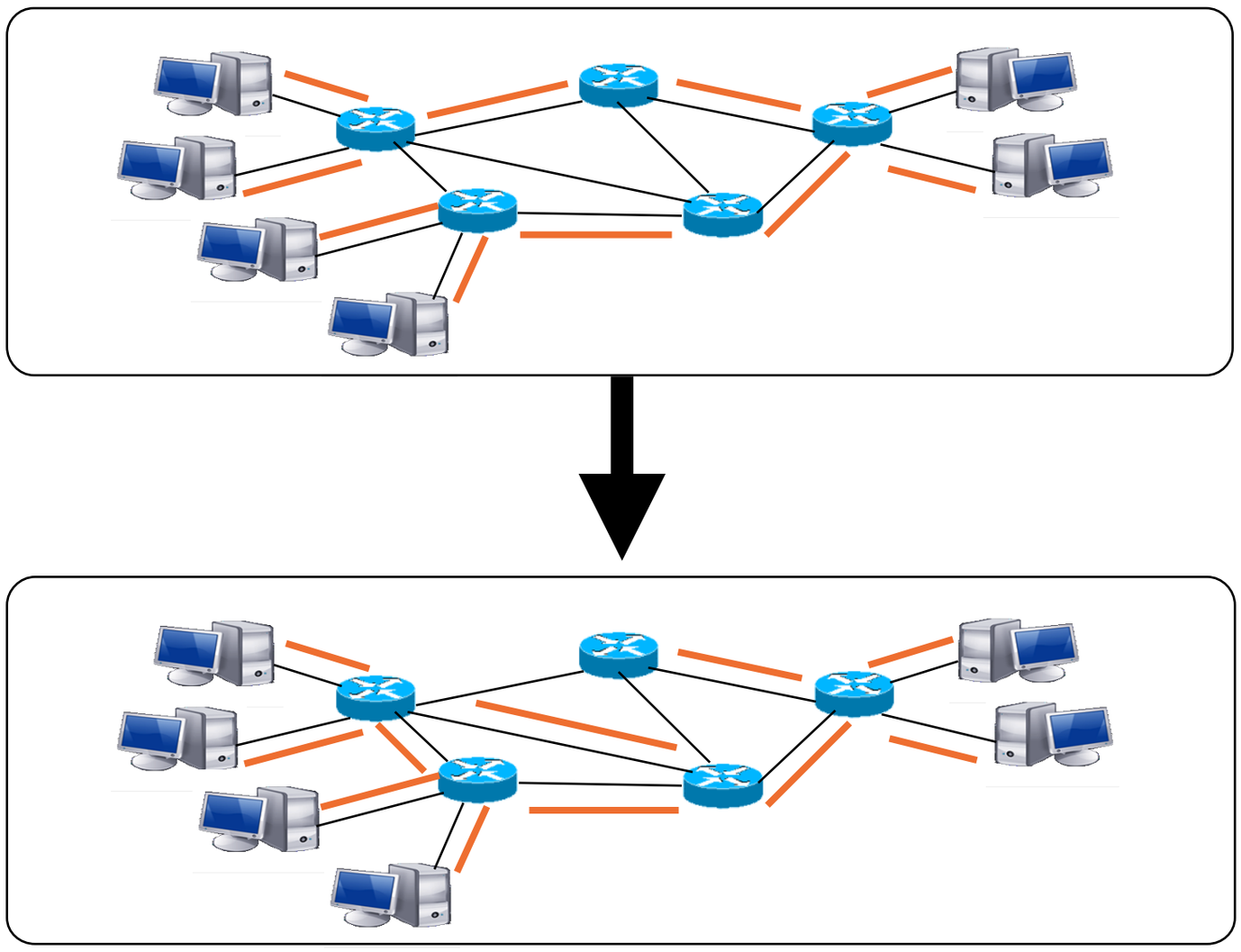
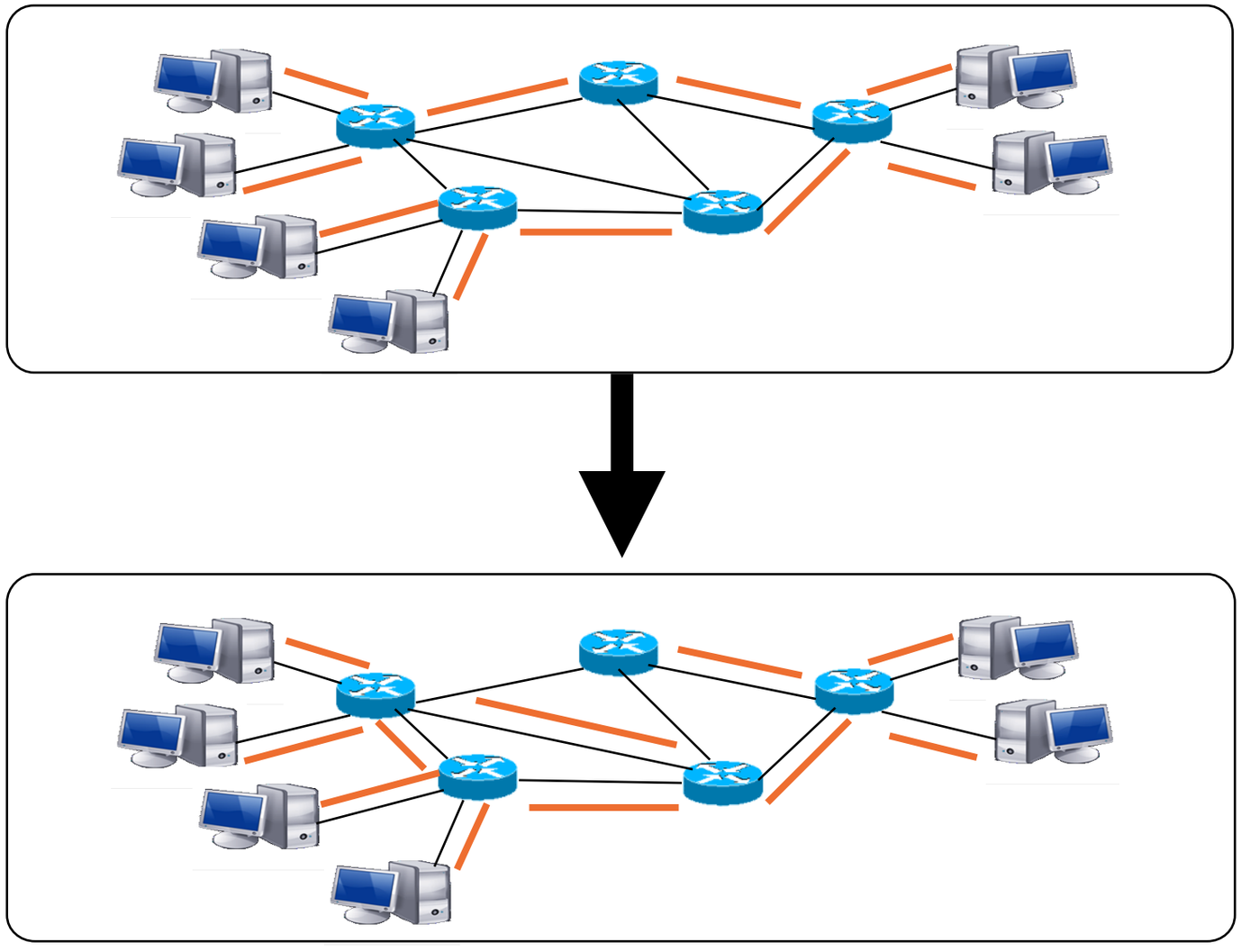
Enforcing Customizable Consistency Properties in Software-Defined Networks

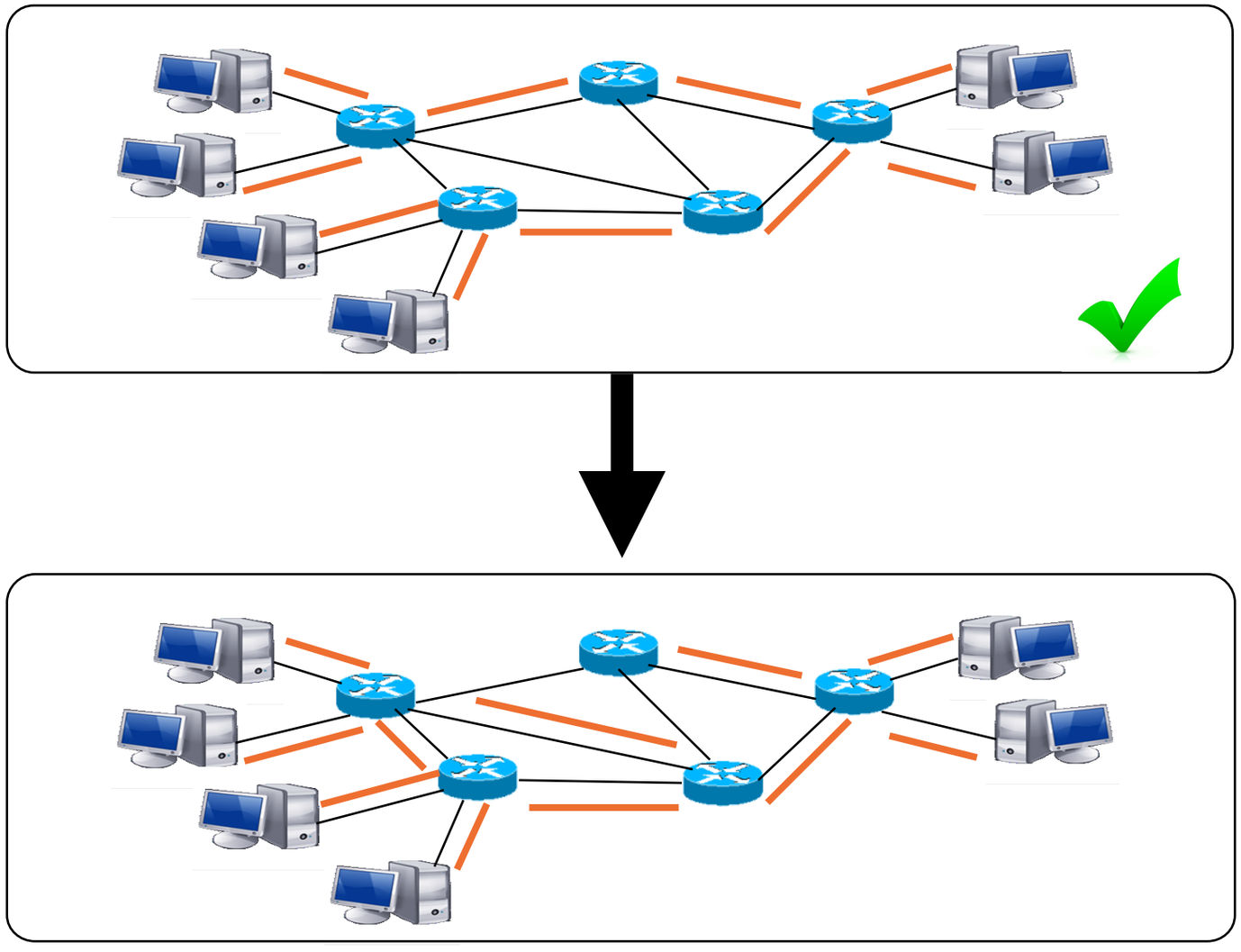
Wenxuan Zhou, Dong Jin, Jason Croft,
Matthew Caesar, Brighten Godfrey





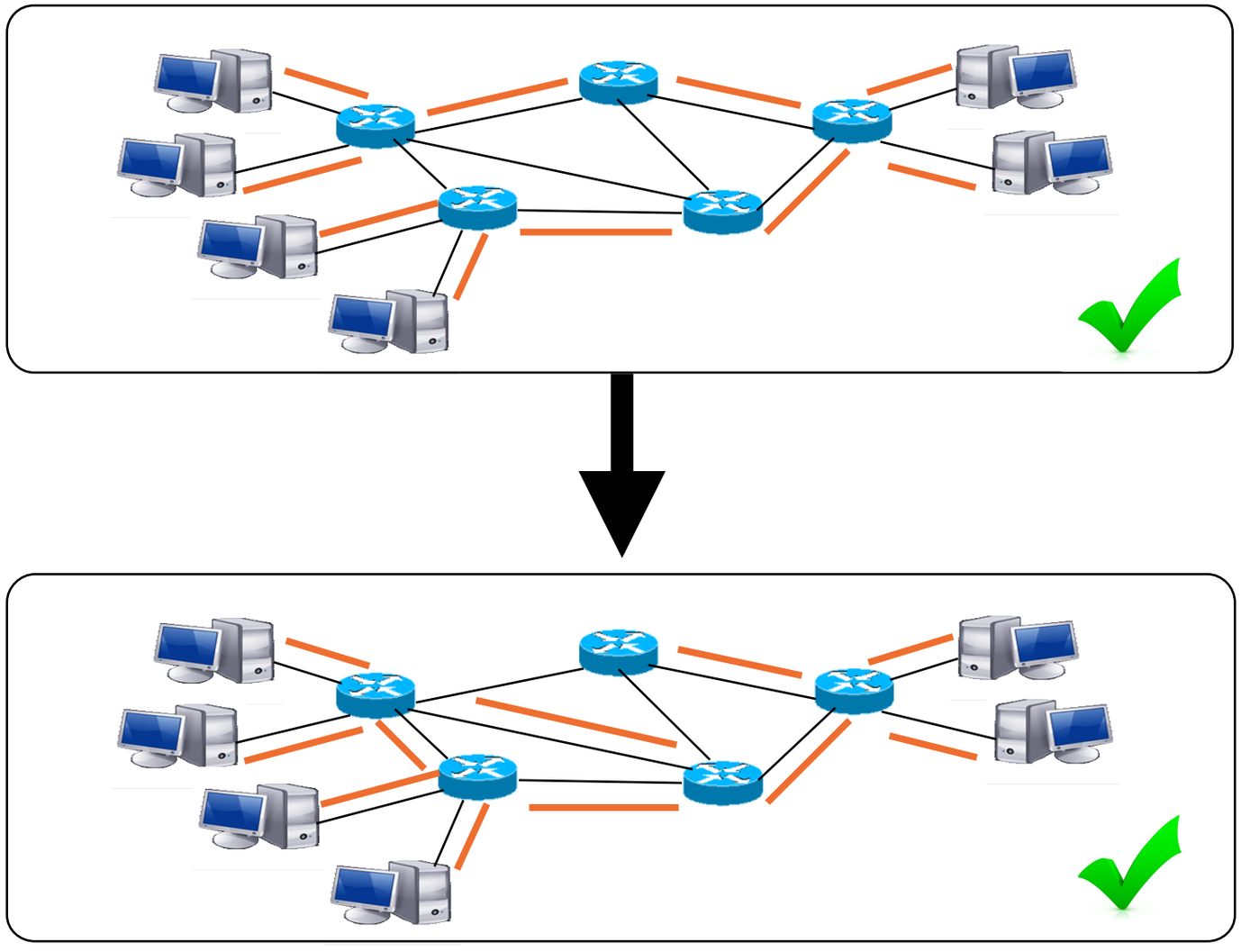
Network changes

- control applications,
- changes in traffic load,
- system upgrades,
- …



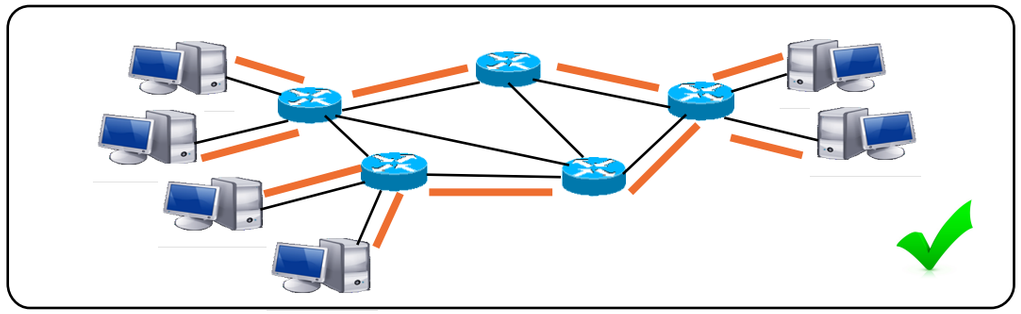
Network changes

- control applications,
- changes in traffic load,
- system upgrades,
- …

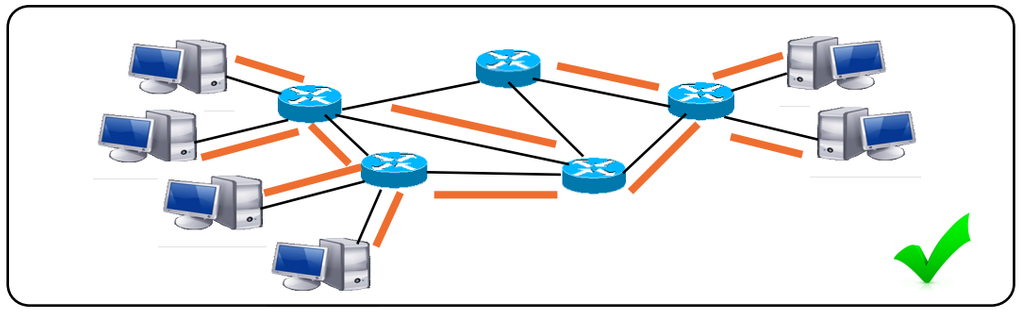


Network changes

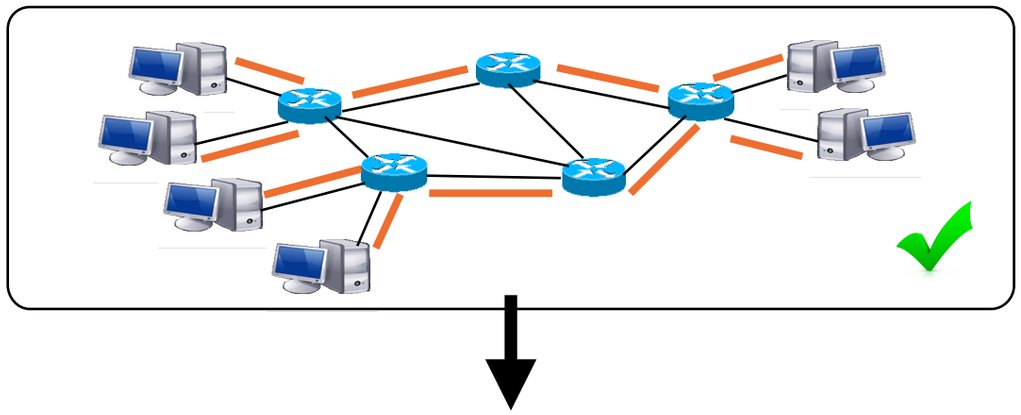
- control applications,
- changes in traffic load,
- system upgrades,
- …



