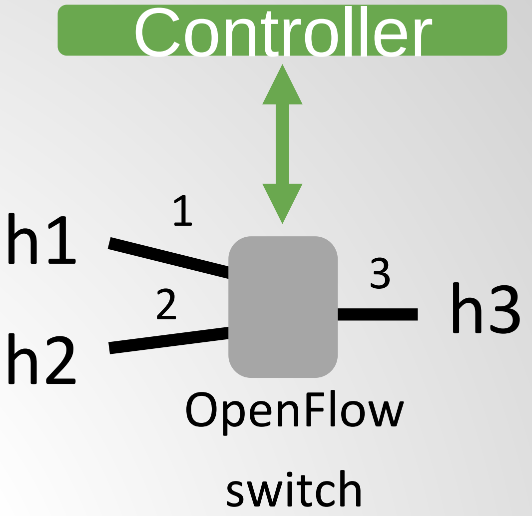
Now: how to do via controller?
Install rules as you learn!
And match on host address and port.

  - h3 sends to h1:
    forward to p1, learn (h3,p3)
  - h1 sends to h3:
    forward to p3



Thanks to Jennifer Rexford for example!

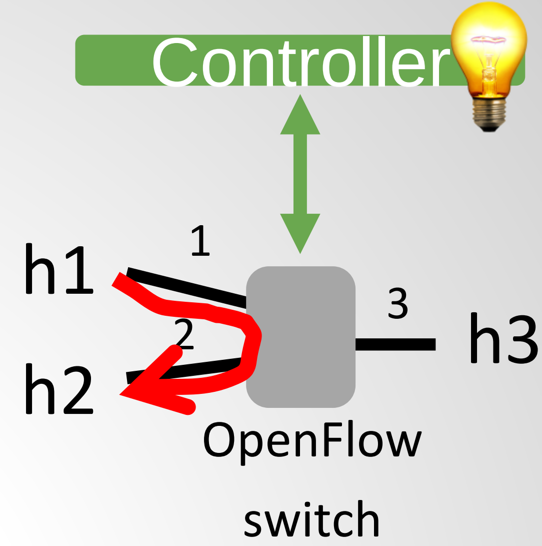
Example: SDN MAC Learning Done Wrong



- Initial rule *: Send everything to controller

- What happens when h1 sends to h2?

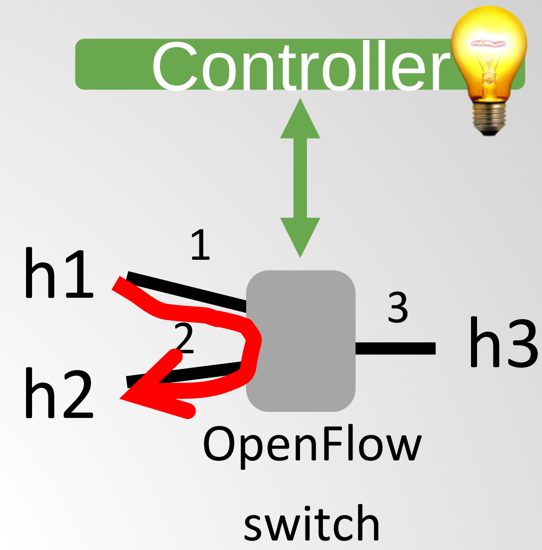
Example: SDN MAC Learning Done Wrong



- Initial rule *: Send everything to controller

- What happens when h1 sends to h2?
  - Controller learns that h1@p1 and installs rule on switch!

Example: SDN MAC Learning Done Wrong



- Initial rule *: Send everything to controller

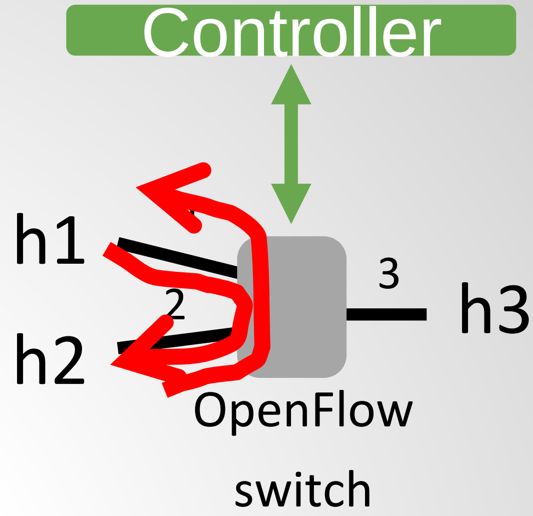
Pattern	Action
*	Send to controller

h1 sends to h2

Pattern	Action
dstmac=h1	Forward(1)
*	Send to controller

- What happens when h1 sends to h2?
  - Controller learns that h1@p1 and installs rule on switch!