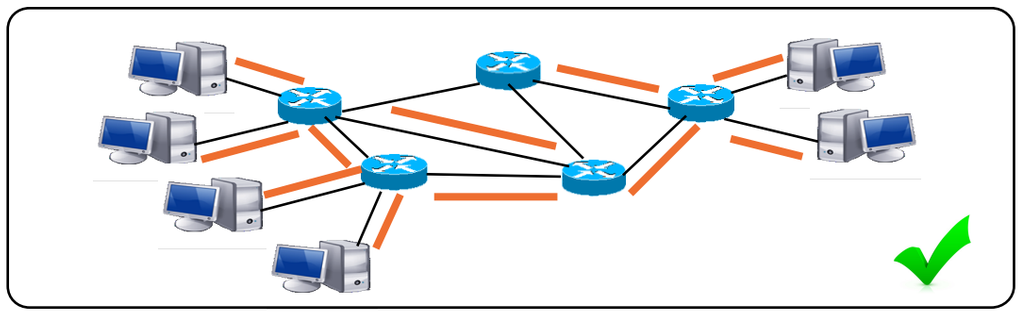
Old correct behavior



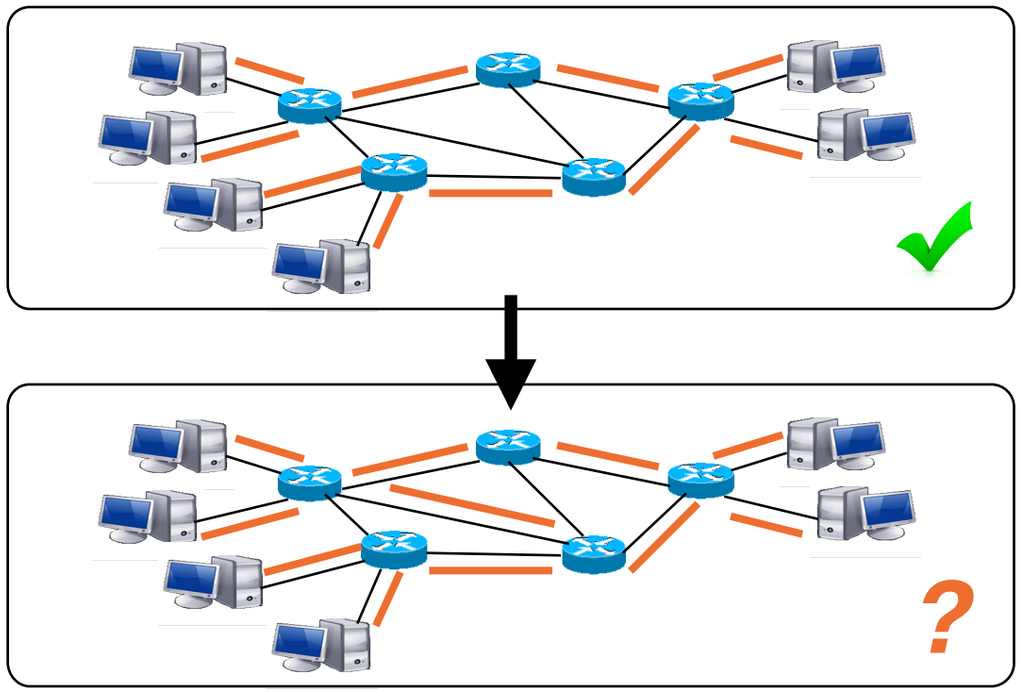
Next correct behavior



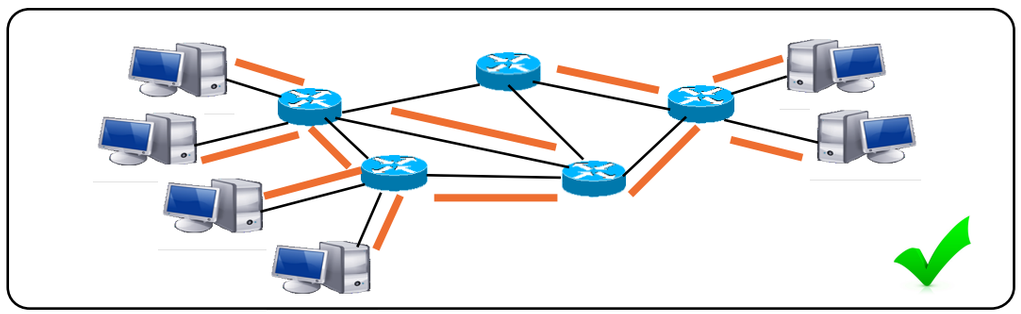
Old correct behavior



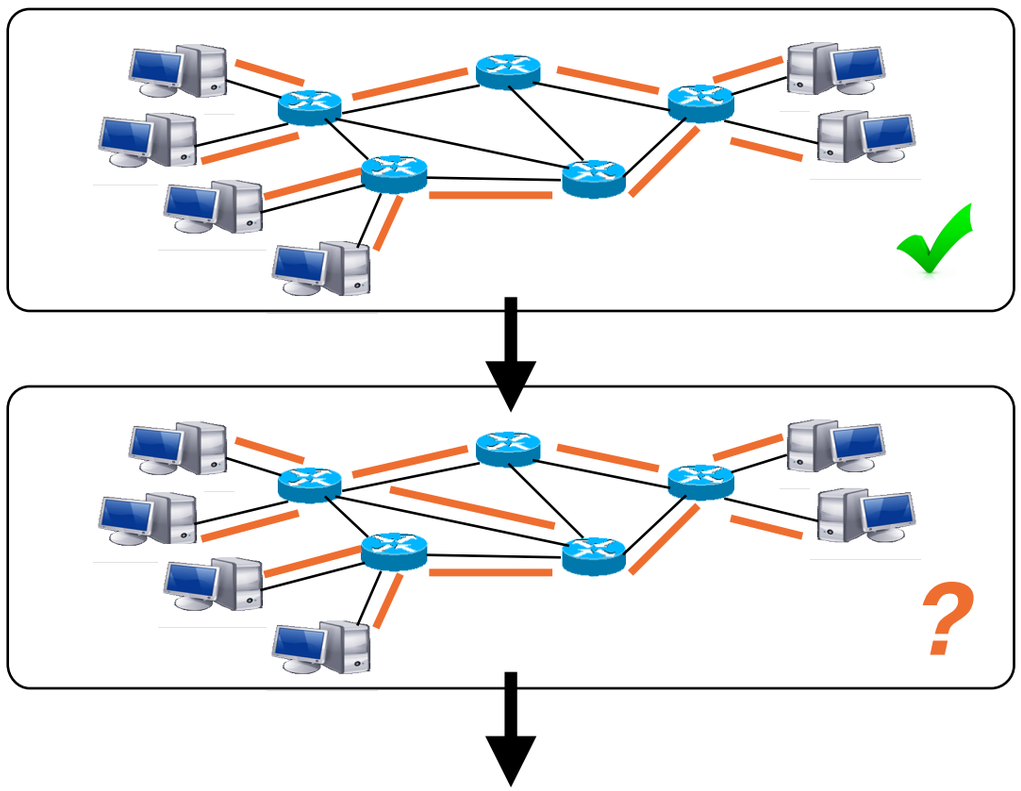
Next correct behavior



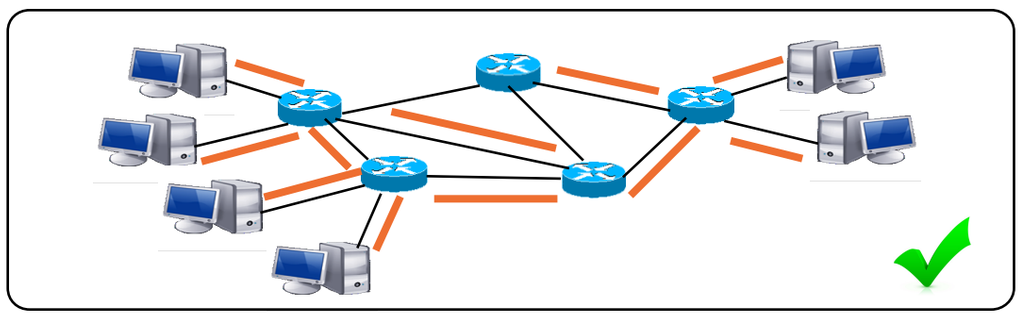
Old correct behavior



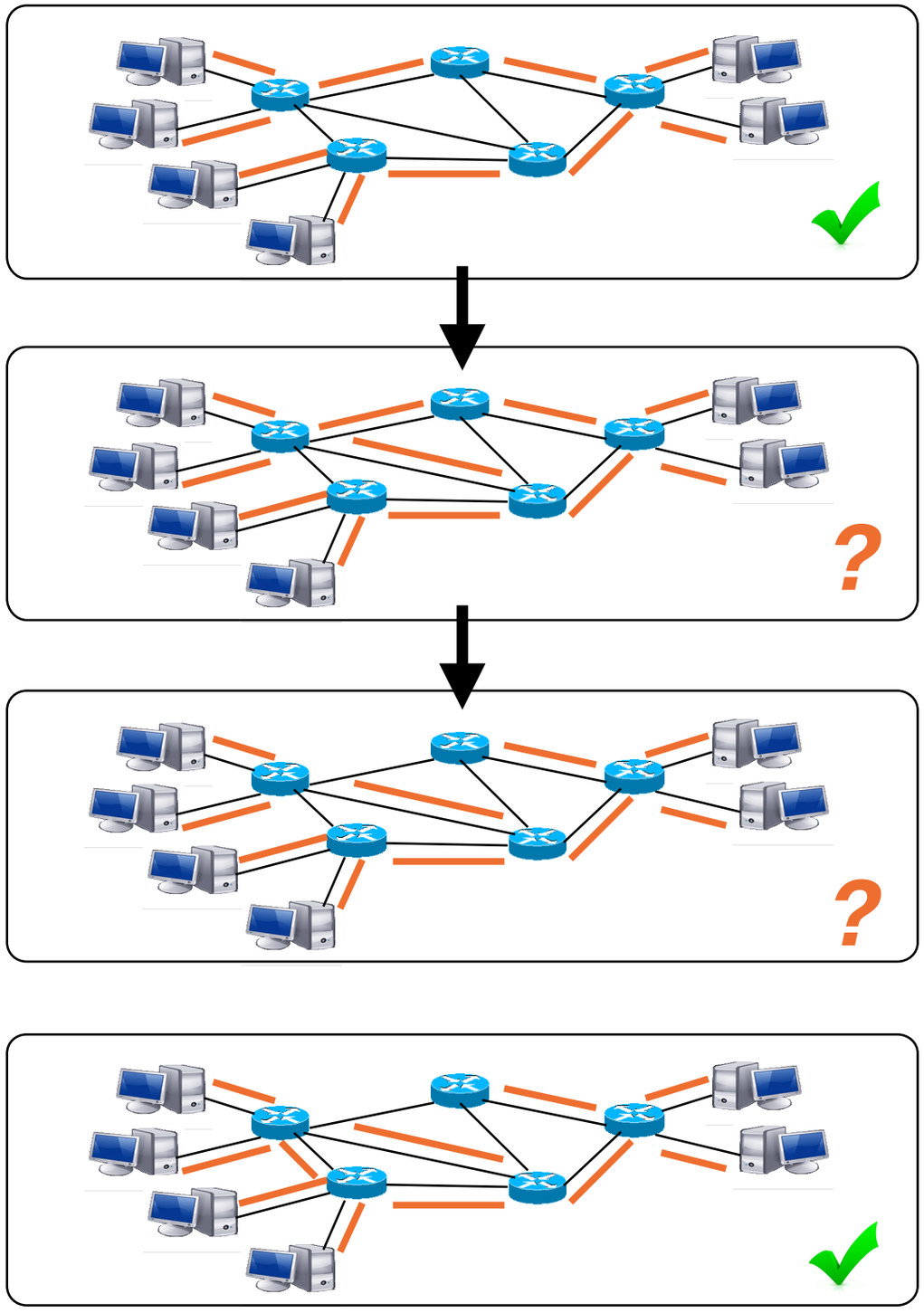
Next correct behavior



Old correct behavior

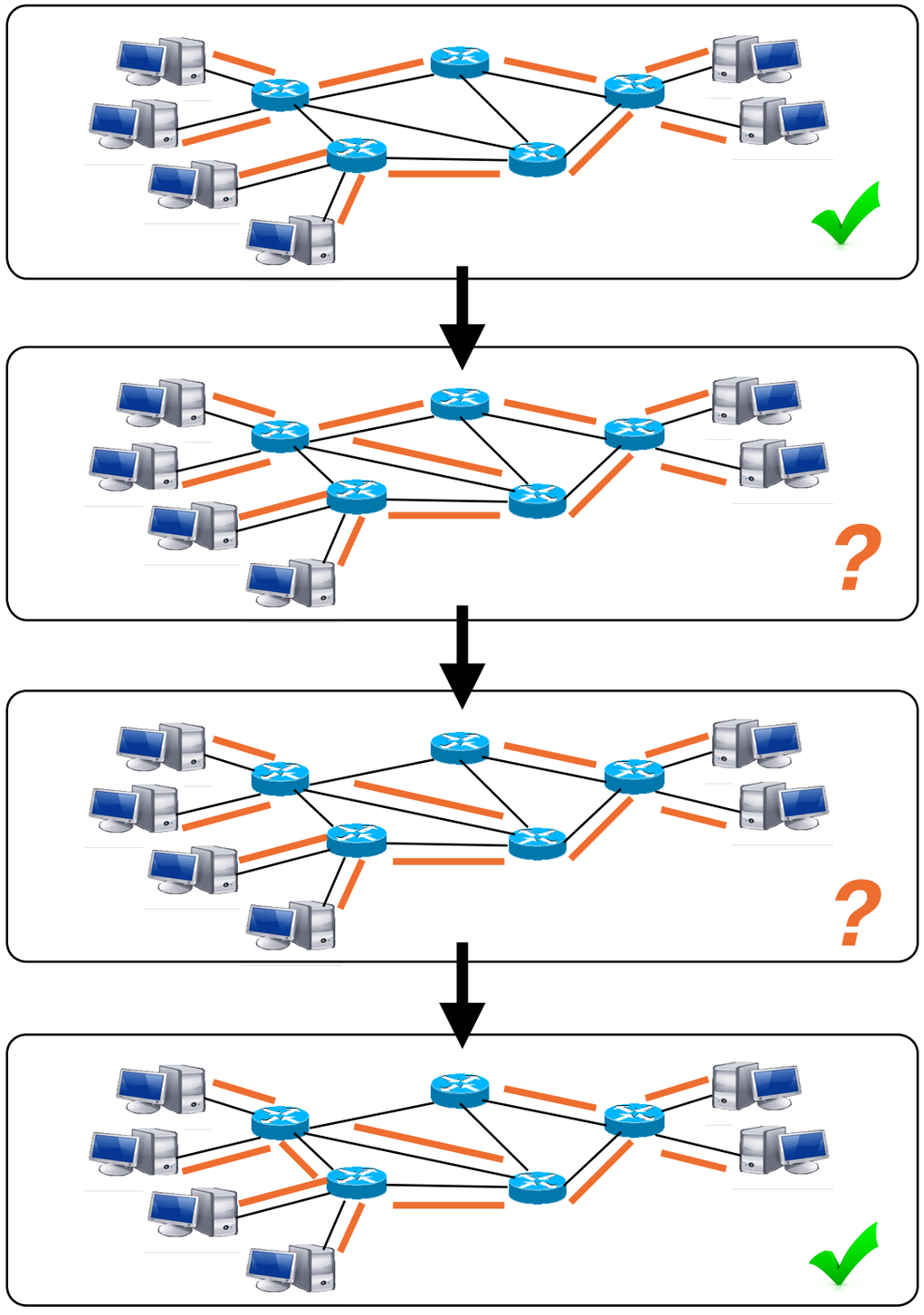


Next correct behavior



Old correct behavior

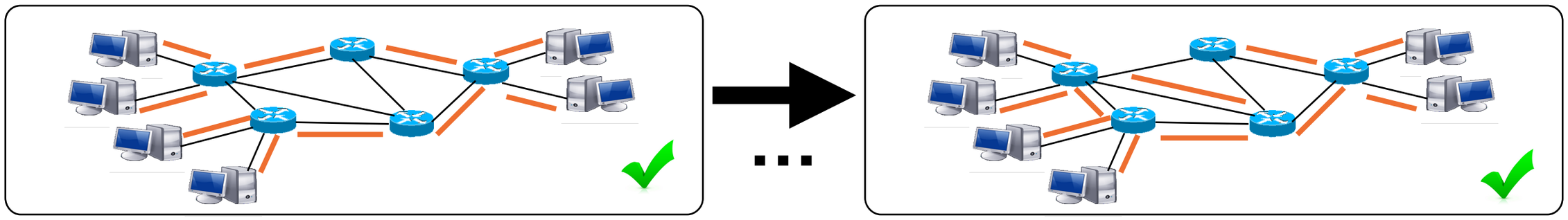
Next correct behavior



Old correct behavior

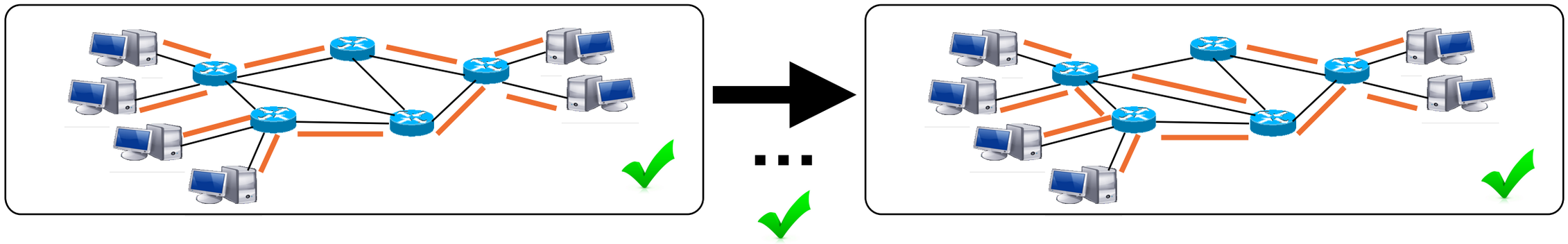
Next correct behavior

Towards a solution



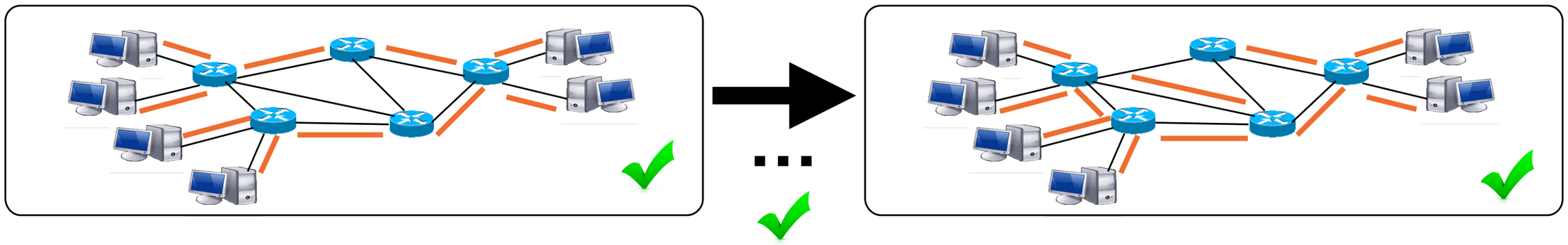
1. Correctness at every step

Towards a solution



1. Correctness at every step

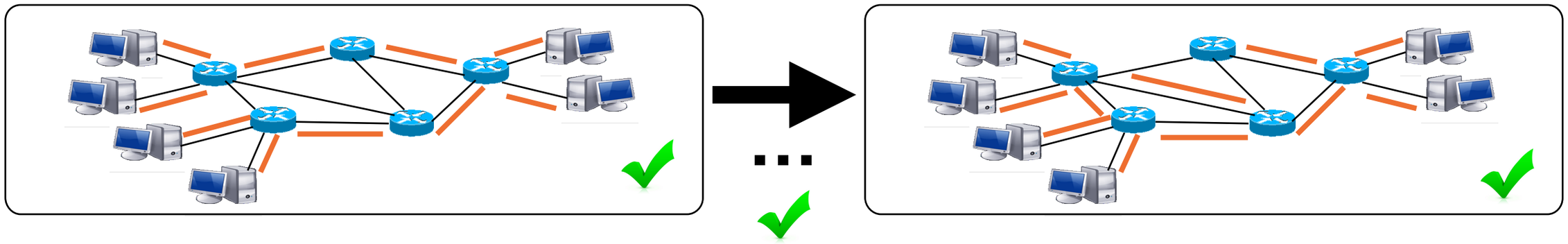
Towards a solution



1. Correctness at every step

2. With efficient update installation

Towards a solution



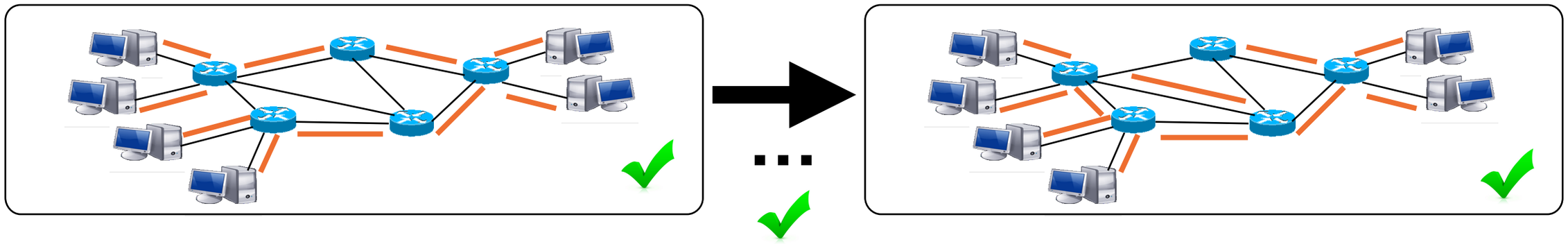
1. Correctness at every step

2. With efficient update installation

What is Correctness?

- firewall traversal,
- access control,
- balanced load,
- loop freedom,
- …

Towards a solution



1. Correctness at every step

2. With efficient update installation

3. Customizable properties

What is Correctness?

- firewall traversal,
- access control,
- balanced load,
- loop freedom,
- …

Problem Statement

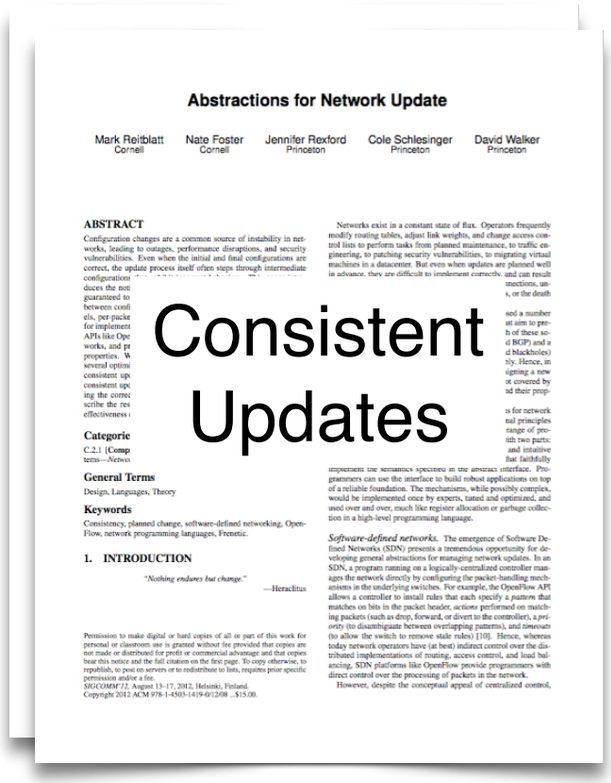
1. Consistency at every step
2. Efficient updates installation
3. Customizable consistency properties

Problem Statement

1. Consistency at every step
2. Efficient updates installation
3. Customizable consistency properties

Is it possible to **efficiently** ensure **customizable** correctness properties as the network **evolves**?

Prior Work



Abstractions for Network Update

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

ABSTRACT

Configuration changes are a common source of instability in networks, leading to outages, performance disruptions, and security vulnerabilities. Even when the initial and final configurations are correct, the update process itself often steps through intermediate configurations

Networks exist in a constant state of flux. Operators frequently modify routing tables, adjust link weights, and change access control lists to perform tasks from planned maintenance, to traffic engineering, to patching security vulnerabilities, to migrating virtual machines in a datacenter. But even when updates are planned well in advance, they are difficult to implement correctly

General Terms

Design, Languages, Theory

Keywords

Consistency, planned change, software-defined networking, OpenFlow, network programming languages, Frenetic.

1. INTRODUCTION

"Nothing endures but change."

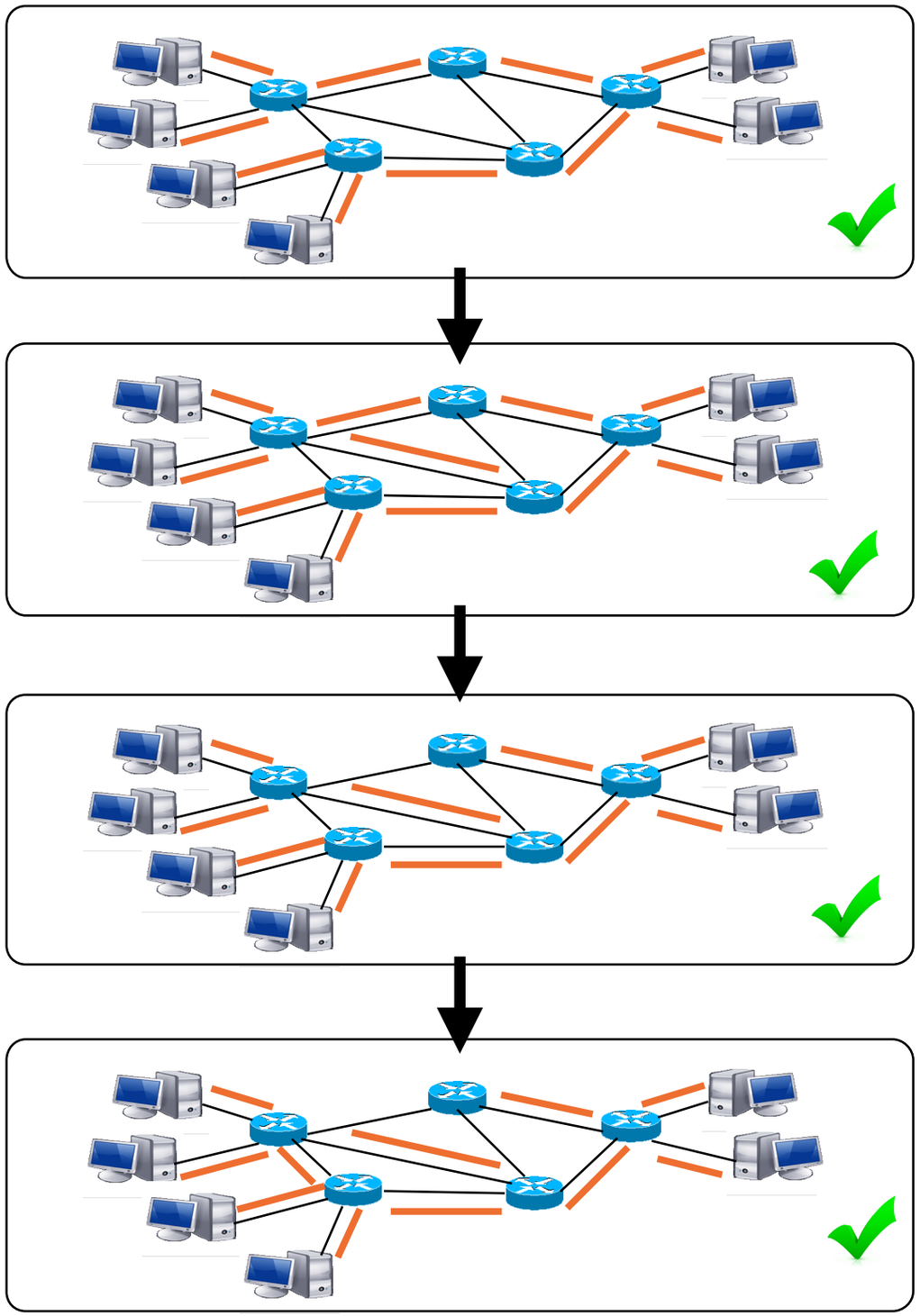
—Heraclitus

Software-defined networks. The emergence of Software Defined Networks (SDN) presents a tremendous opportunity for developing general abstractions for managing network updates. In an SDN, a program running on a logically-centralized controller manages the network directly by configuring the packet-handling mechanisms in the underlying switches. For example, the OpenFlow API allows a controller to install rules that each specify a *pattern* that matches on bits in the packet header, *actions* performed on matching packets (such as drop, forward, or divert to the controller), a *priority* (to disambiguate between overlapping patterns), and *timeouts* (to allow the switch to remove stale rules) [10]. Hence, whereas today network operators have (at best) indirect control over the distributed implementations of routing, access control, and load balancing, SDN platforms like OpenFlow provide programmers with direct control over the processing of packets in the network.

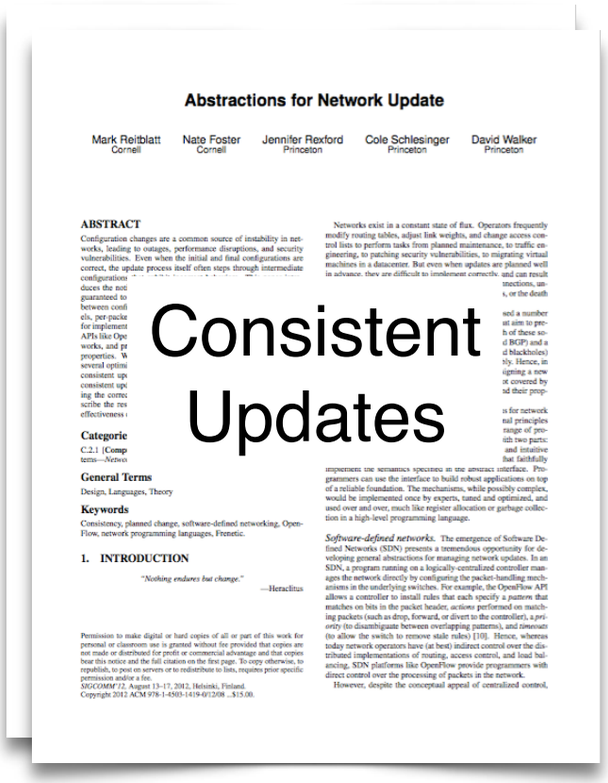
However, despite the conceptual appeal of centralized control,

SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.

Consistent Updates



Prior Work



Abstractions for Network Update

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

ABSTRACT

Configuration changes are a common source of instability in networks, leading to outages, performance disruptions, and security vulnerabilities. Even when the initial and final configurations are correct, the update process itself often steps through intermediate configurations

Networks exist in a constant state of flux. Operators frequently modify routing tables, adjust link weights, and change access control lists to perform tasks from planned maintenance, to traffic engineering, to patching security vulnerabilities, to migrating virtual machines in a datacenter. But even when updates are planned well in advance, they are difficult to implement correctly

Categories

General Terms

Design, Languages, Theory

Keywords

Consistency, planned change, software-defined networking, OpenFlow, network programming languages, Frenetic.

1. INTRODUCTION

"Nothing endures but change."