Example: SDN MAC Learning Done Wrong



- Initial rule *: Send everything to controller

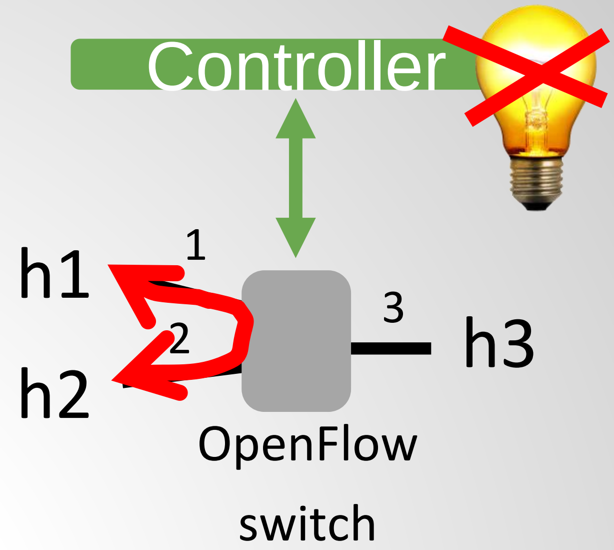
Pattern	Action
*	Send to controller

h1 sends to h2

Pattern	Action
dstmac=h1	Forward(1)
*	Send to controller

- What happens when h2 sends to h1?

Example: SDN MAC Learning Done Wrong



- Initial rule *: Send everything to controller

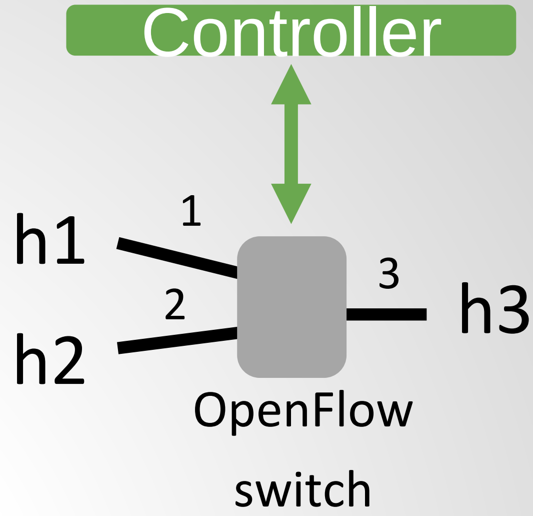
Pattern	Action
*	Send to controller

h1 sends to h2

Pattern	Action
dstmac=h1	Forward(1)
*	Send to controller

- What happens when h2 sends to h1?
  - Switch knows destination: message forwarded to h1
  - No controller interaction, no new rule for h2

Example: SDN MAC Learning Done Wrong



- Initial rule *: Send everything to controller

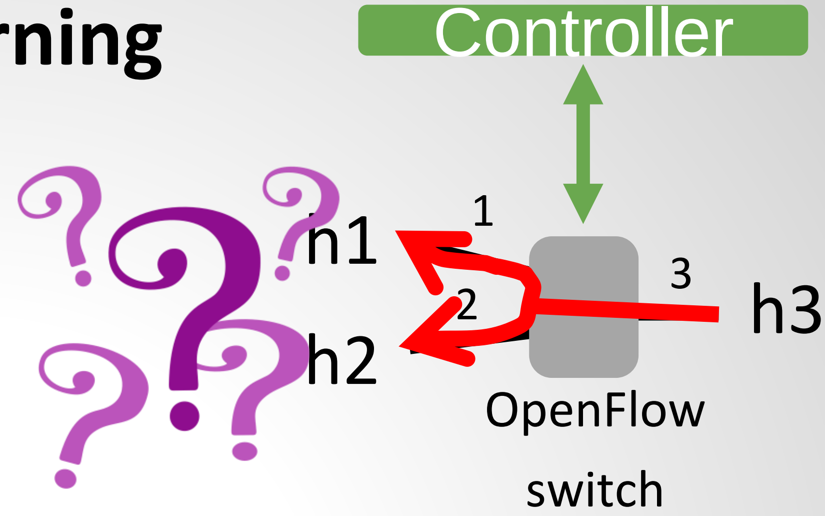
Pattern	Action
*	Send to controller

h1 sends to h2

Pattern	Action
dstmac=h1	Forward(1)
*	Send to controller

- What happens when h2 sends to h1?
  - Switch knows destination: message forwarded to h1
  - No controller interaction, no new rule for h2
- What happens when h3 sends to h2?