—Heraclitus

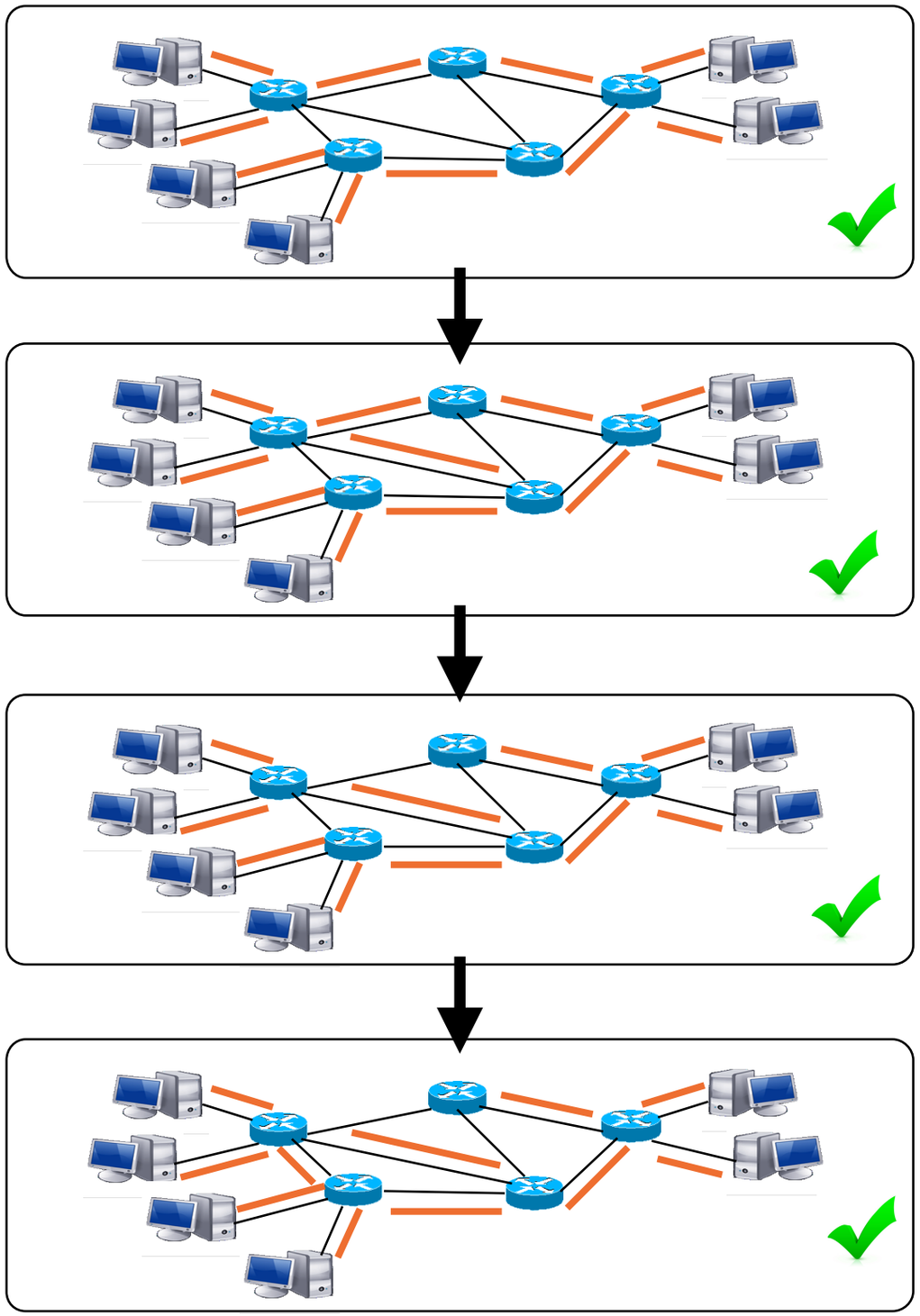
Software-defined networks. The emergence of Software Defined Networks (SDN) presents a tremendous opportunity for developing general abstractions for managing network updates. In an SDN, a program running on a logically-centralized controller manages the network directly by configuring the packet-handling mechanisms in the underlying switches. For example, the OpenFlow API allows a controller to install rules that each specify a *pattern* that matches on bits in the packet header, *actions* performed on matching packets (such as drop, forward, or divert to the controller), a *priority* (to disambiguate between overlapping patterns), and *timeouts* (to allow the switch to remove stale rules) [10]. Hence, whereas today network operators have (at best) indirect control over the distributed implementations of routing, access control, and load balancing, SDN platforms like OpenFlow provide programmers with direct control over the processing of packets in the network.

However, despite the conceptual appeal of centralized control,

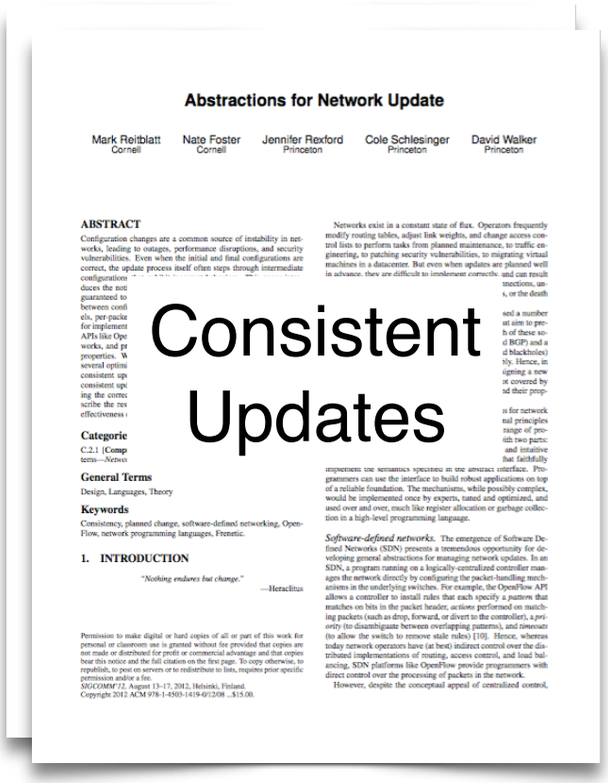
SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.

Consistent Updates

Any property <= “packet coherence"



Prior Work



Abstractions for Network Update

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

ABSTRACT

Configuration changes are a common source of instability in networks, leading to outages, performance disruptions, and security vulnerabilities. Even when the initial and final configurations are correct, the update process itself often steps through intermediate configurations

Networks exist in a constant state of flux. Operators frequently modify routing tables, adjust link weights, and change access control lists to perform tasks from planned maintenance, to traffic engineering, to patching security vulnerabilities, to migrating virtual machines in a datacenter. But even when updates are planned well in advance, they are difficult to implement correctly

General Terms

Design, Languages, Theory

Keywords

Consistency, planned change, software-defined networking, OpenFlow, network programming languages, Frenetic.

1. INTRODUCTION

"Nothing endures but change."

—Heraclitus

Software-defined networks. The emergence of Software Defined Networks (SDN) presents a tremendous opportunity for developing general abstractions for managing network updates. In an SDN, a program running on a logically-centralized controller manages the network directly by configuring the packet-handling mechanisms in the underlying switches. For example, the OpenFlow API allows a controller to install rules that each specify a *pattern* that matches on bits in the packet header, *actions* performed on matching packets (such as drop, forward, or divert to the controller), a *priority* (to disambiguate between overlapping patterns), and *timeouts* (to allow the switch to remove stale rules) [10]. Hence, whereas today network operators have (at best) indirect control over the distributed implementations of routing, access control, and load balancing, SDN platforms like OpenFlow provide programmers with direct control over the processing of packets in the network.

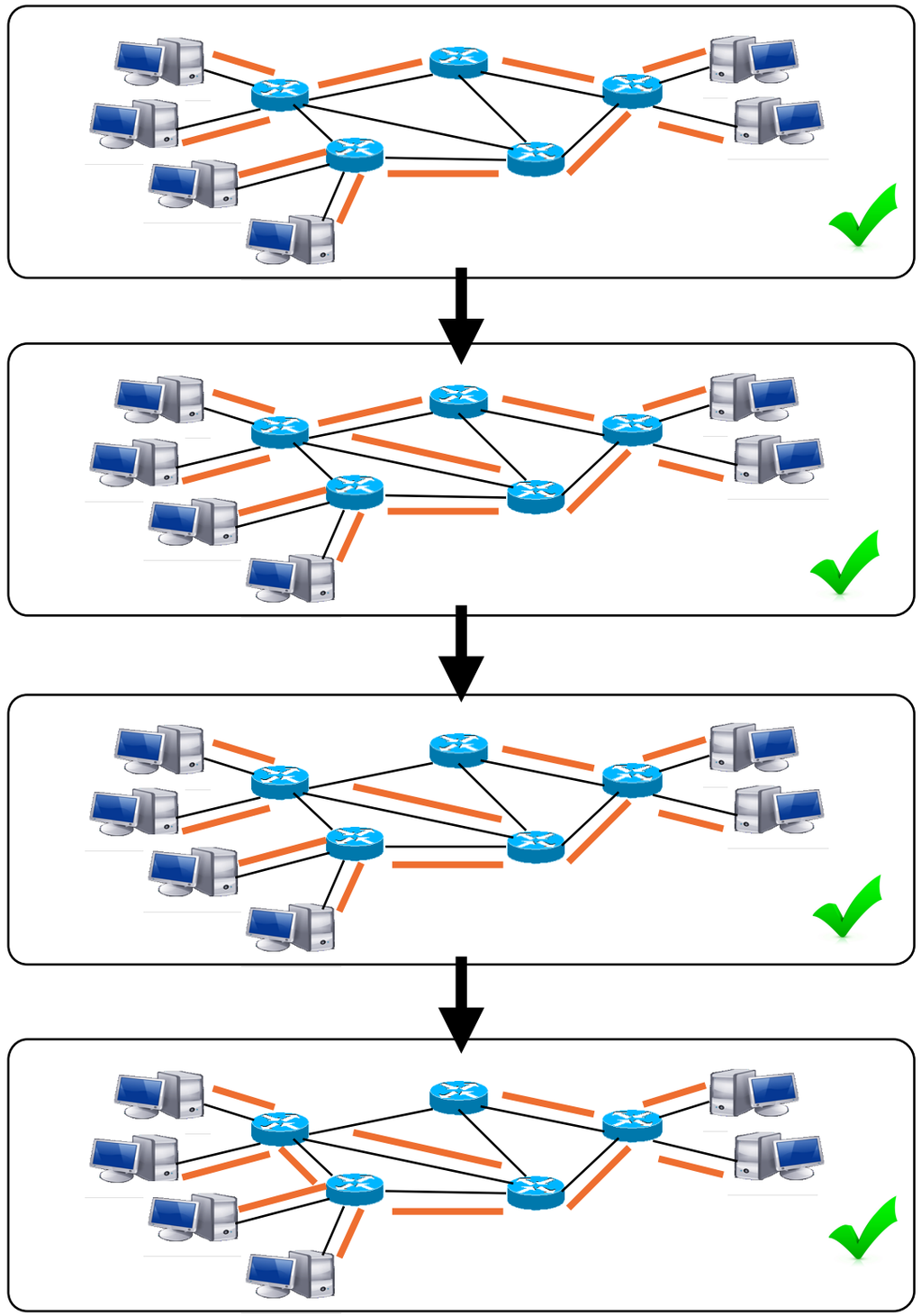
However, despite the conceptual appeal of centralized control,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.
Copyright 2012 ACM 978-1-4503-1419-0/12/08 ..$15.00.

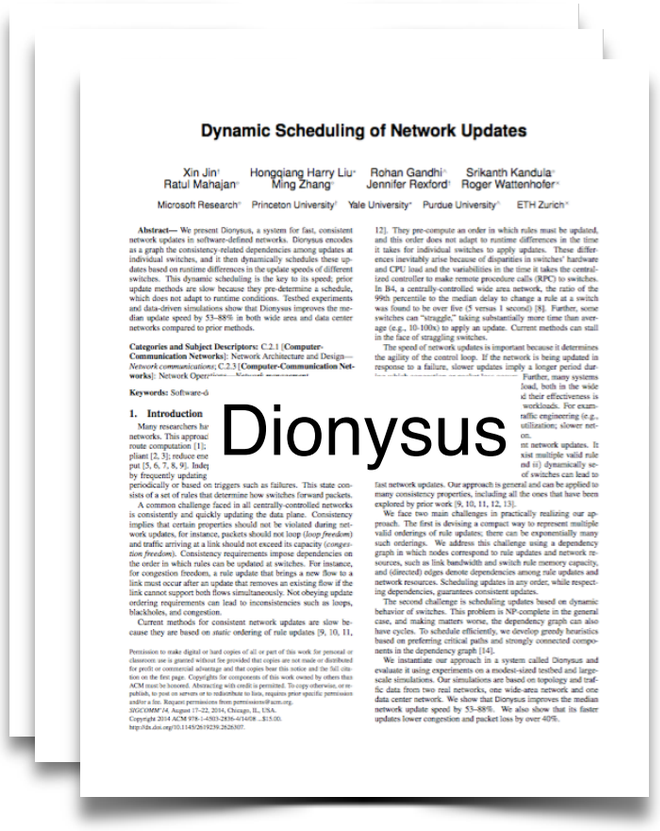
Consistent Updates

Any property <= “packet coherence"

Sacrifices efficiency



Prior Work



Dynamic Scheduling of Network Updates

Xin Jin[†] Hongqiang Harry Liu[◦] Rohan Gandhi[◦] Srikanth Kandula[◦]
Ratul Mahajan[◦] Ming Zhang[◦] Jennifer Rexford[†] Roger Wattenhofer[•]

Microsoft Research[◦] Princeton University[†] Yale University[•] Purdue University[•] ETH Zurich[•]

Abstract— We present Dionysus, a system for fast, consistent network updates in software-defined networks. Dionysus encodes as a graph the consistency-related dependencies among updates at individual switches, and it then dynamically schedules these updates based on runtime differences in the update speeds of different switches. This dynamic scheduling is the key to its speed; prior update methods are slow because they pre-determine a schedule, which does not adapt to runtime conditions. Testbed experiments and data-driven simulations show that Dionysus improves the median update speed by 53–88% in both wide area and data center networks compared to prior methods.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications*; C.2.3 [**Computer-Communication Networks**]: Network Oper

Keywords: Software-d

1. Introduction

Many researchers ha
networks. This approac
route computation [1];
plant [2, 3]; reduce ene
put [5, 6, 7, 8, 9]. Indee
by frequently updating
periodically or based on triggers such as failures. This state consists of a set of rules that determine how switches forward packets.

A common challenge faced in all centrally-controlled networks is consistently and quickly updating the data plane. Consistency implies that certain properties should not be violated during network updates, for instance, packets should not loop (*loop freedom*) and traffic arriving at a link should not exceed its capacity (*congestion freedom*). Consistency requirements impose dependencies on the order in which rules can be updated at switches. For instance, for congestion freedom, a rule update that brings a new flow to a link must occur after an update that removes an existing flow if the link cannot support both flows simultaneously. Not obeying update ordering requirements can lead to inconsistencies such as loops, blackholes, and congestion.

Current methods for consistent network updates are slow because they are based on *static* ordering of rule updates [9, 10, 11, 12]. They pre-compute an order in which rules must be updated, and this order does not adapt to runtime differences in the time it takes for individual switches to apply updates. These differences inevitably arise because of disparities in switches' hardware and CPU load and the variabilities in the time it takes the centralized controller to make remote procedure calls (RPC) to switches. In B4, a centrally-controlled wide area network, the ratio of the 99th percentile to the median delay to change a rule at a switch was found to be over five (5 versus 1 second) [8]. Further, some switches can "straggle," taking substantially more time than average (e.g., 10-100x) to apply an update. Current methods can stall in the face of straggling switches.

The speed of network updates is important because it determines the agility of the control loop. If the network is being updated in response to a failure, slower updates imply a longer period dur-

Further, many systems
load, both in the wide
d their effectiveness is
workloads. For exam-
raffic engineering (e.g.,
utilization; slower net-
on.
nt network updates. It
xist multiple valid rule
nd ii) dynamically se-
of switches can lead to

Dionysus

fast network updates. Our approach is general and can be applied to many consistency properties, including all the ones that have been explored by prior work [9, 10, 11, 12, 13].

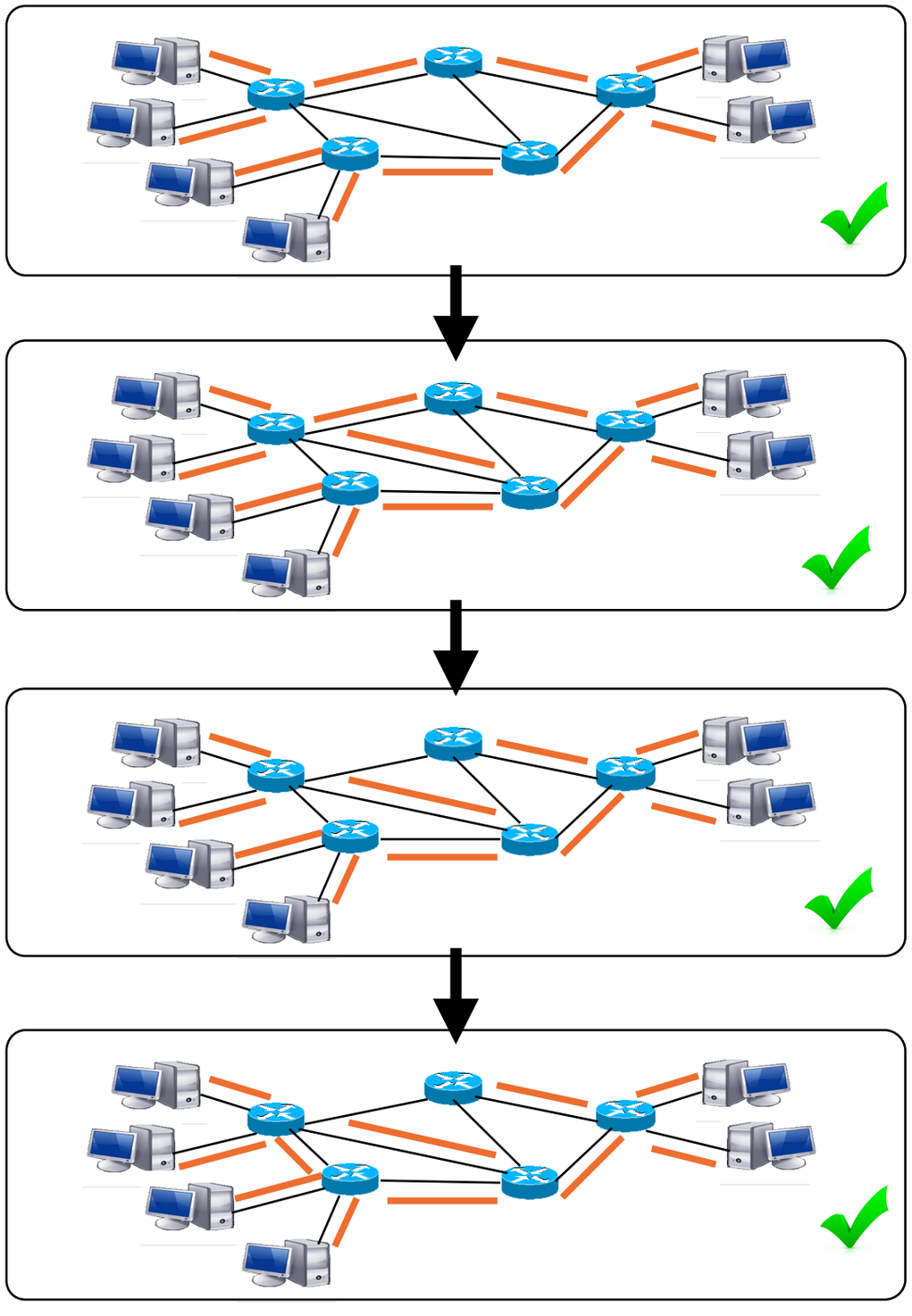
We face two main challenges in practically realizing our approach. The first is devising a compact way to represent multiple valid orderings of rule updates; there can be exponentially many such orderings. We address this challenge using a dependency graph in which nodes correspond to rule updates and network resources, such as link bandwidth and switch rule memory capacity, and (directed) edges denote dependencies among rule updates and network resources. Scheduling updates in any order, while respecting dependencies, guarantees consistent updates.

The second challenge is scheduling updates based on dynamic behavior of switches. This problem is NP-complete in the general case, and making matters worse, the dependency graph can also have cycles. To schedule efficiently, we develop greedy heuristics based on preferring critical paths and strongly connected components in the dependency graph [14].

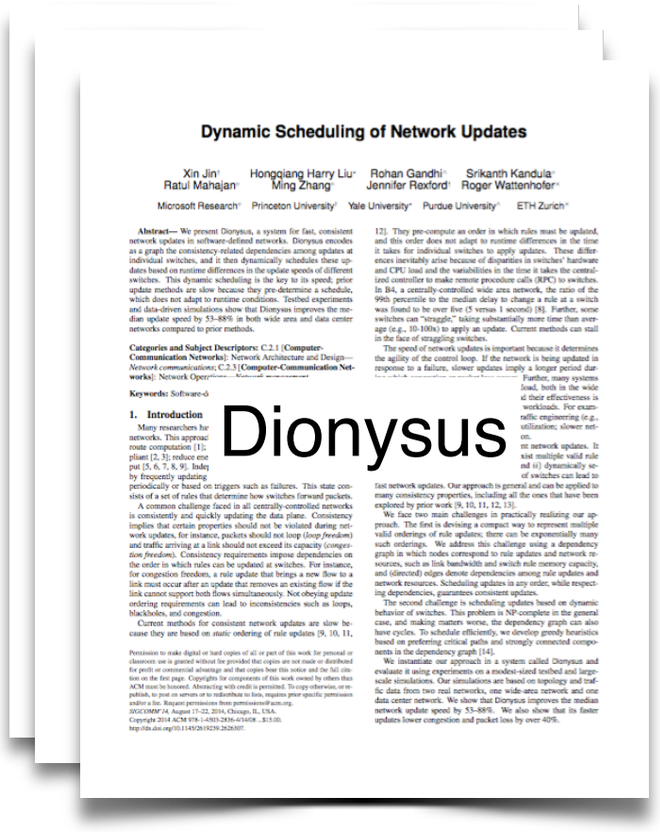
We instantiate our approach in a system called *Dionysus* and evaluate it using experiments on a modest-sized testbed and large-scale simulations. Our simulations are based on topology and traffic data from two real networks, one wide-area network and one data center network. We show that Dionysus improves the median network update speed by 53–88%. We also show that its faster updates lower congestion and packet loss by over 40%.

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2619239.2626307.



Prior Work



Dynamic Scheduling of Network Updates

Xin Jin[†] Hongqiang Harry Liu[◦] Rohan Gandhi[◦] Srikanth Kandula[◦]
Ratul Mahajan[◦] Ming Zhang[◦] Jennifer Rexford[†] Roger Wattenhofer[•]

Microsoft Research[◦] Princeton University[†] Yale University[•] Purdue University[•] ETH Zurich[•]

Abstract— We present Dionysus, a system for fast, consistent network updates in software-defined networks. Dionysus encodes as a graph the consistency-related dependencies among updates at individual switches, and it then dynamically schedules these updates based on runtime differences in the update speeds of different switches. This dynamic scheduling is the key to its speed; prior update methods are slow because they pre-determine a schedule, which does not adapt to runtime conditions. Testbed experiments and data-driven simulations show that Dionysus improves the median update speed by 53–88% in both wide area and data center networks compared to prior methods.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications*; C.2.3 [**Computer-Communication Networks**]: Network Oper

Keywords: Software-d

1. Introduction

Many researchers ha
networks. This approac
route computation [1];
plant [2, 3]; reduce ene
put [5, 6, 7, 8, 9]. Inde
by frequently updating
periodically or based on triggers such as failures. This state consists of a set of rules that determine how switches forward packets.

A common challenge faced in all centrally-controlled networks is consistently and quickly updating the data plane. Consistency implies that certain properties should not be violated during network updates, for instance, packets should not loop (*loop freedom*) and traffic arriving at a link should not exceed its capacity (*congestion freedom*). Consistency requirements impose dependencies on the order in which rules can be updated at switches. For instance, for congestion freedom, a rule update that brings a new flow to a link must occur after an update that removes an existing flow if the link cannot support both flows simultaneously. Not obeying update ordering requirements can lead to inconsistencies such as loops, blackholes, and congestion.

Current methods for consistent network updates are slow because they are based on *static* ordering of rule updates [9, 10, 11, 12]. They pre-compute an order in which rules must be updated, and this order does not adapt to runtime differences in the time it takes for individual switches to apply updates. These differences inevitably arise because of disparities in switches' hardware and CPU load and the variabilities in the time it takes the centralized controller to make remote procedure calls (RPC) to switches. In B4, a centrally-controlled wide area network, the ratio of the 99th percentile to the median delay to change a rule at a switch was found to be over five (5 versus 1 second) [8]. Further, some switches can "straggle," taking substantially more time than average (e.g., 10-100x) to apply an update. Current methods can stall in the face of straggling switches.

The speed of network updates is important because it determines the agility of the control loop. If the network is being updated in response to a failure, slower updates imply a longer period dur-

Further, many systems
load, both in the wide
d their effectiveness is
workloads. For exam-
raffic engineering (e.g.,
utilization; slower net-
on.
nt network updates. It
xist multiple valid rule
nd ii) dynamically se-
of switches can lead to
fast network updates. Our approach is general and can be applied to many consistency properties, including all the ones that have been explored by prior work [9, 10, 11, 12, 13].

We face two main challenges in practically realizing our approach. The first is devising a compact way to represent multiple valid orderings of rule updates; there can be exponentially many such orderings. We address this challenge using a dependency graph in which nodes correspond to rule updates and network resources, such as link bandwidth and switch rule memory capacity, and (directed) edges denote dependencies among rule updates and network resources. Scheduling updates in any order, while respecting dependencies, guarantees consistent updates.

The second challenge is scheduling updates based on dynamic behavior of switches. This problem is NP-complete in the general case, and making matters worse, the dependency graph can also have cycles. To schedule efficiently, we develop greedy heuristics based on preferring critical paths and strongly connected components in the dependency graph [14].

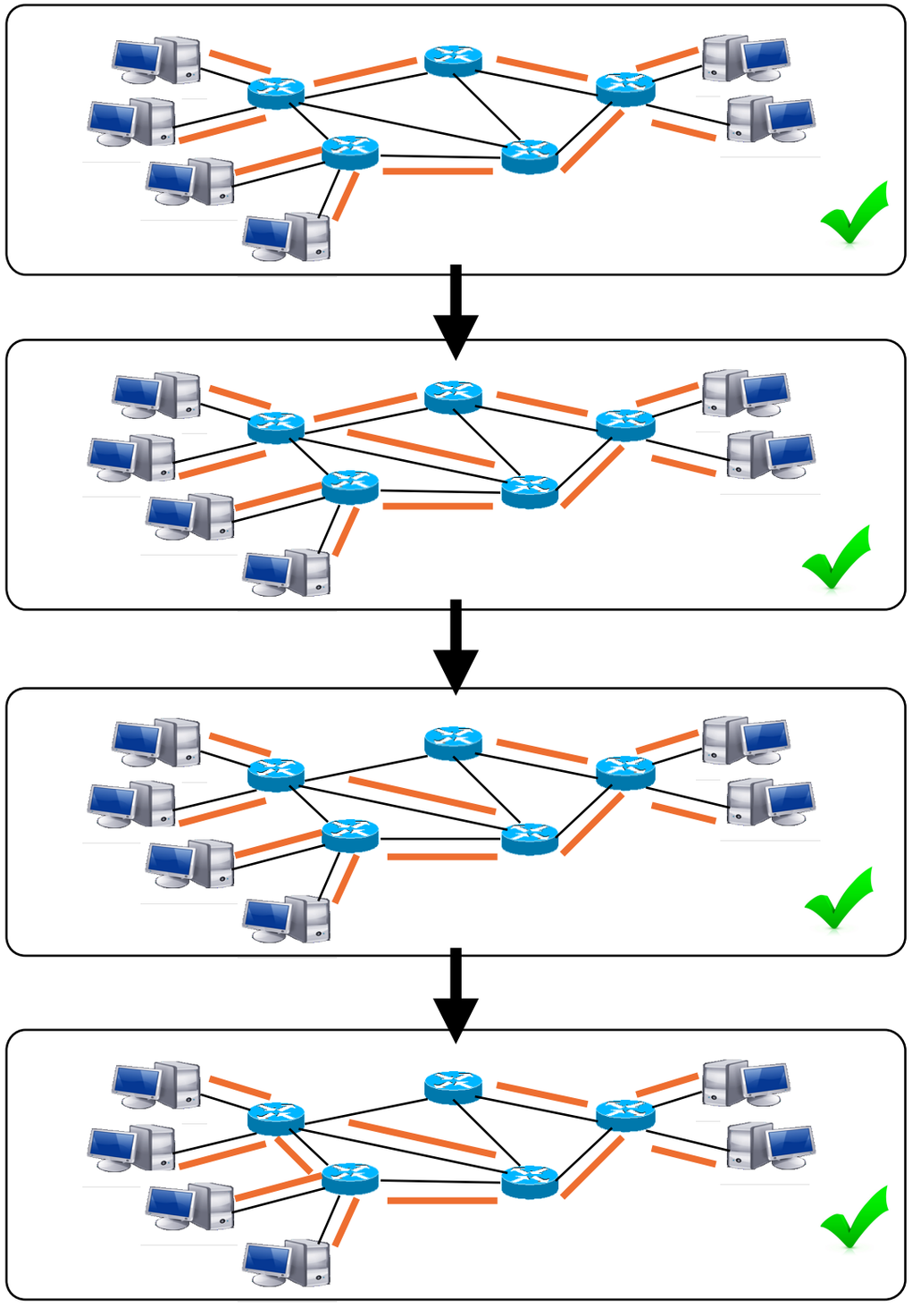
We instantiate our approach in a system called *Dionysus* and evaluate it using experiments on a modest-sized testbed and large-scale simulations. Our simulations are based on topology and traffic data from two real networks, one wide-area network and one data center network. We show that Dionysus improves the median network update speed by 53–88%. We also show that its faster updates lower congestion and packet loss by over 40%.

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

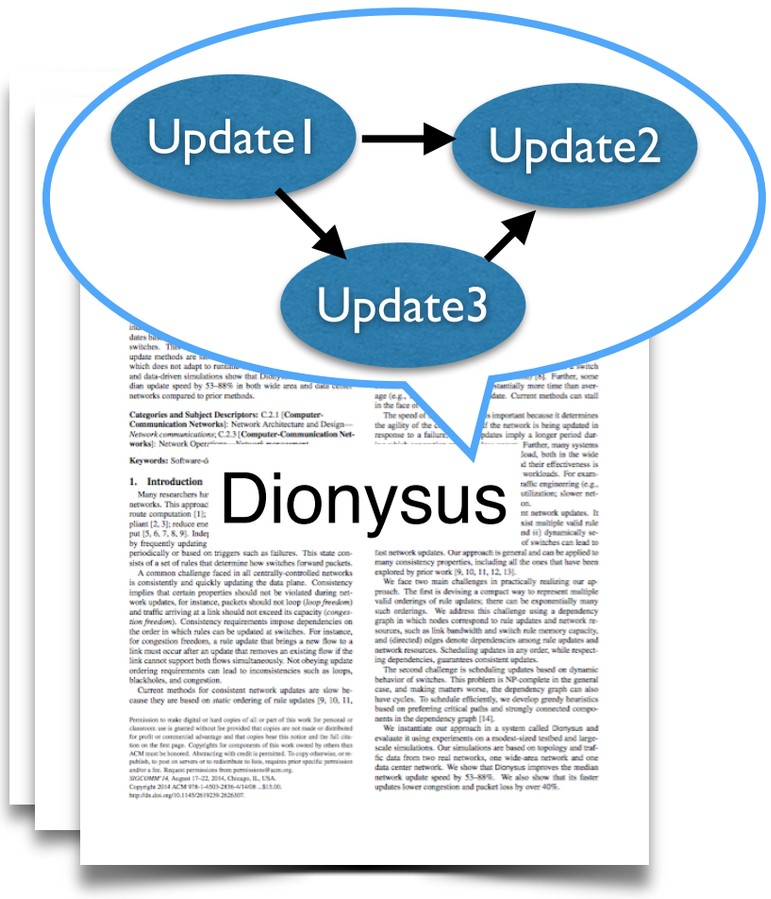
http://dx.doi.org/10.1145/2619239.2626307.