Example: SDN MAC Learning Done Wrong



- Initial rule *: Send everything to controller

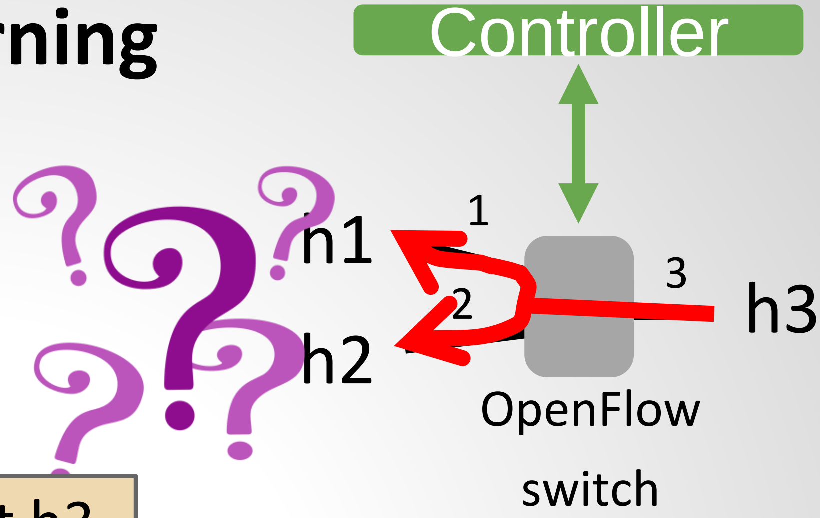
Pattern	Action
*	Send to controller

h1 sends to h2

Pattern	Action
dstmac=h1	Forward(1)
*	Send to controller

- What happens when h2 sends to h1?
  - Switch knows destination: message forwarded to h1
  - No controller interaction, no new rule for h2
- What happens when h3 sends to h2?
  - Flooded! Controller did not put the rule to h2.

Example: SDN MAC Learning Done Wrong



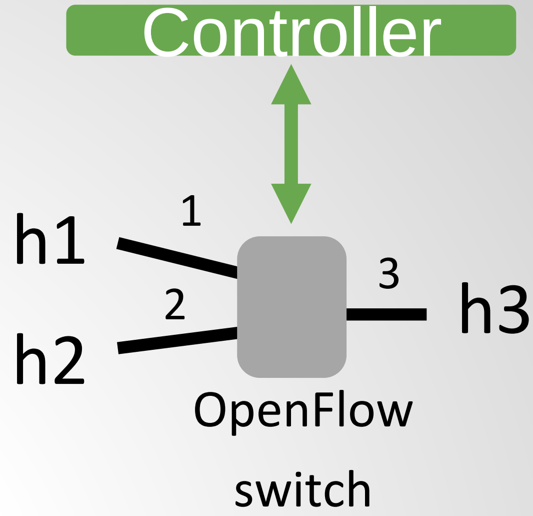
- Initial rule *: Send everything to controller

> Controller however does learn about h3. Then answer from h2 missed by controller too: all future requests to h2 flooded?!?

Pattern	Action
dstmac=h1	Forward(1)
*	Send to controller

- Wha pens when h2 sends to h1?
  - Switc ows destination: message forwarded to h1
  - No cont oller interaction, no new rule for h2
- What happens when h3 sends to h2?
  - Flooded! Controller did not put the rule to h2!

Example: SDN MAC Learning Done Wrong



- Initial rule *: Send everything to controller

A bug in early controller software.
Hard to catch! A performance issue, not a consistency one
(arguably a key strength of SDN?).

- What happens when h2 sends to h1?
  - Switch knows destination: message forwarded to h1
  - No controller interaction, no new rule for h2
- What happens when h3 sends to h2?
  - Flooded! Controller did not put the rule to h2!