Dionysus

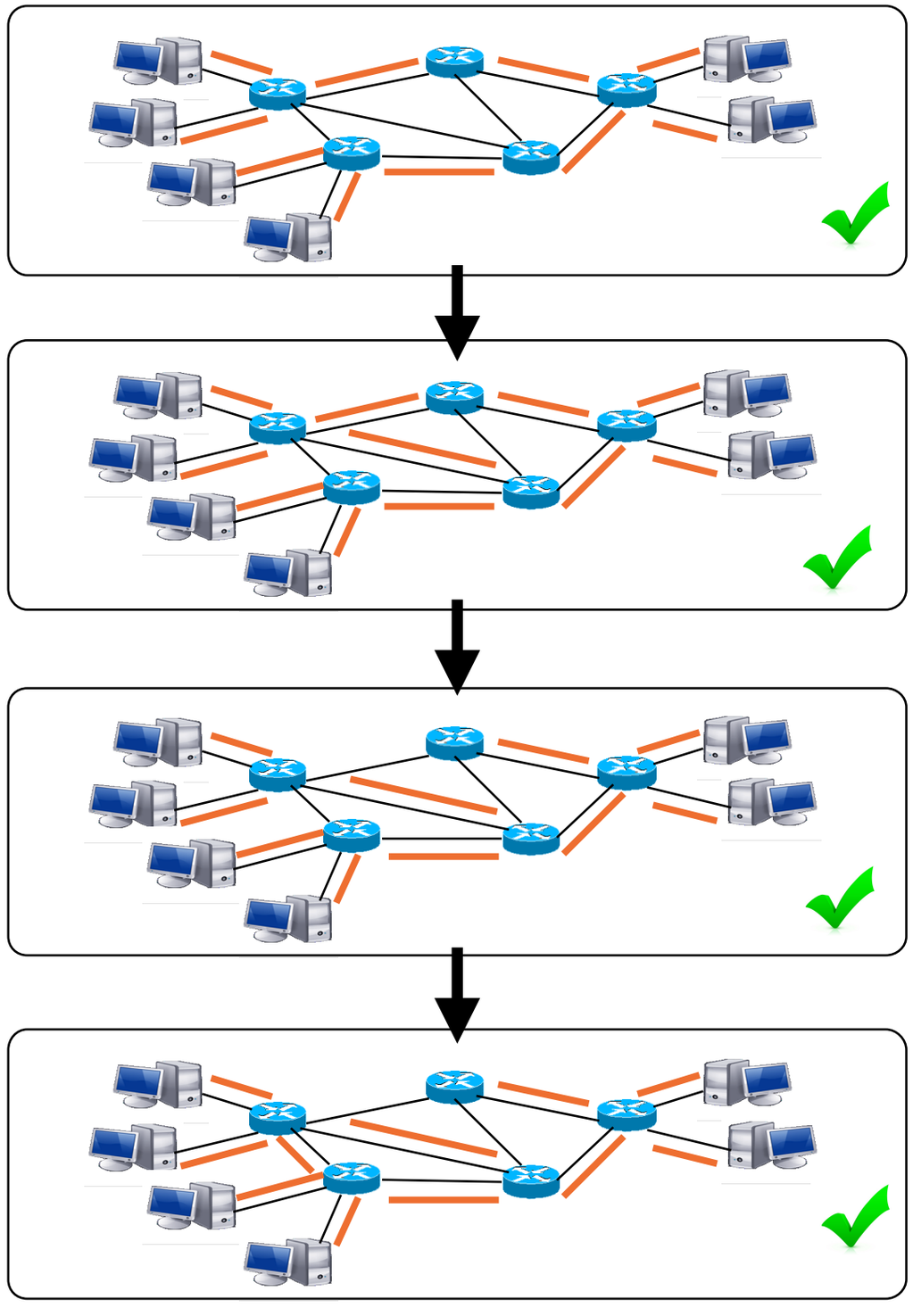
Dynamic scheduling



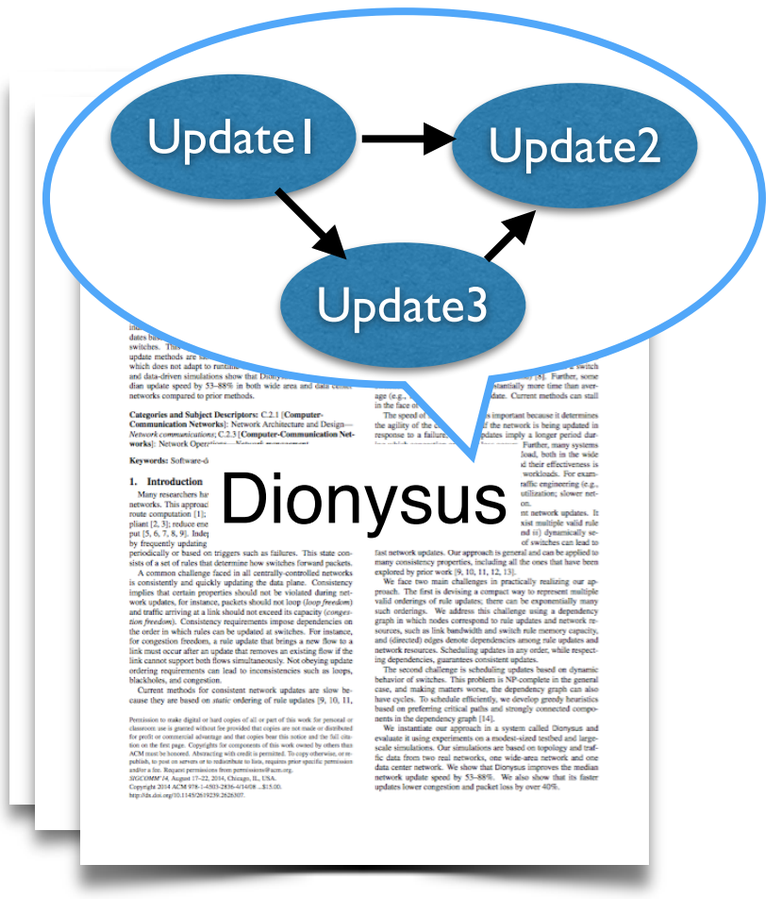
Prior Work



Dynamic scheduling

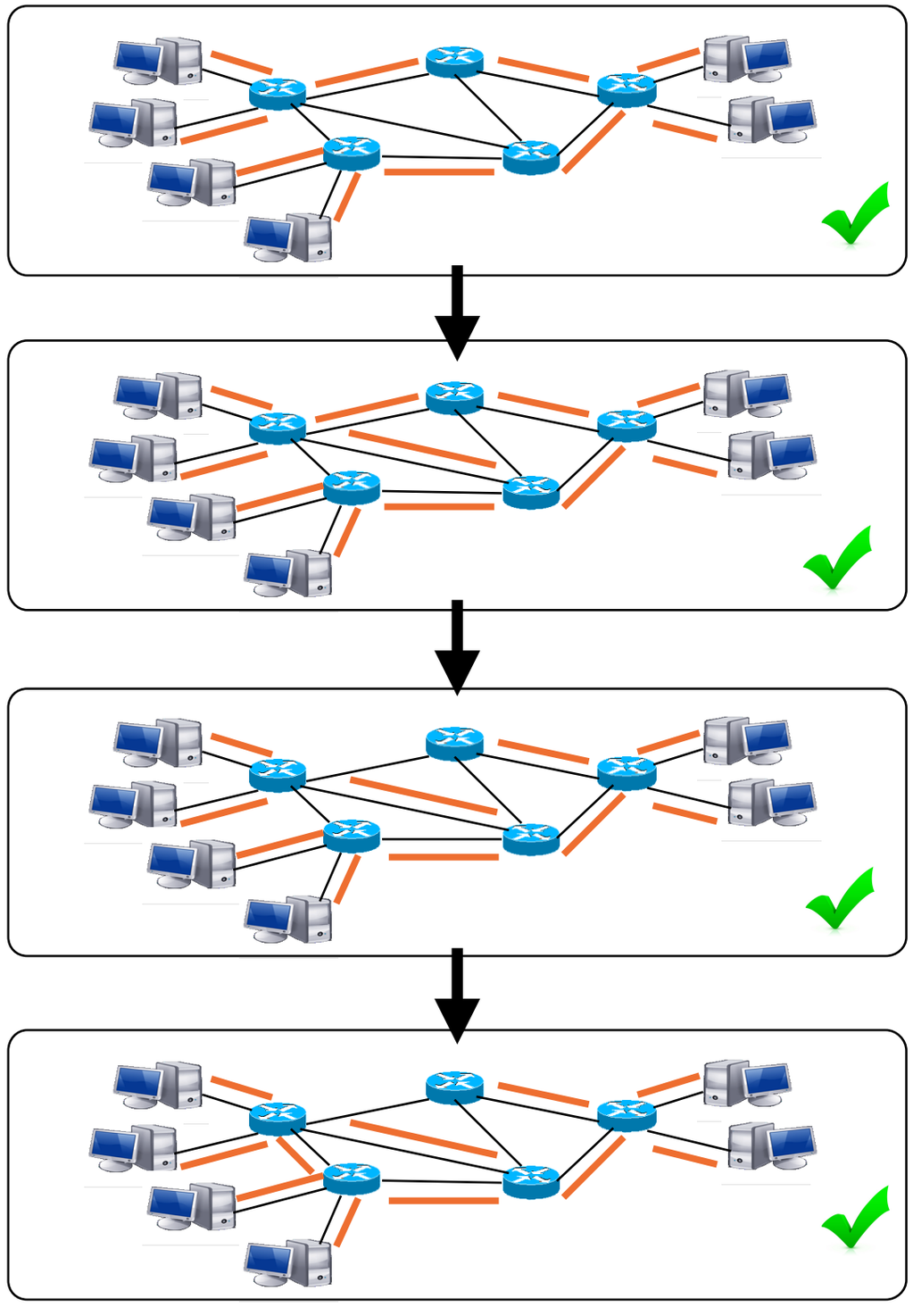


Prior Work

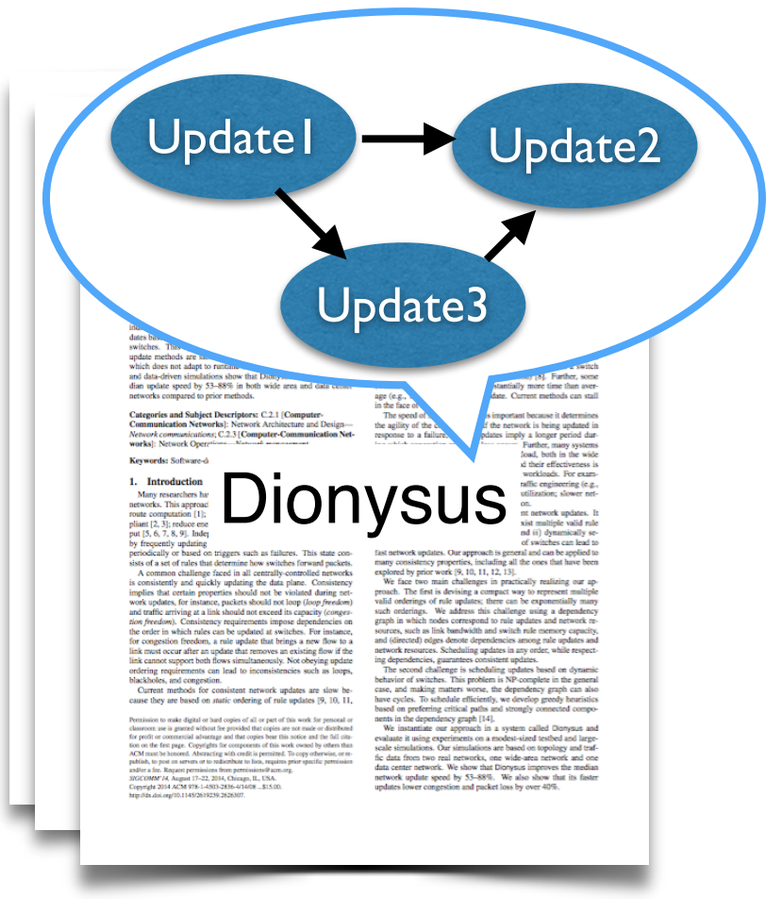


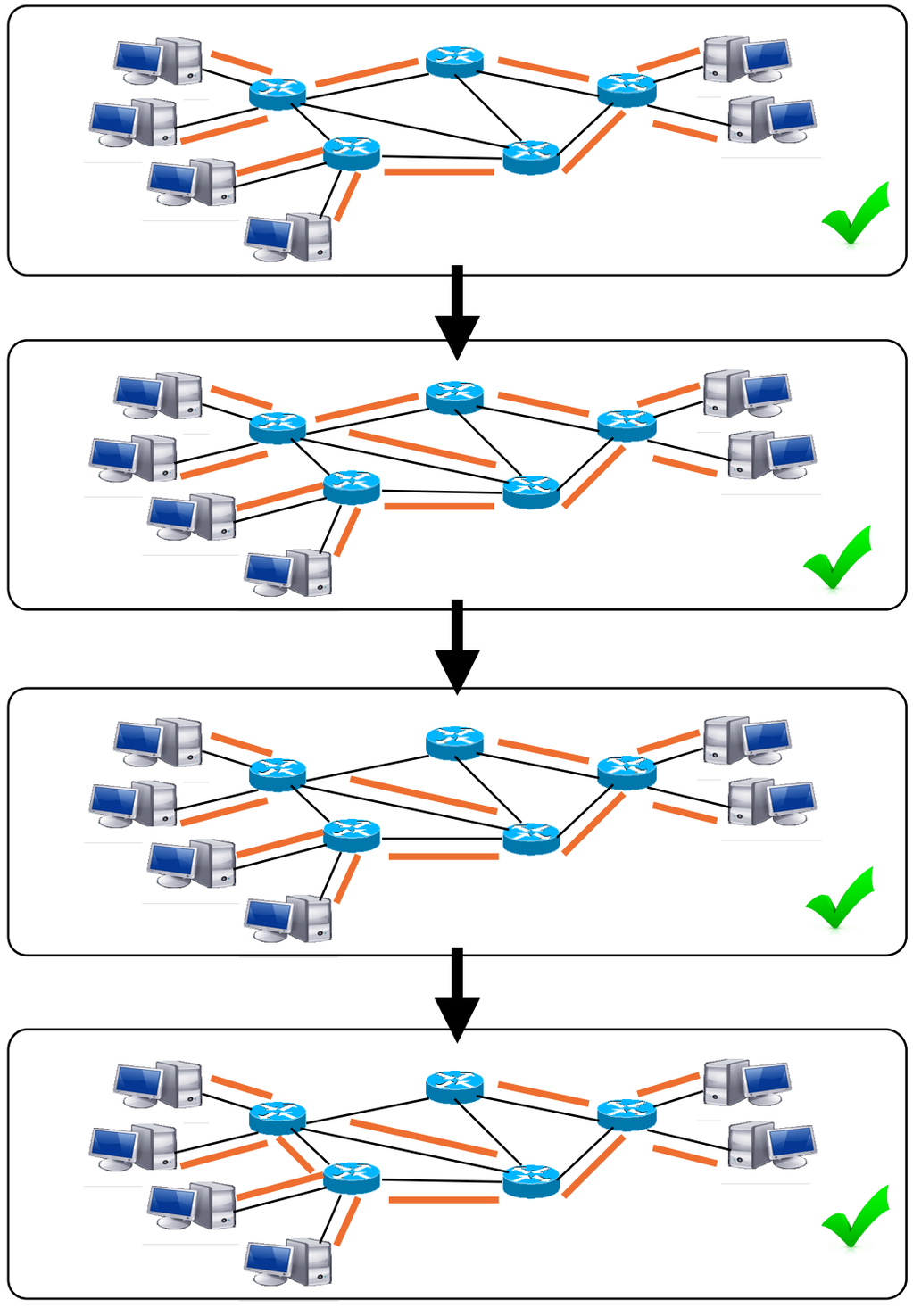
Dynamic scheduling

Customizes consistency for efficiency



Prior Work



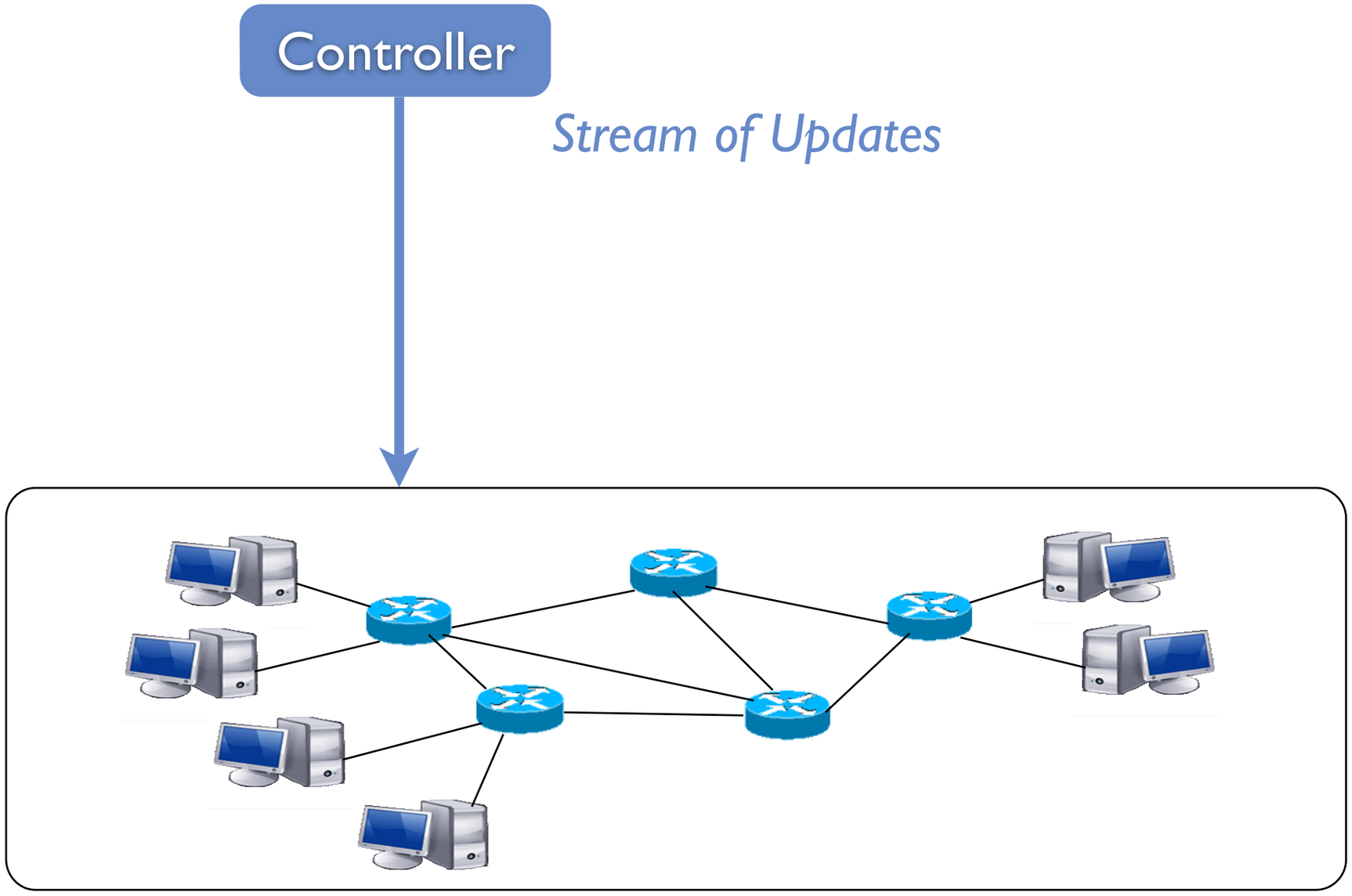


Dynamic scheduling

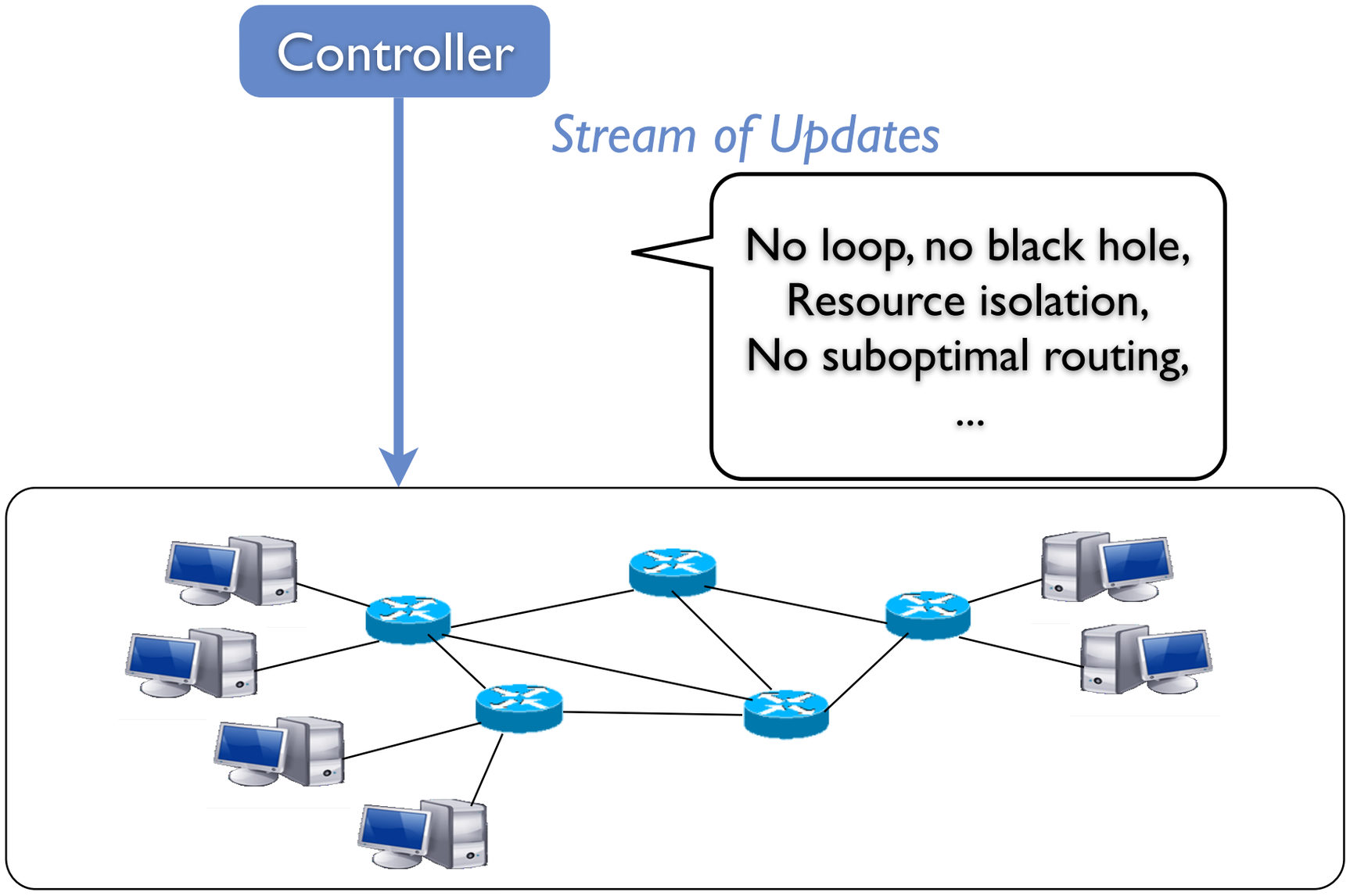
Customizes consistency for efficiency

Specific algorithms for different properties

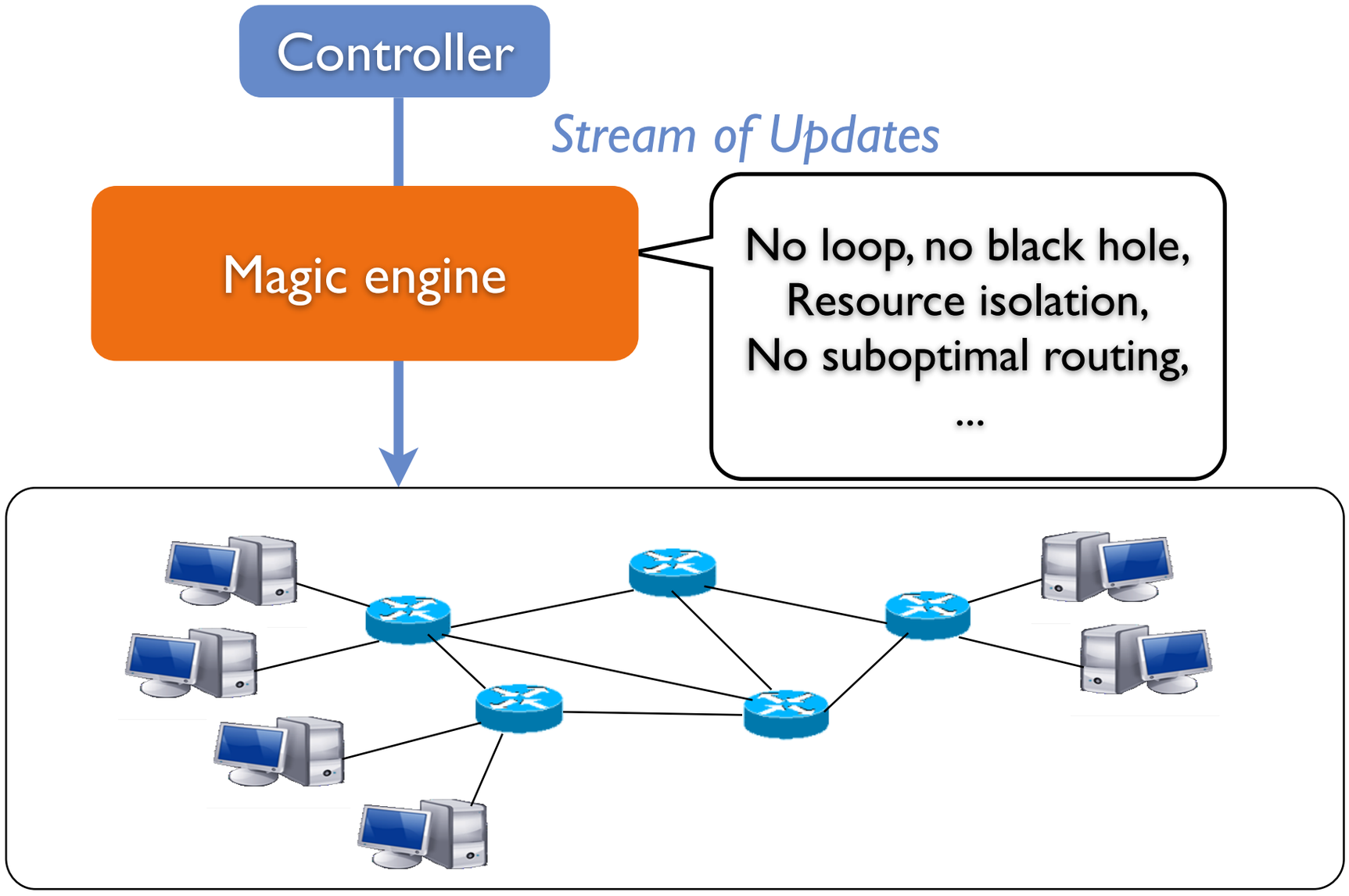
Ideally given an **arbitrary** set of properties, a sequence with **minimized** update overhead is produced



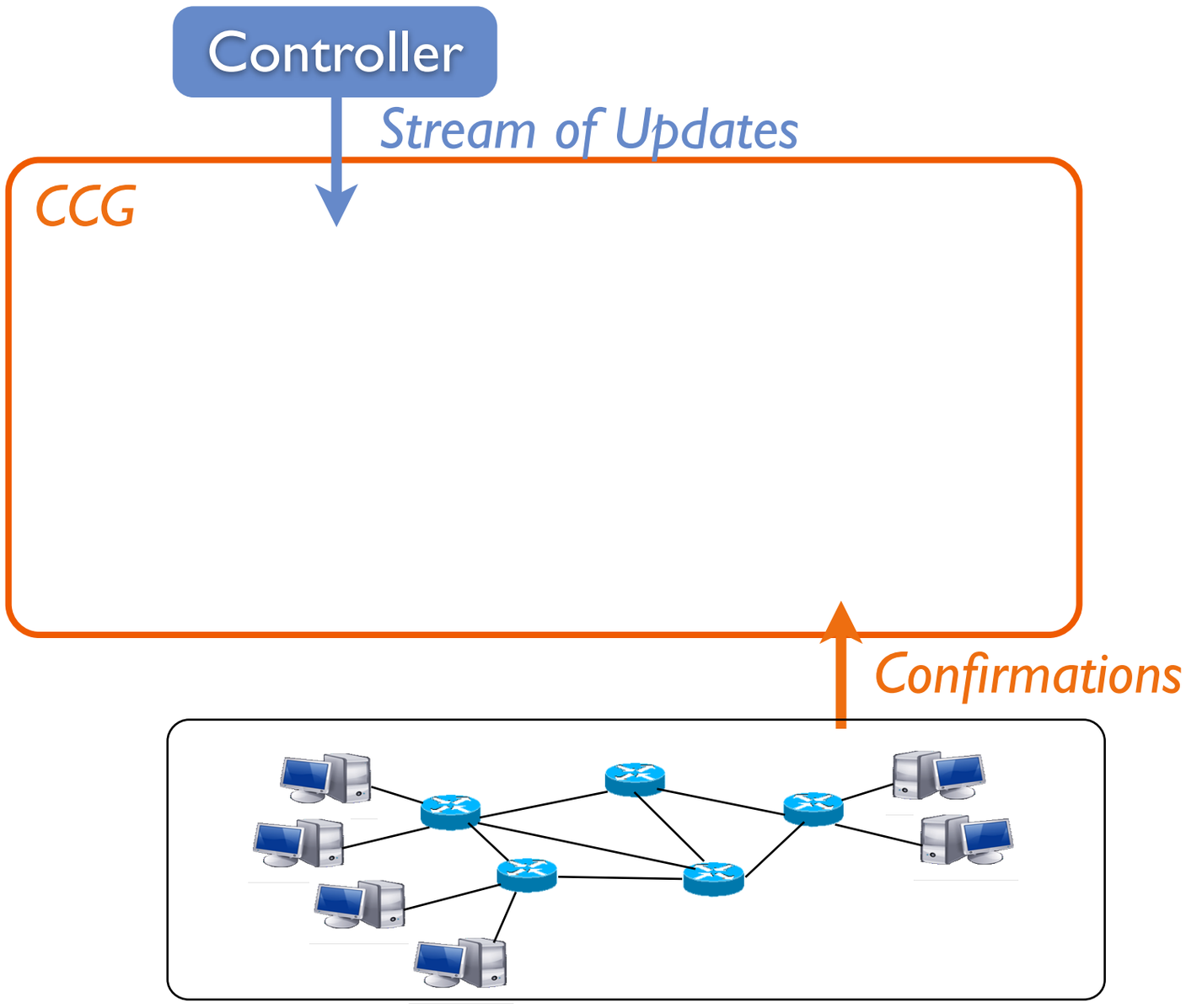
Ideally given an **arbitrary** set of properties, a sequence with **minimized** update overhead is produced



Ideally given an **arbitrary** set of properties, a sequence with **minimized** update overhead is produced

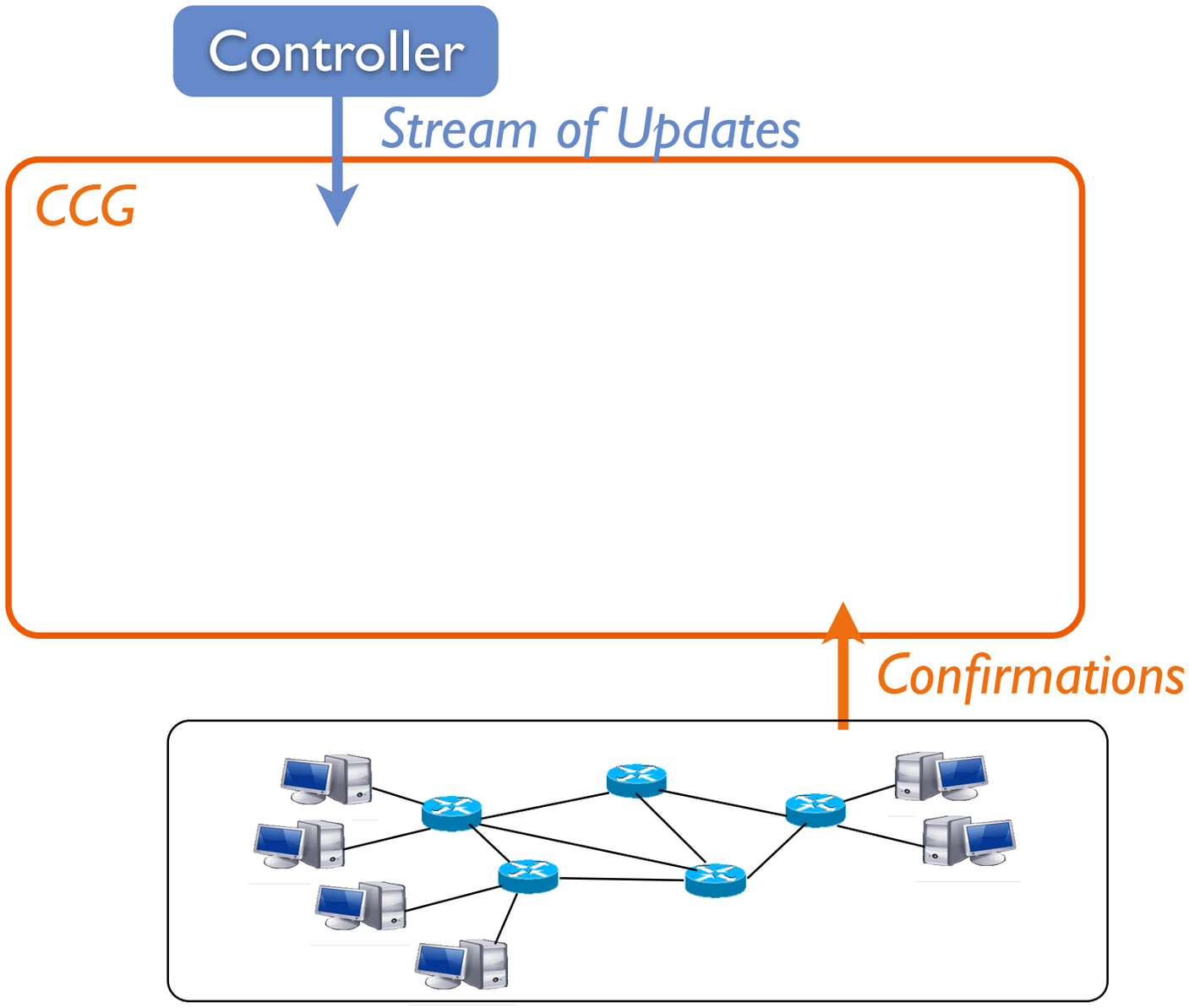


Our design: Customizable Consistency Generator



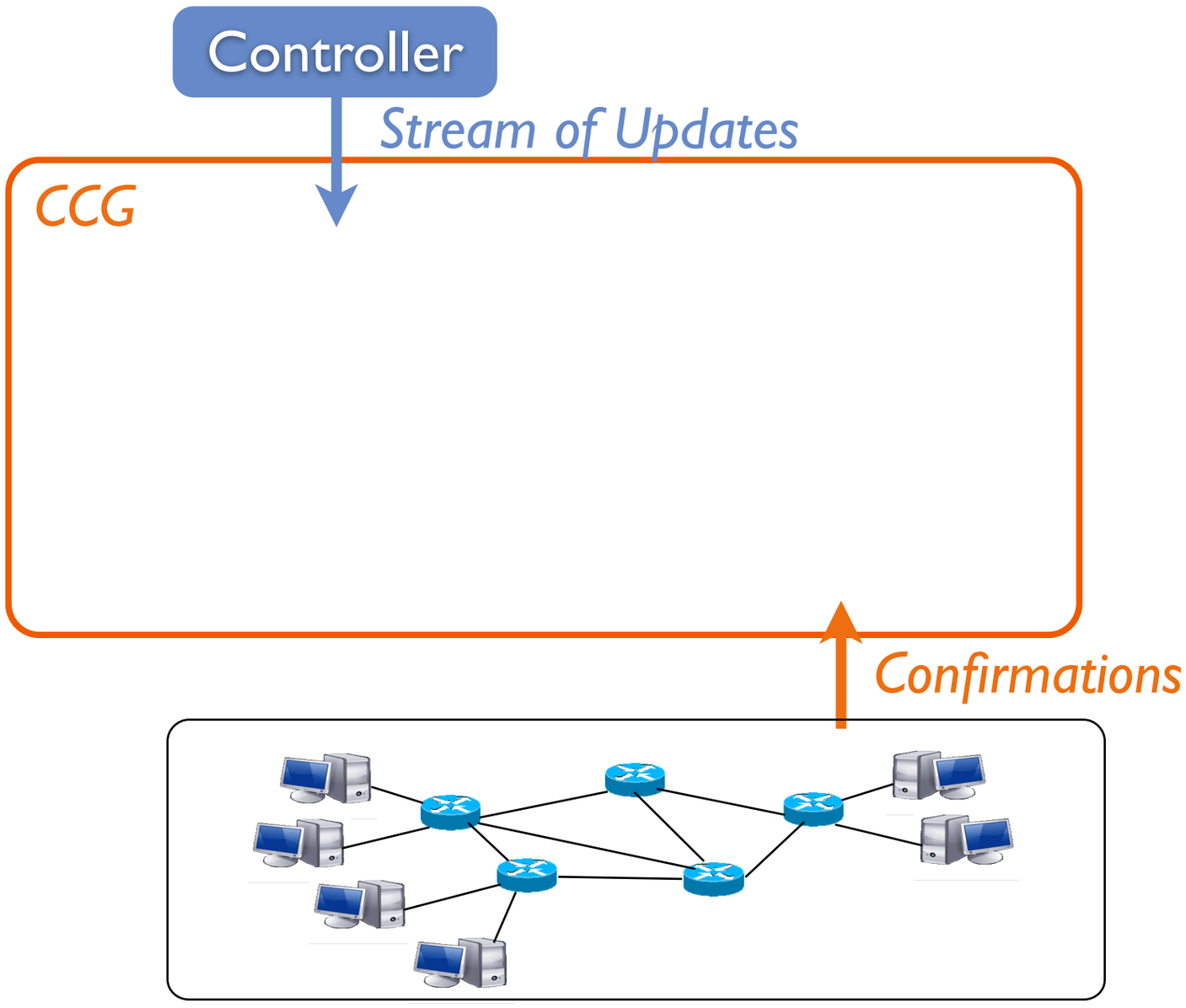
Our design: Customizable Consistency Generator

Key insight:



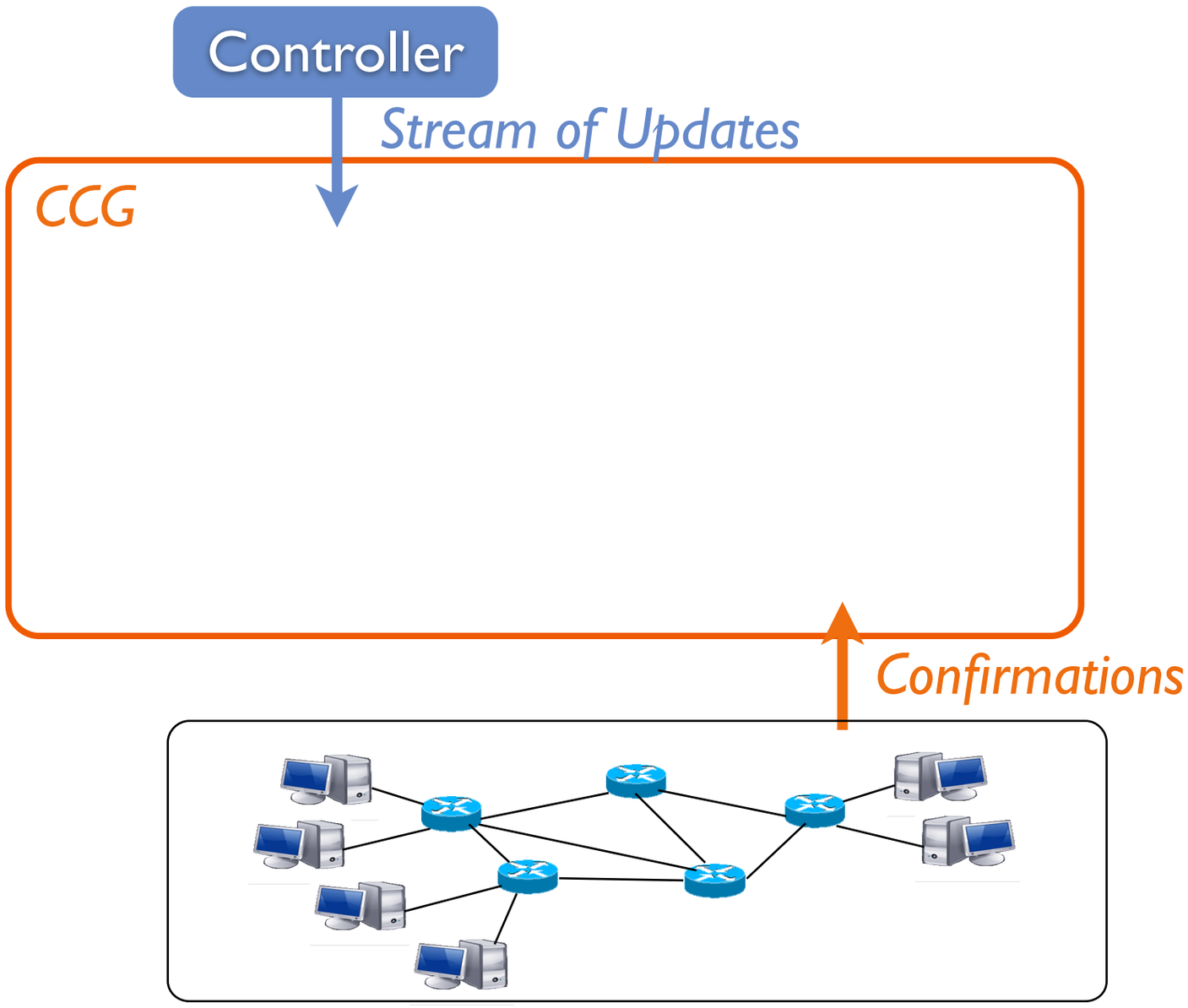
Our design: Customizable Consistency Generator

Key insight: **Synthesis**



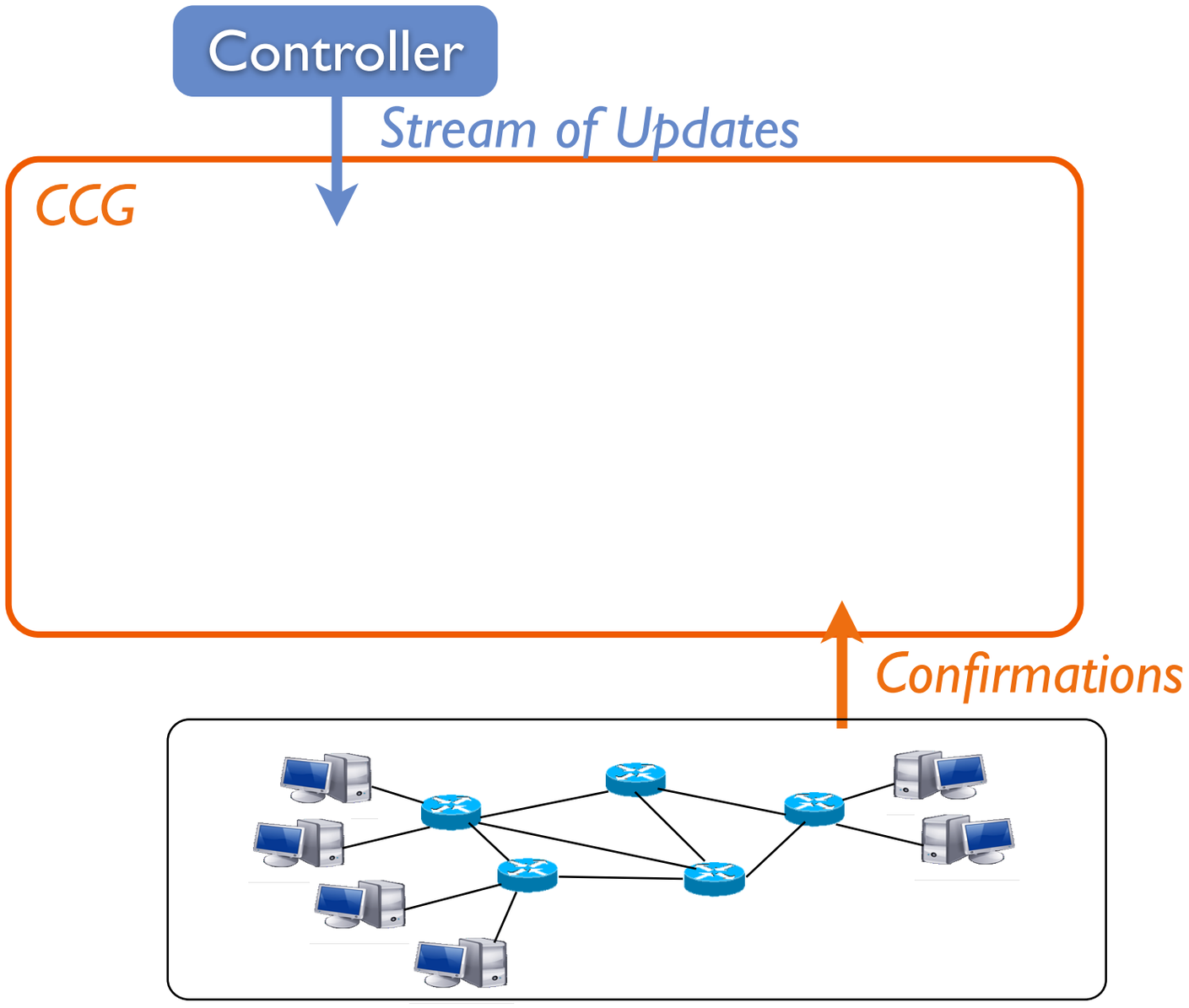
Our design: Customizable Consistency Generator

Key insight: **Synthesis** **Verification**



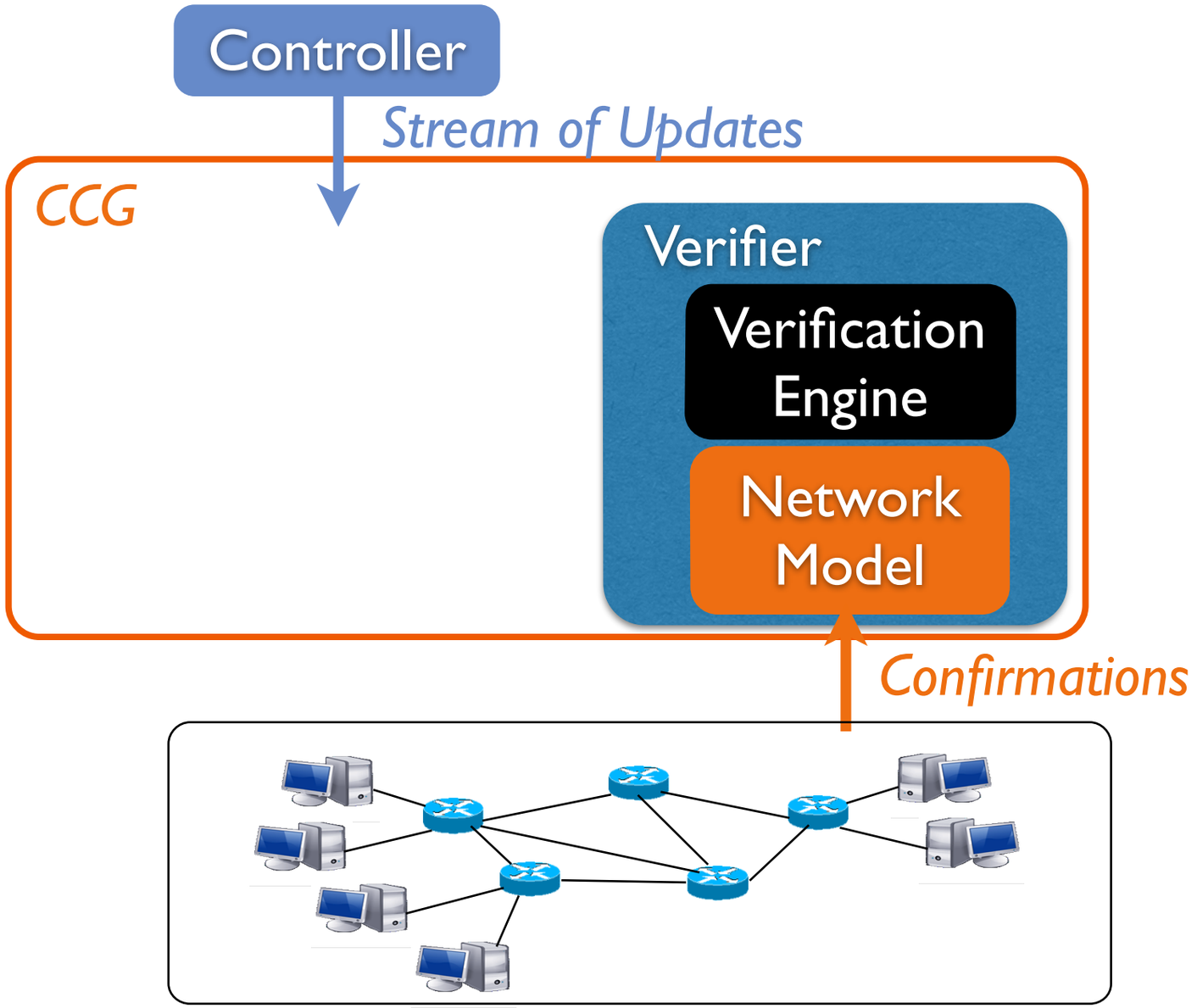
Our design: Customizable Consistency Generator

Key insight: Synthesis → Verification



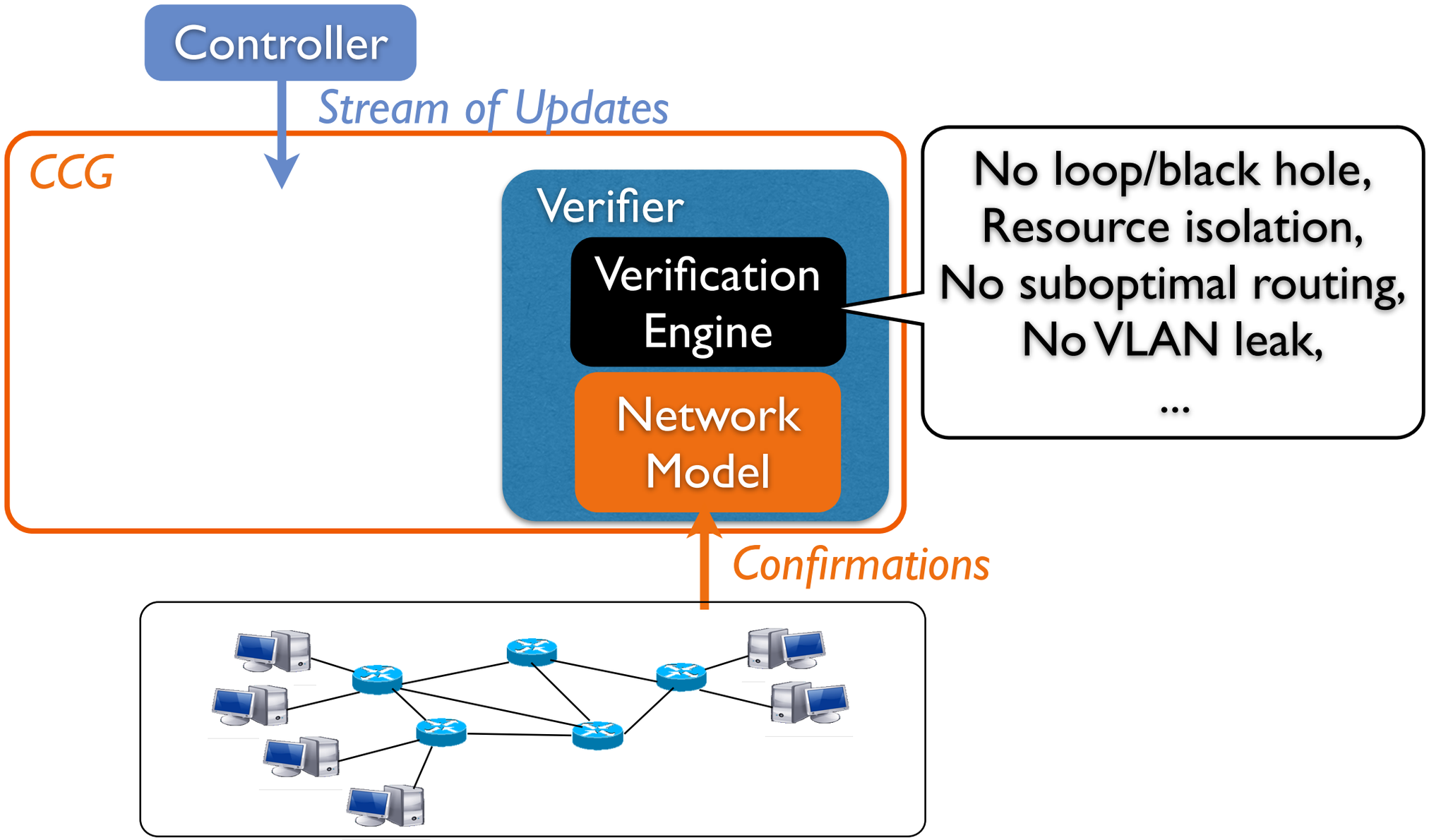
Our design: Customizable Consistency Generator

Key insight: Synthesis → Verification



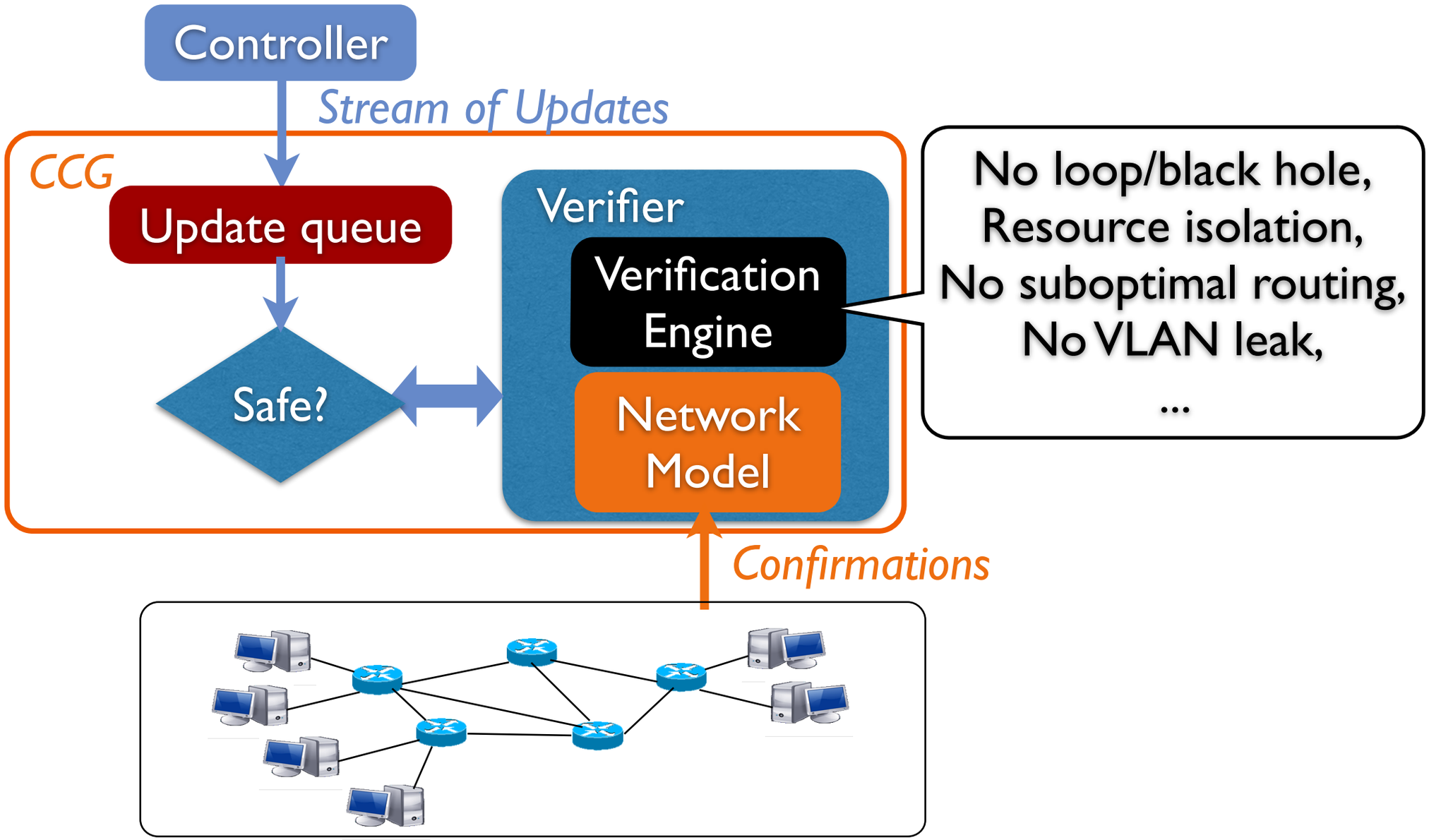
Our design: Customizable Consistency Generator

Key insight: Synthesis → Verification



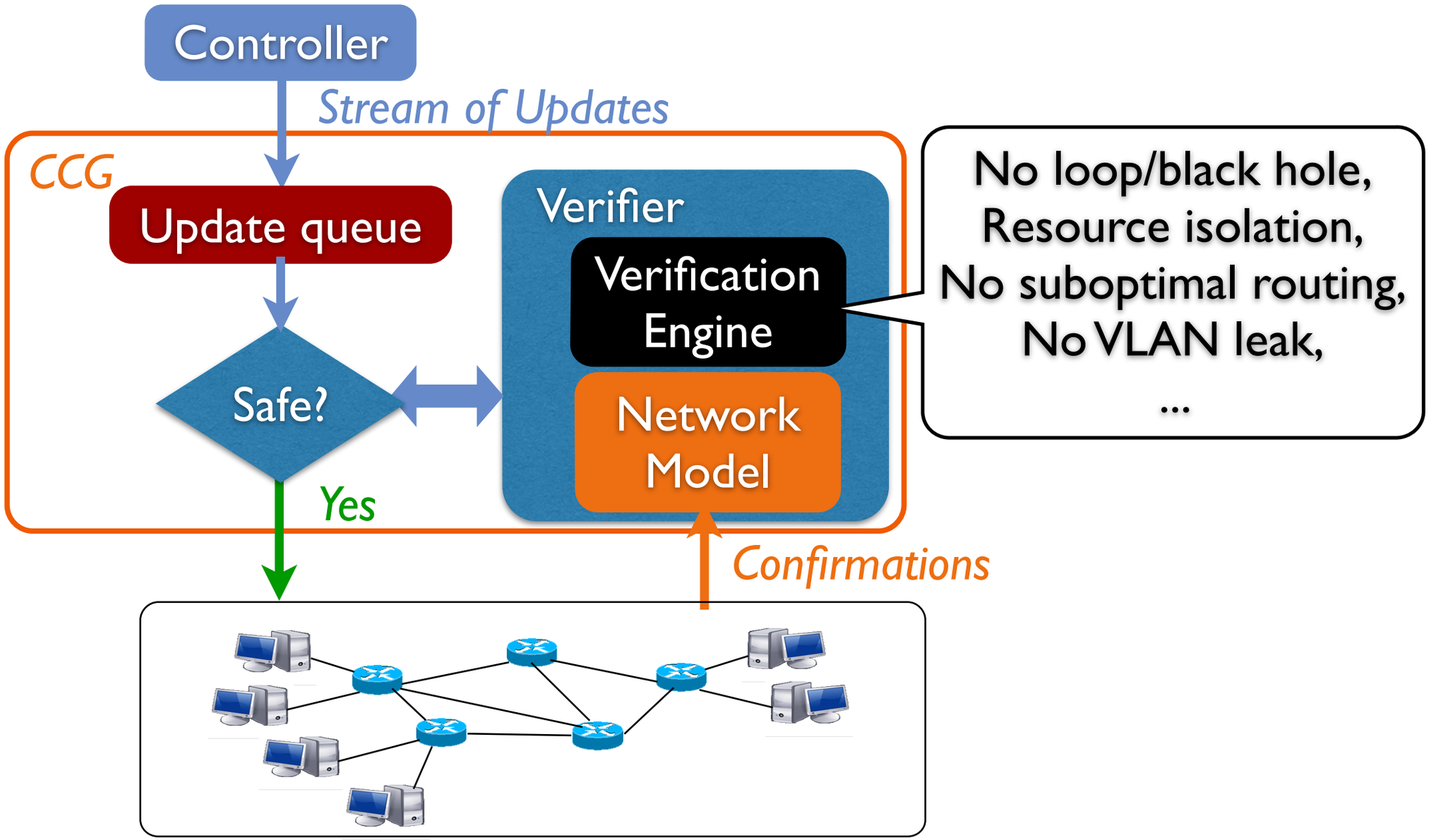
Our design: Customizable Consistency Generator

Key insight: Synthesis → Verification



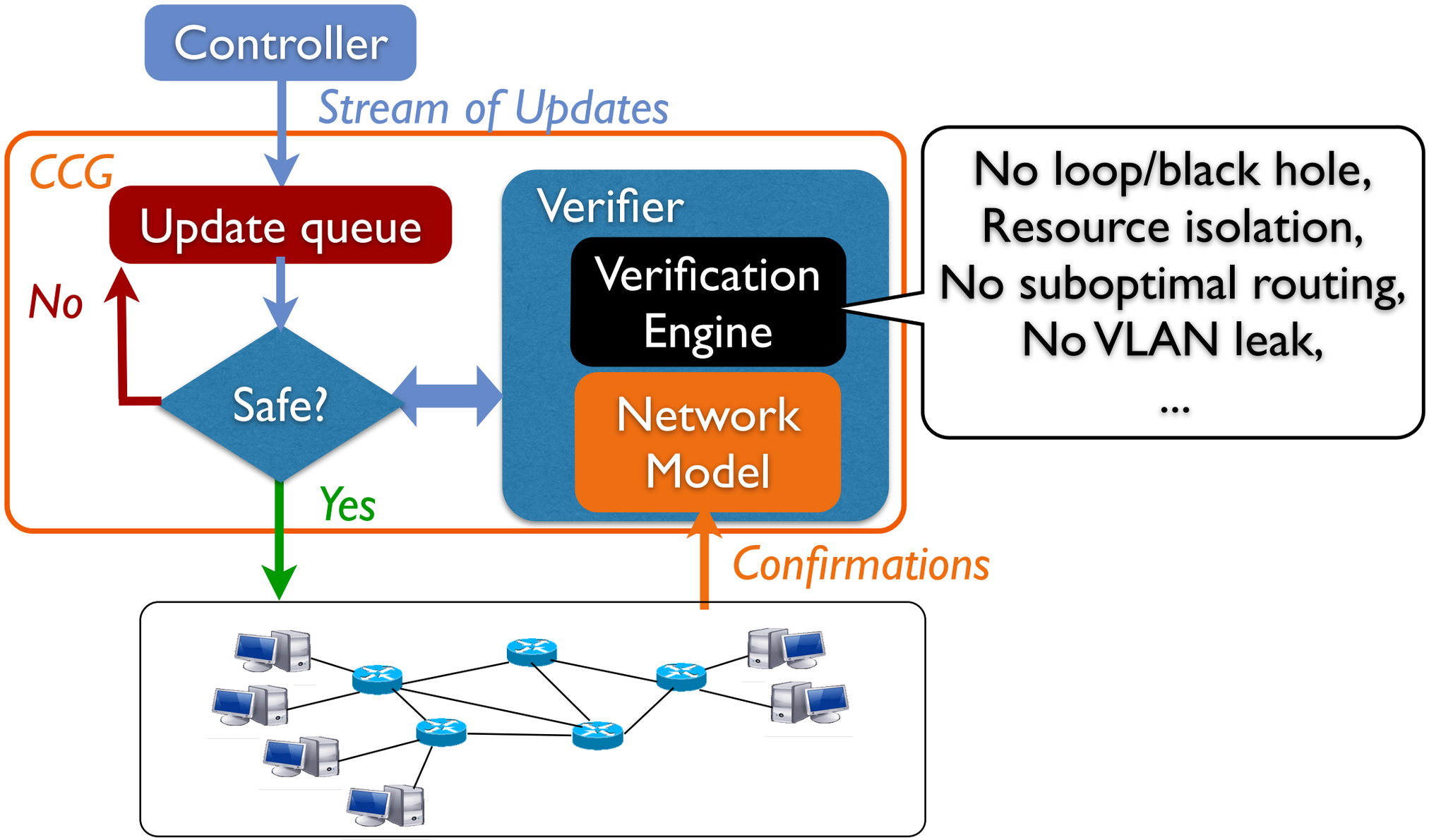
Our design: Customizable Consistency Generator

Key insight: Synthesis → Verification



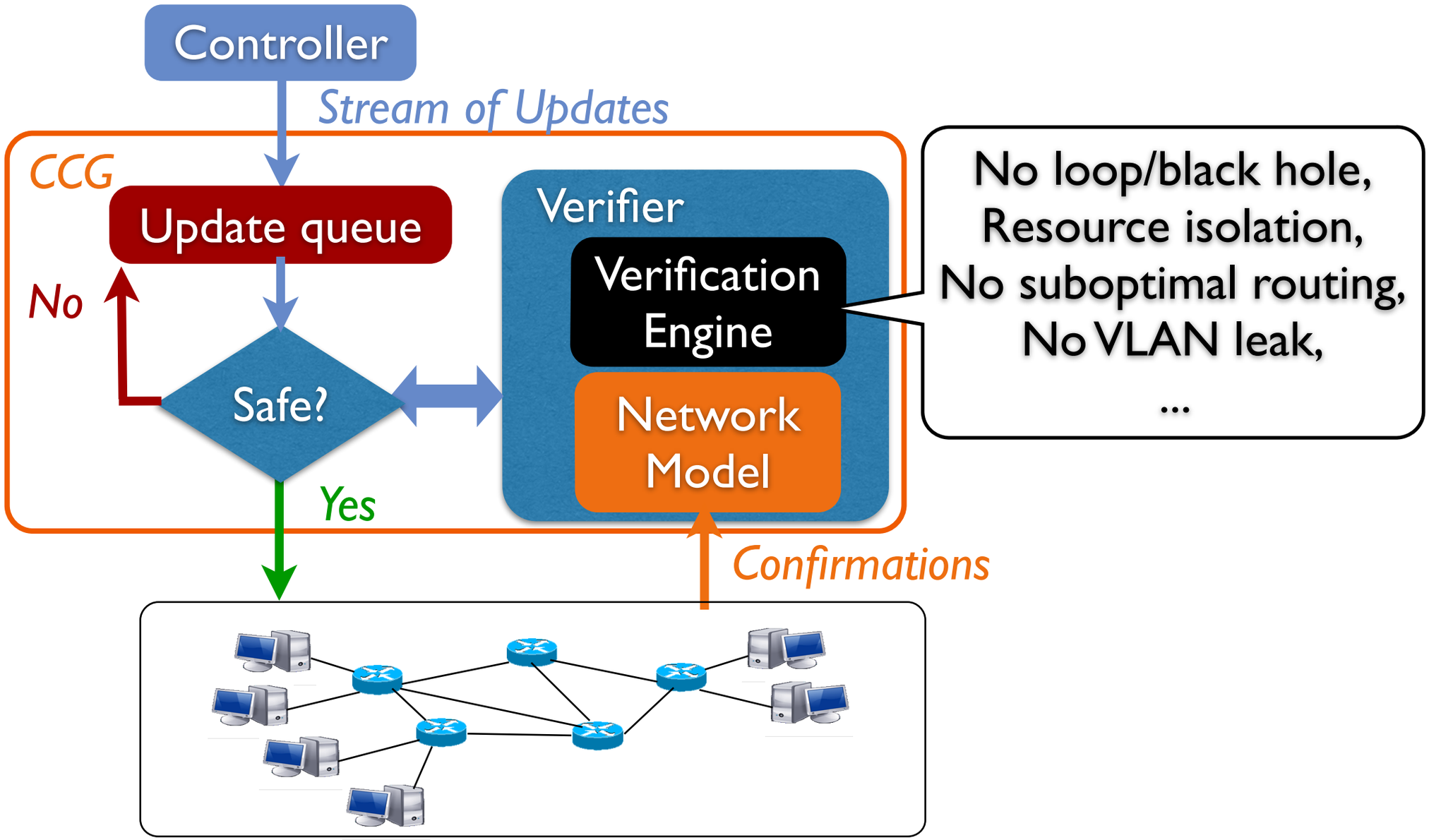
Our design: Customizable Consistency Generator

Key insight: Synthesis → Verification

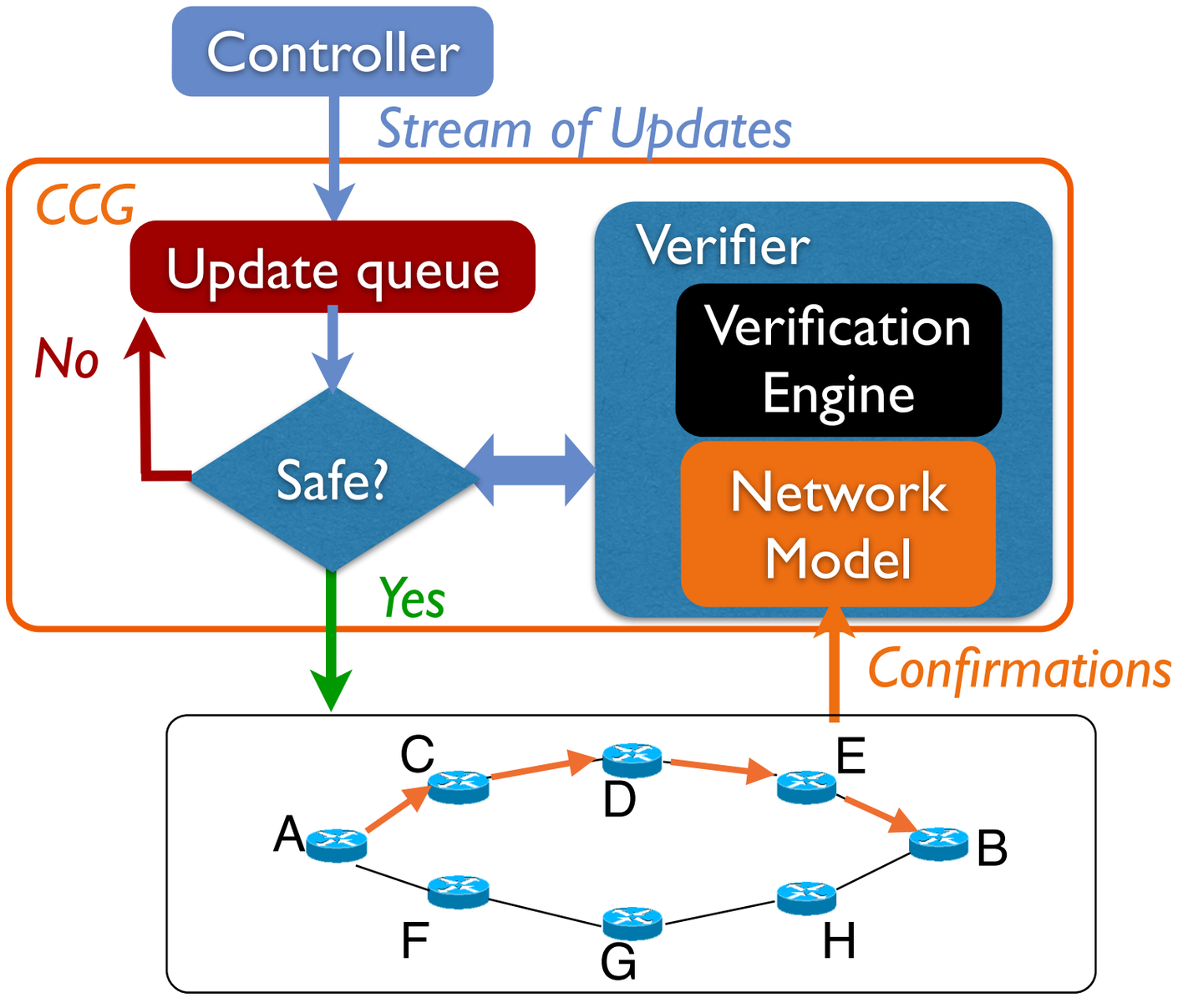


Our design: Customizable Consistency Generator

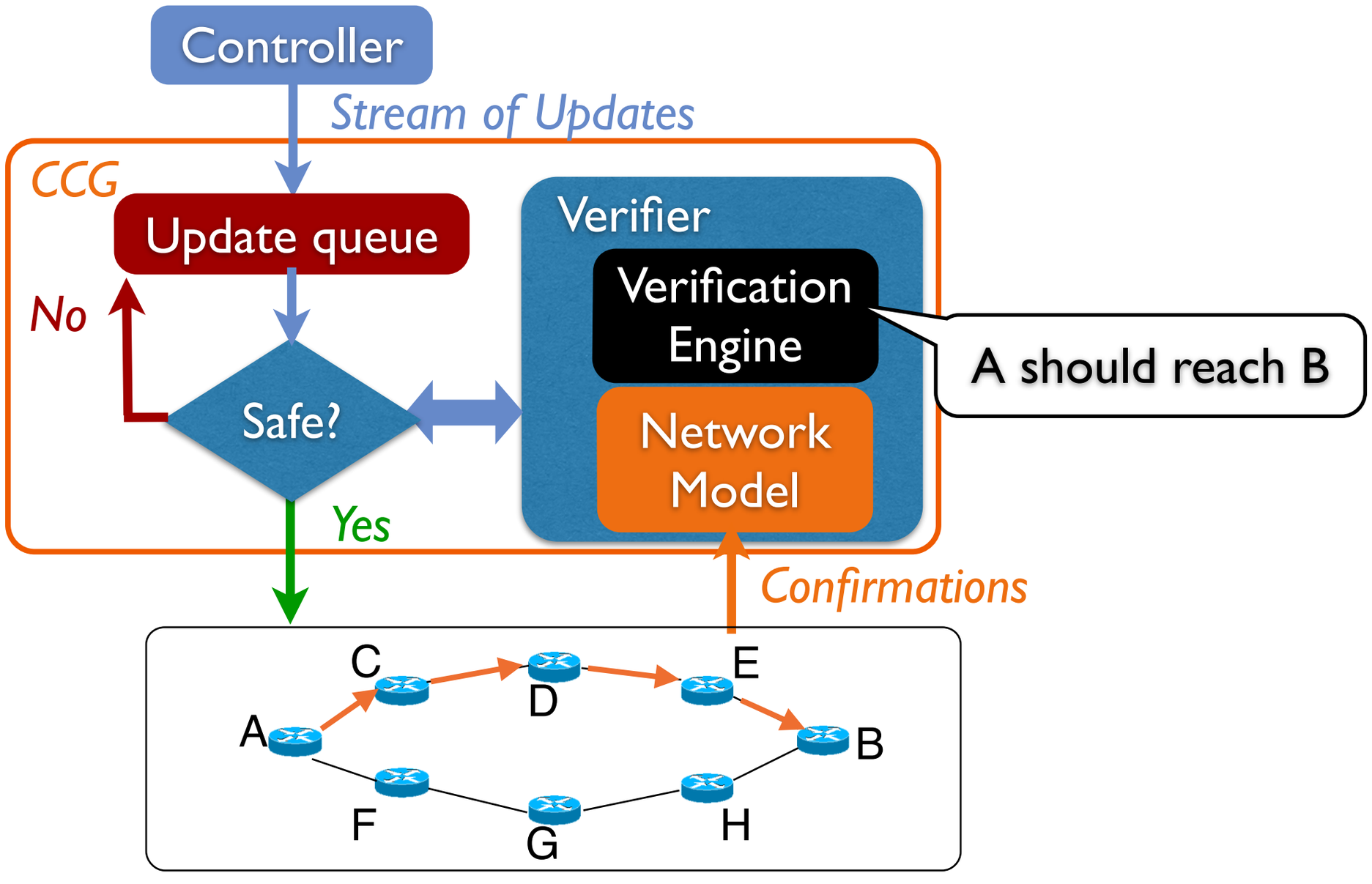
Key insight: Synthesis → Verification



Our design: Customizable Consistency Generator

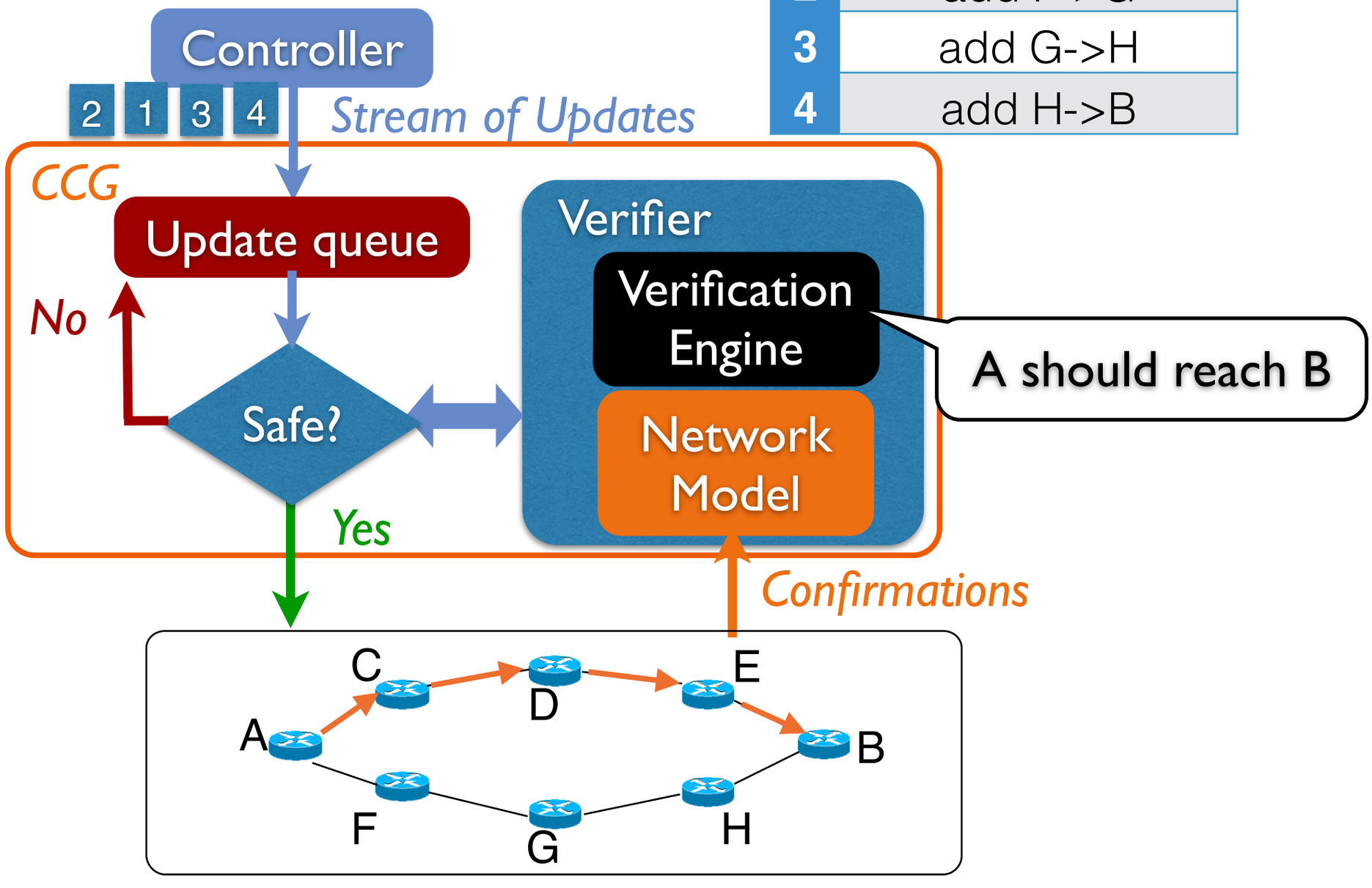


Our design: Customizable Consistency Generator

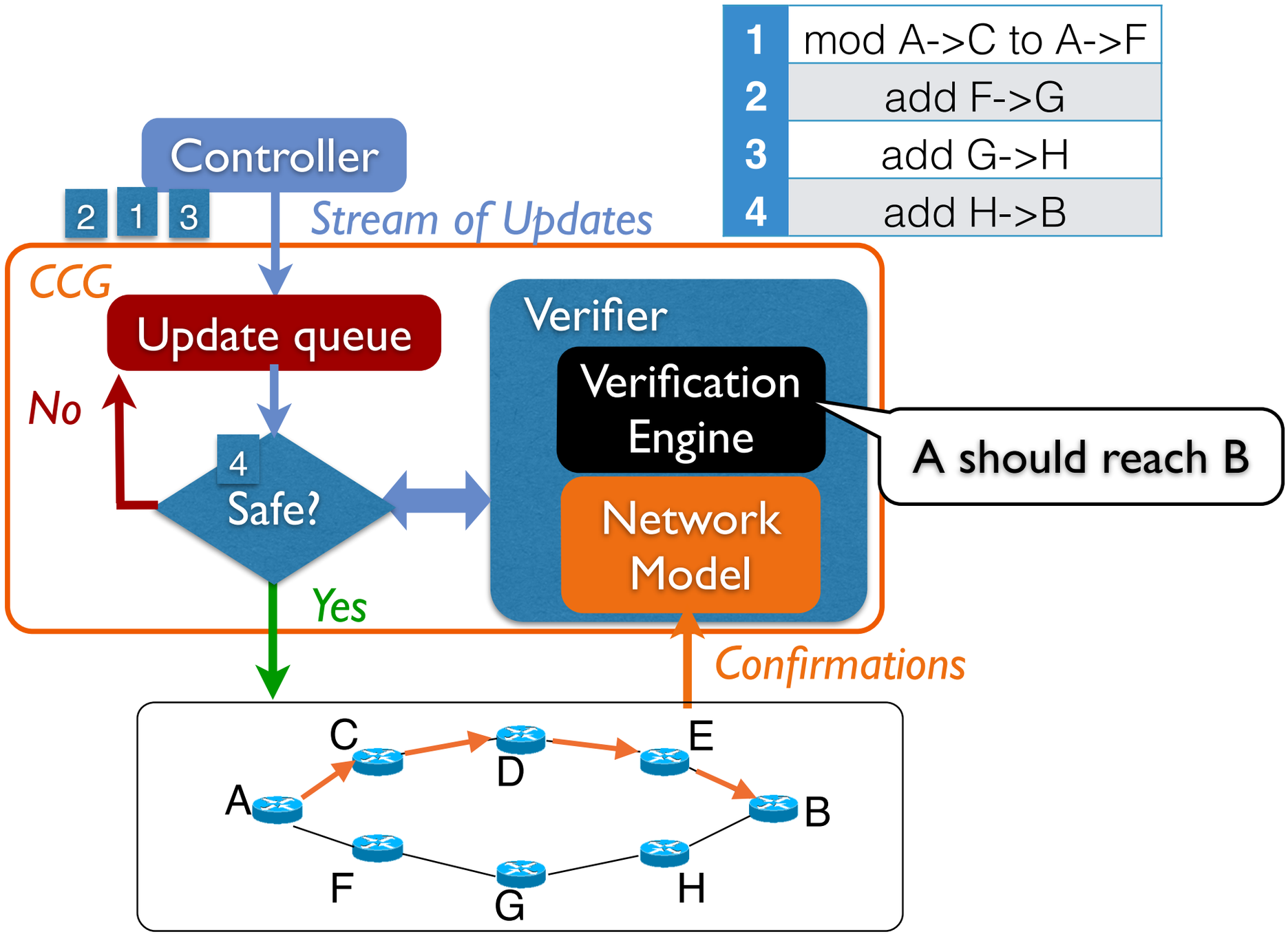


Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B

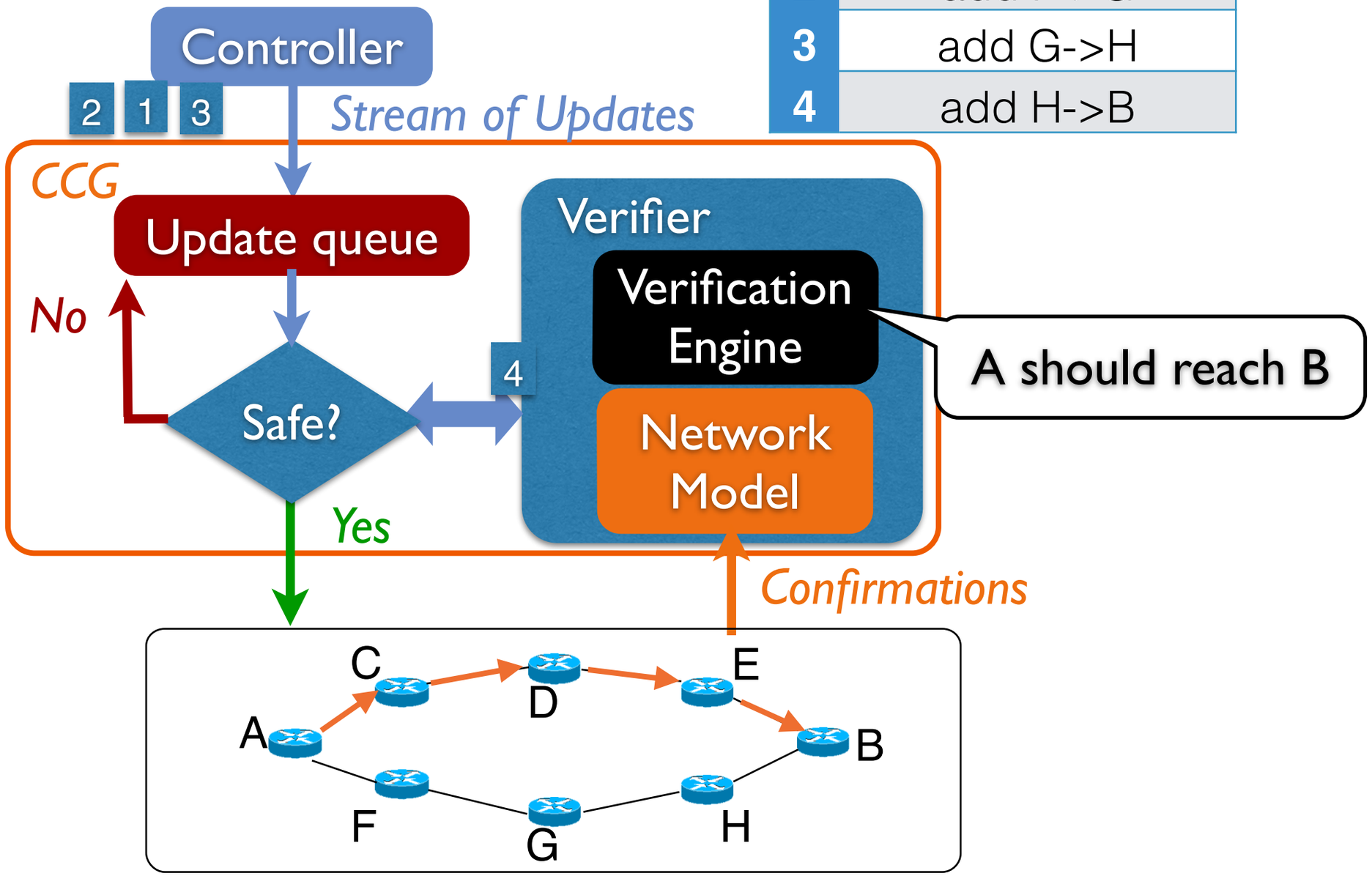


Our design: Customizable Consistency Generator



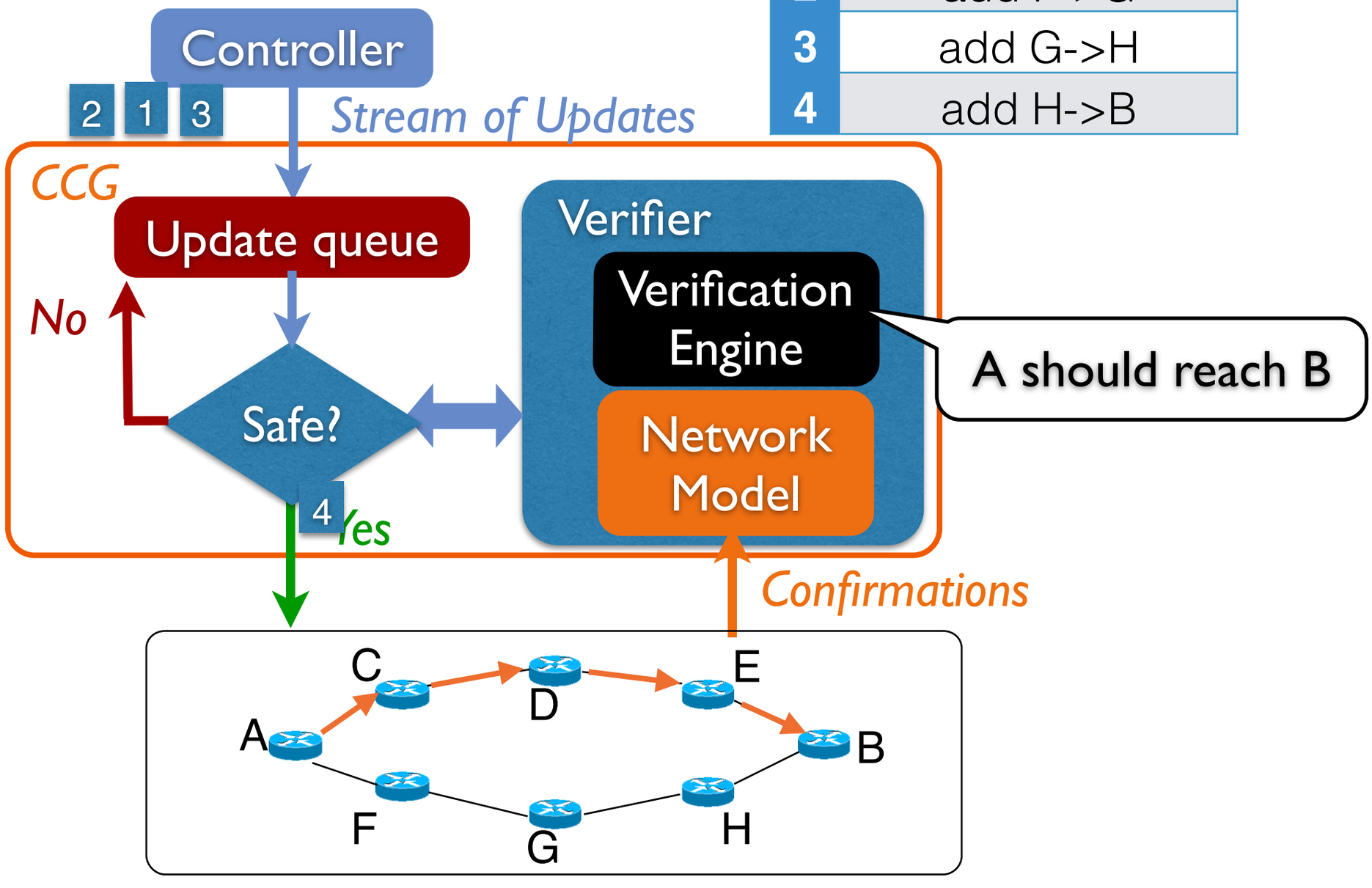
Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B



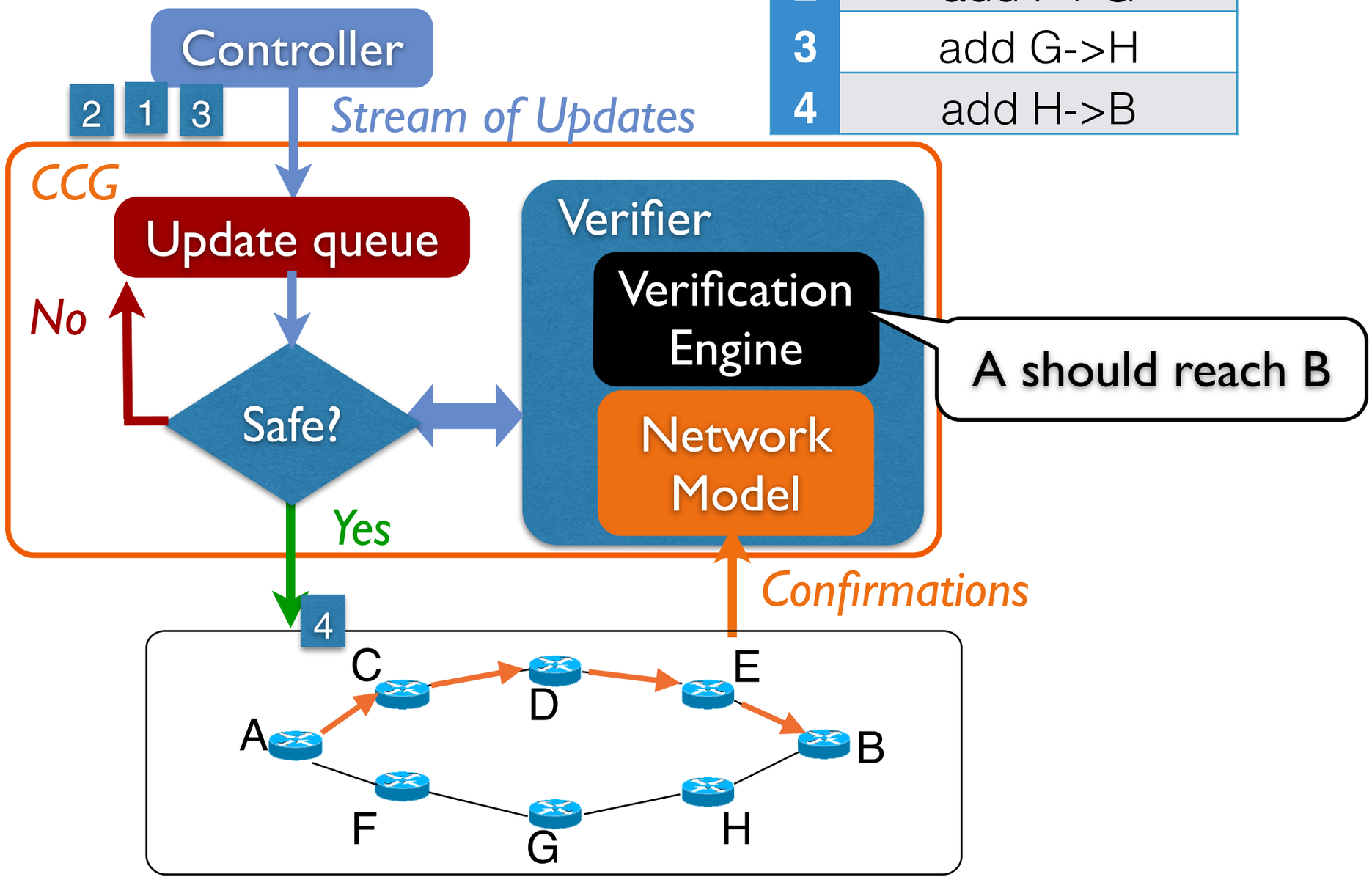
Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B



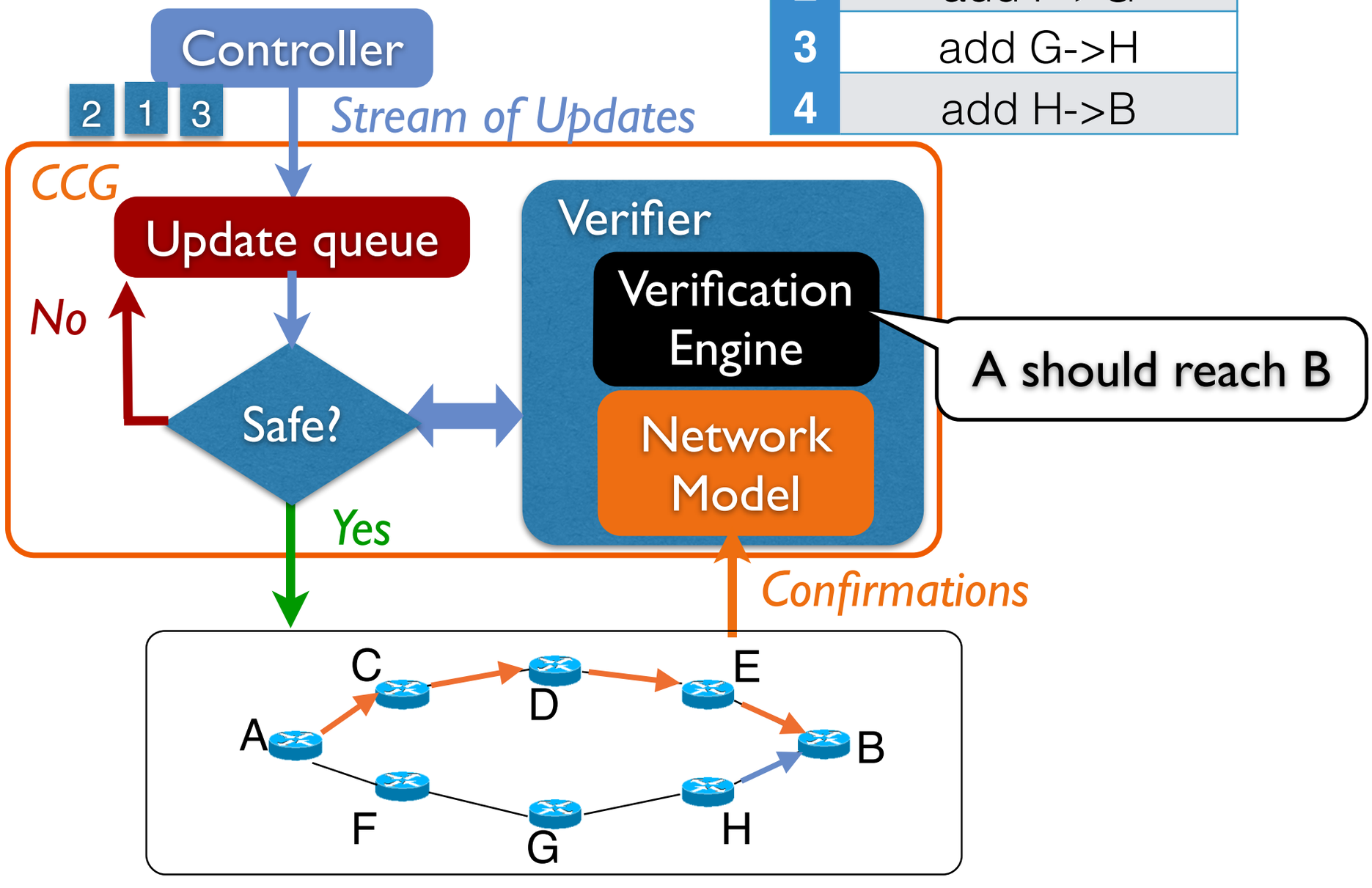
Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B



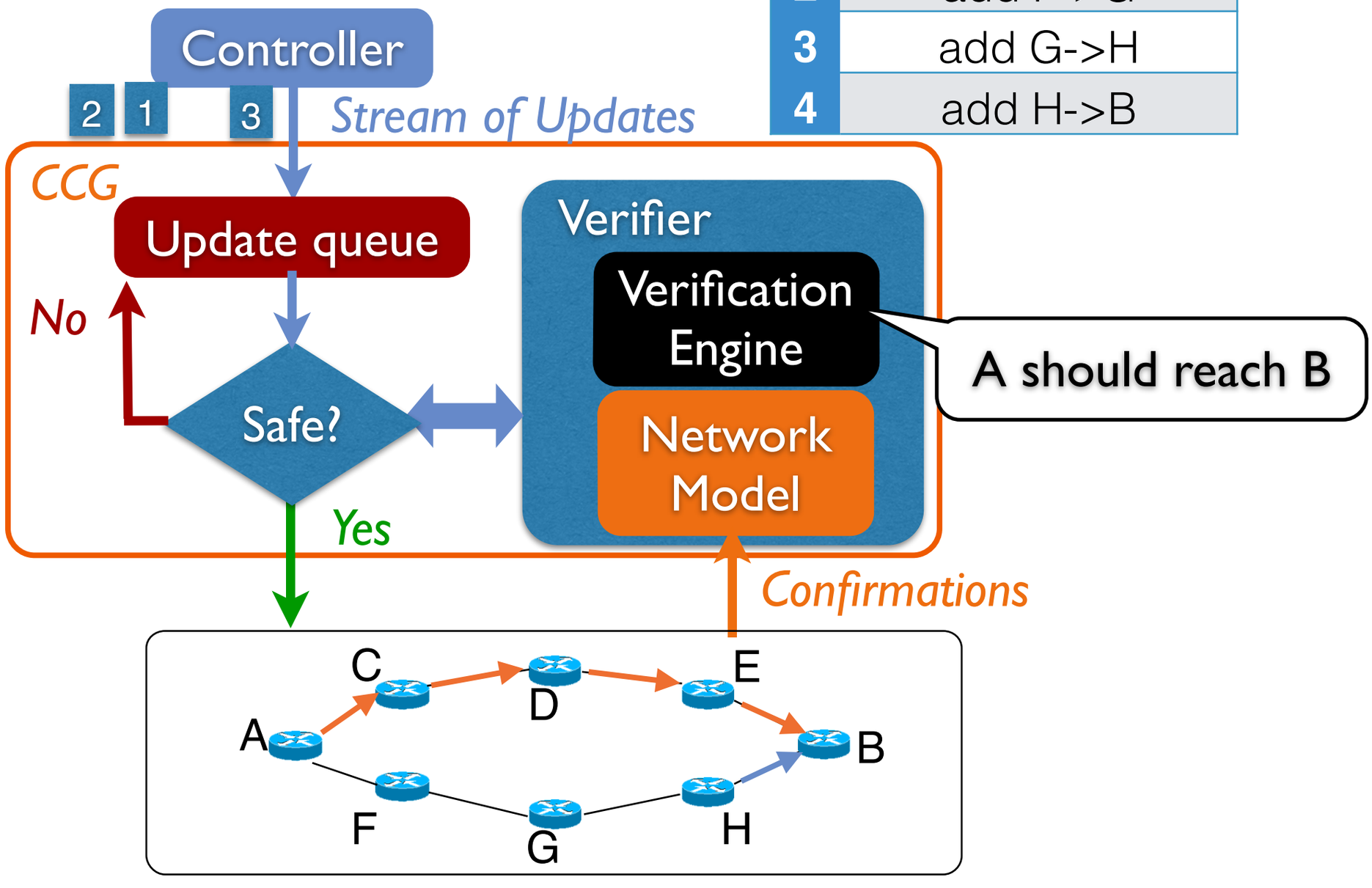
Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B

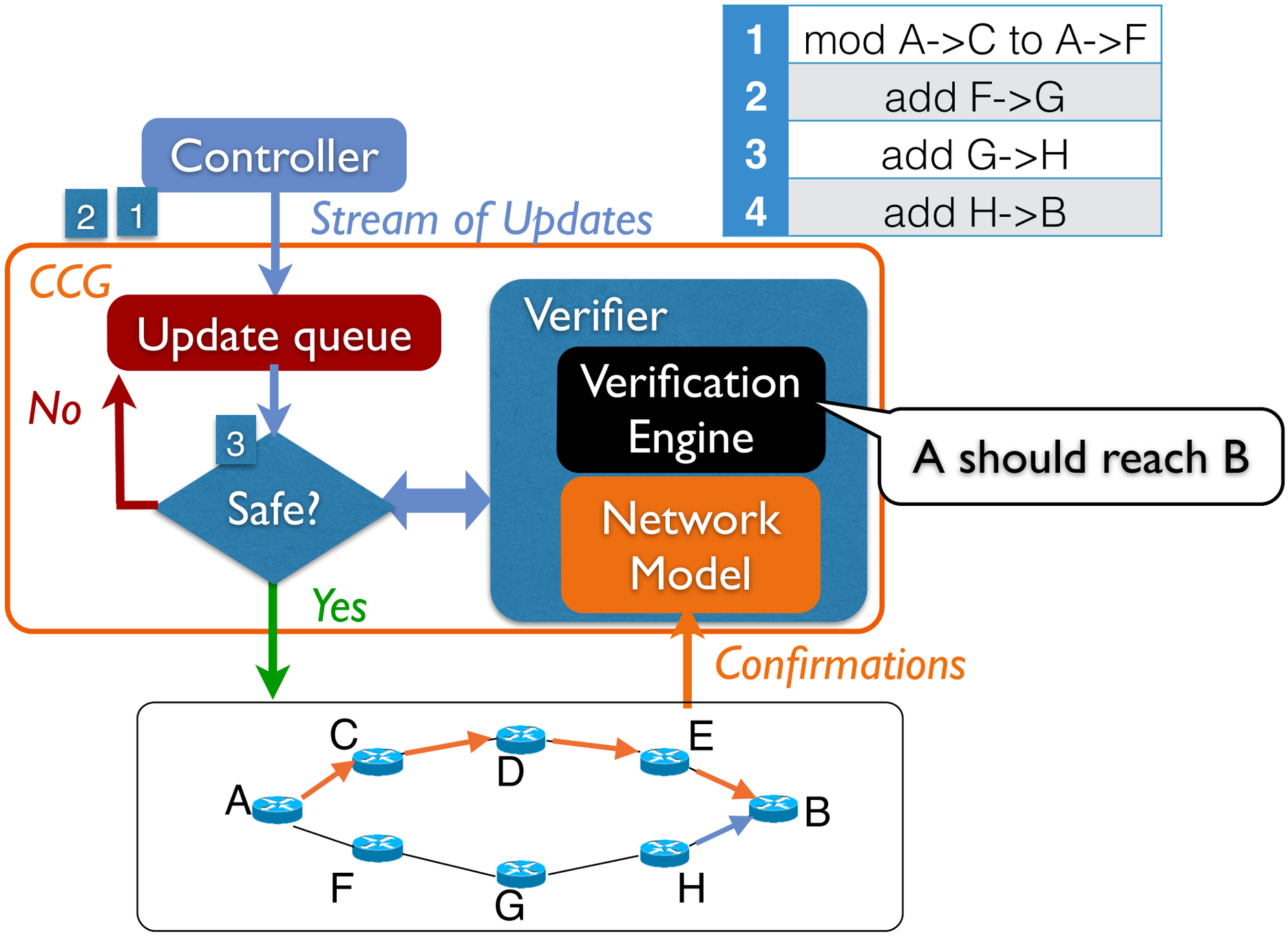


Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B

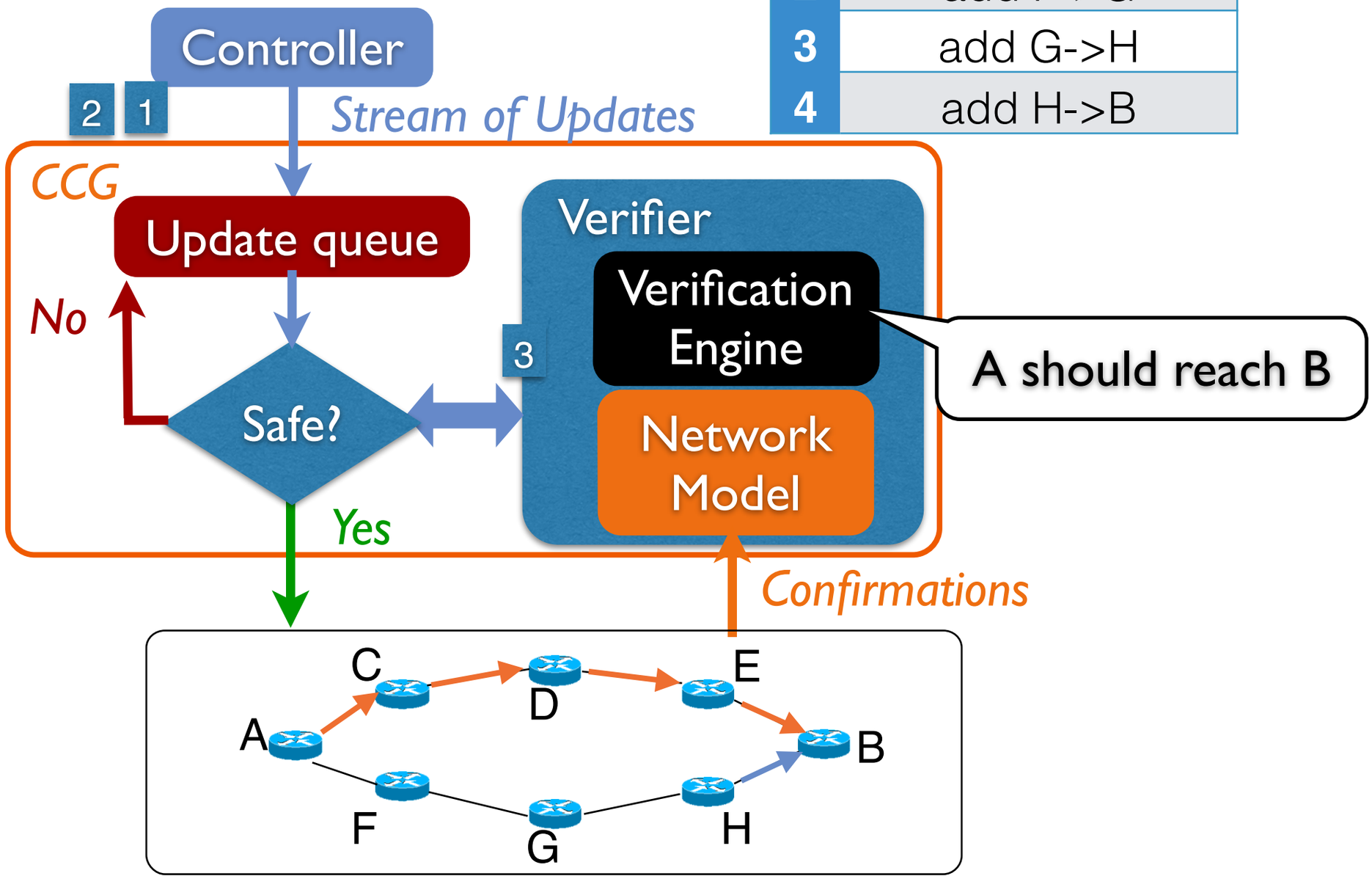


Our design: Customizable Consistency Generator

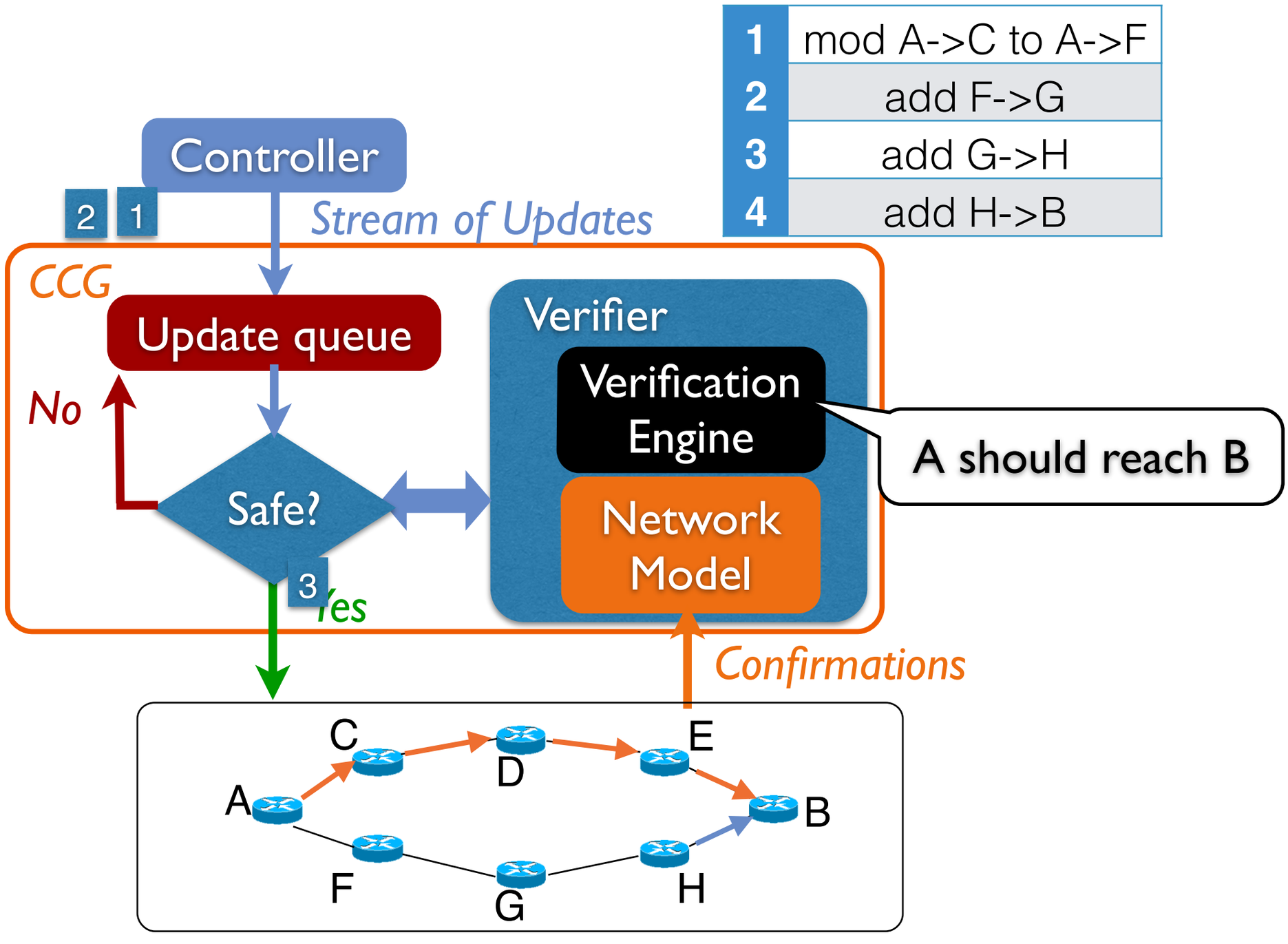


Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B

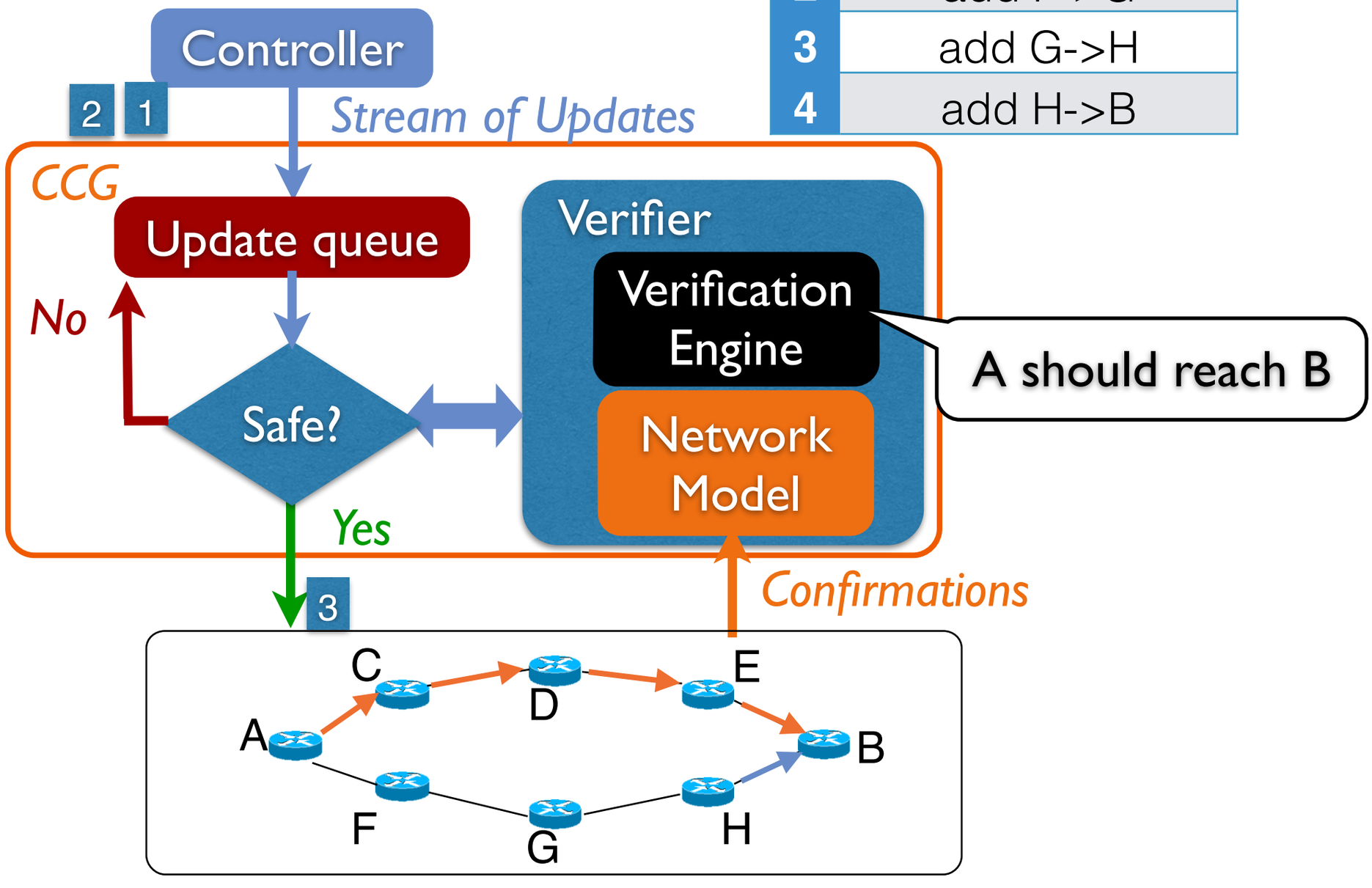


Our design: Customizable Consistency Generator

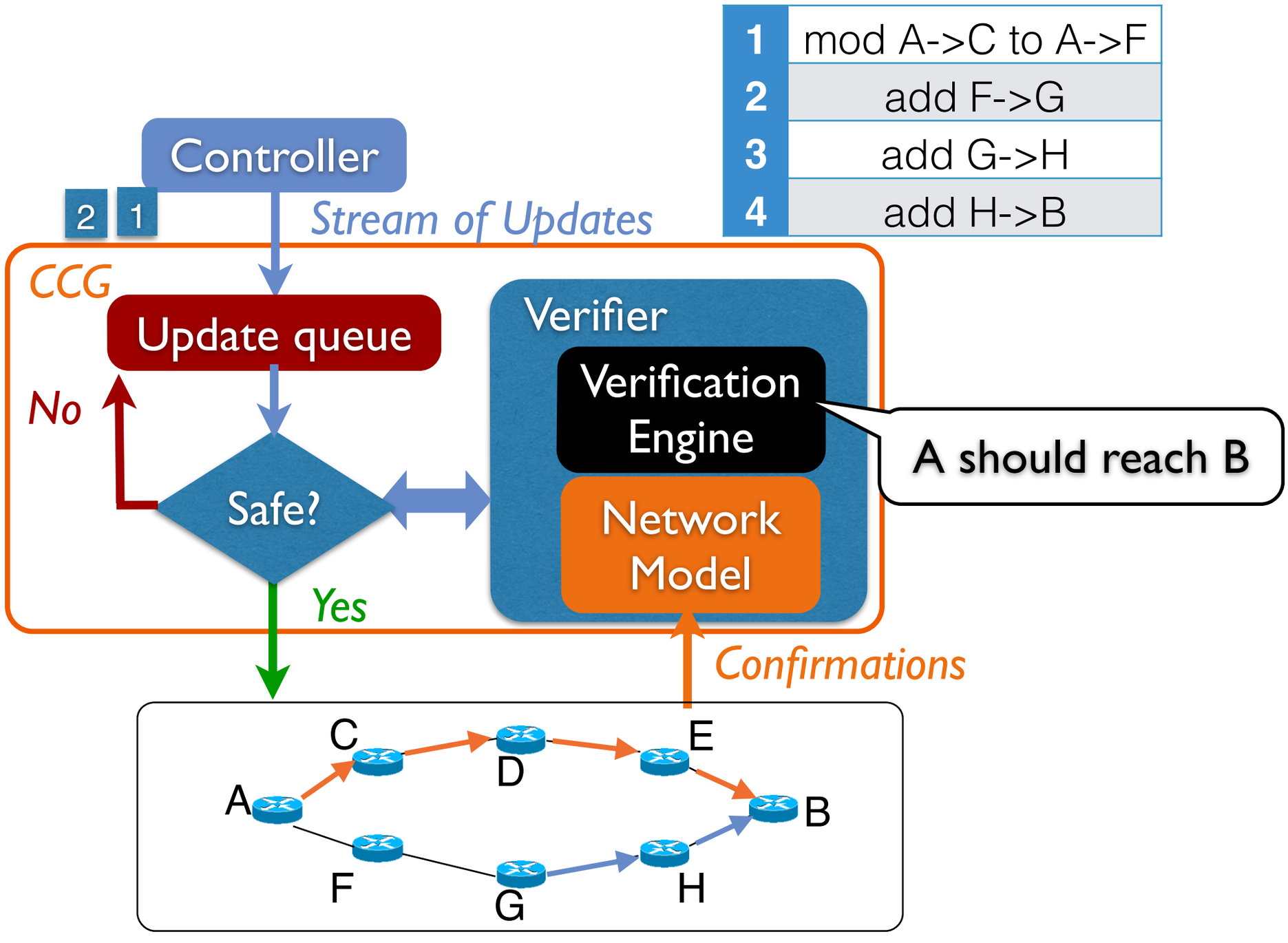


Our design: Customizable Consistency Generator

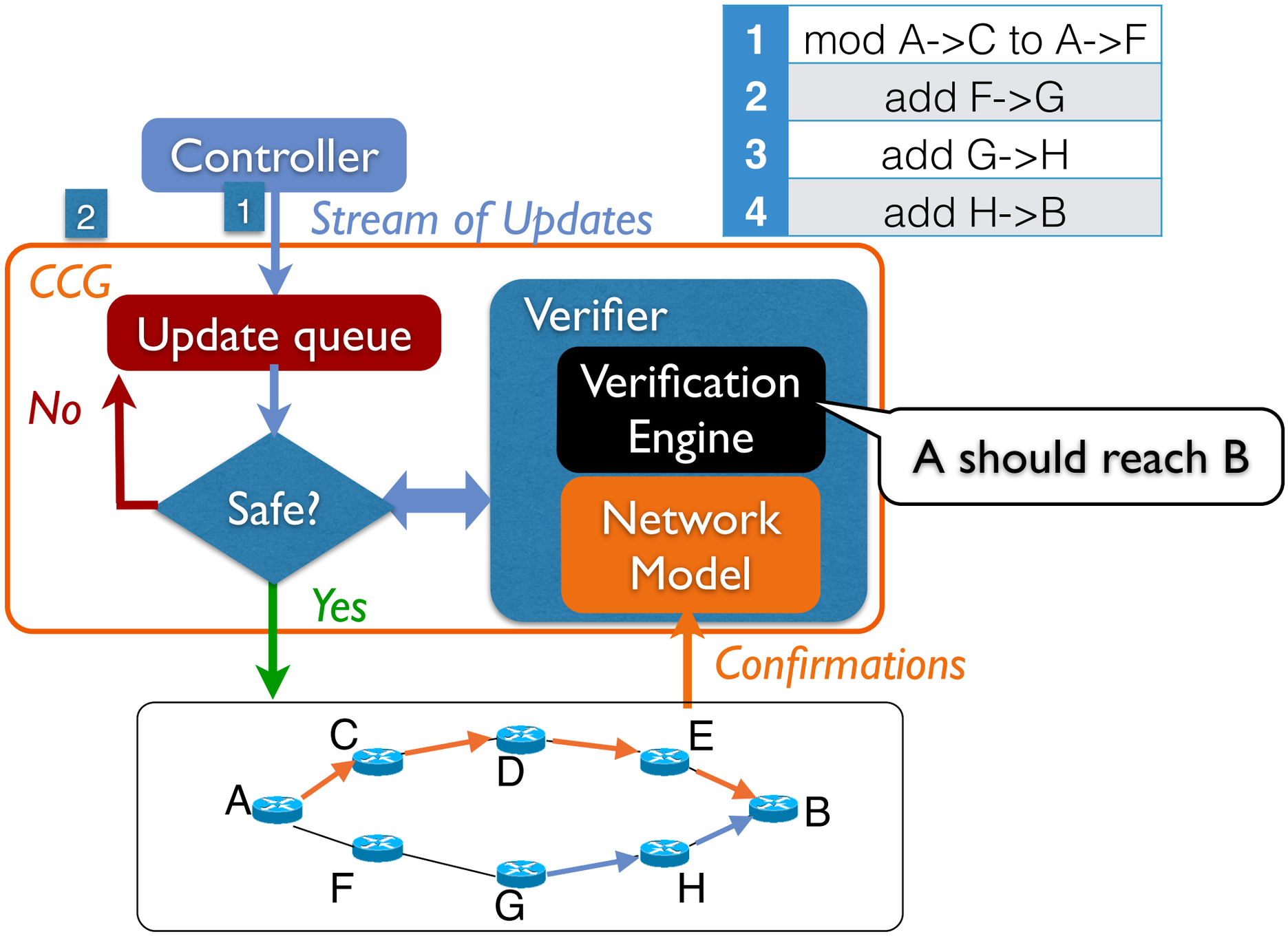
1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B



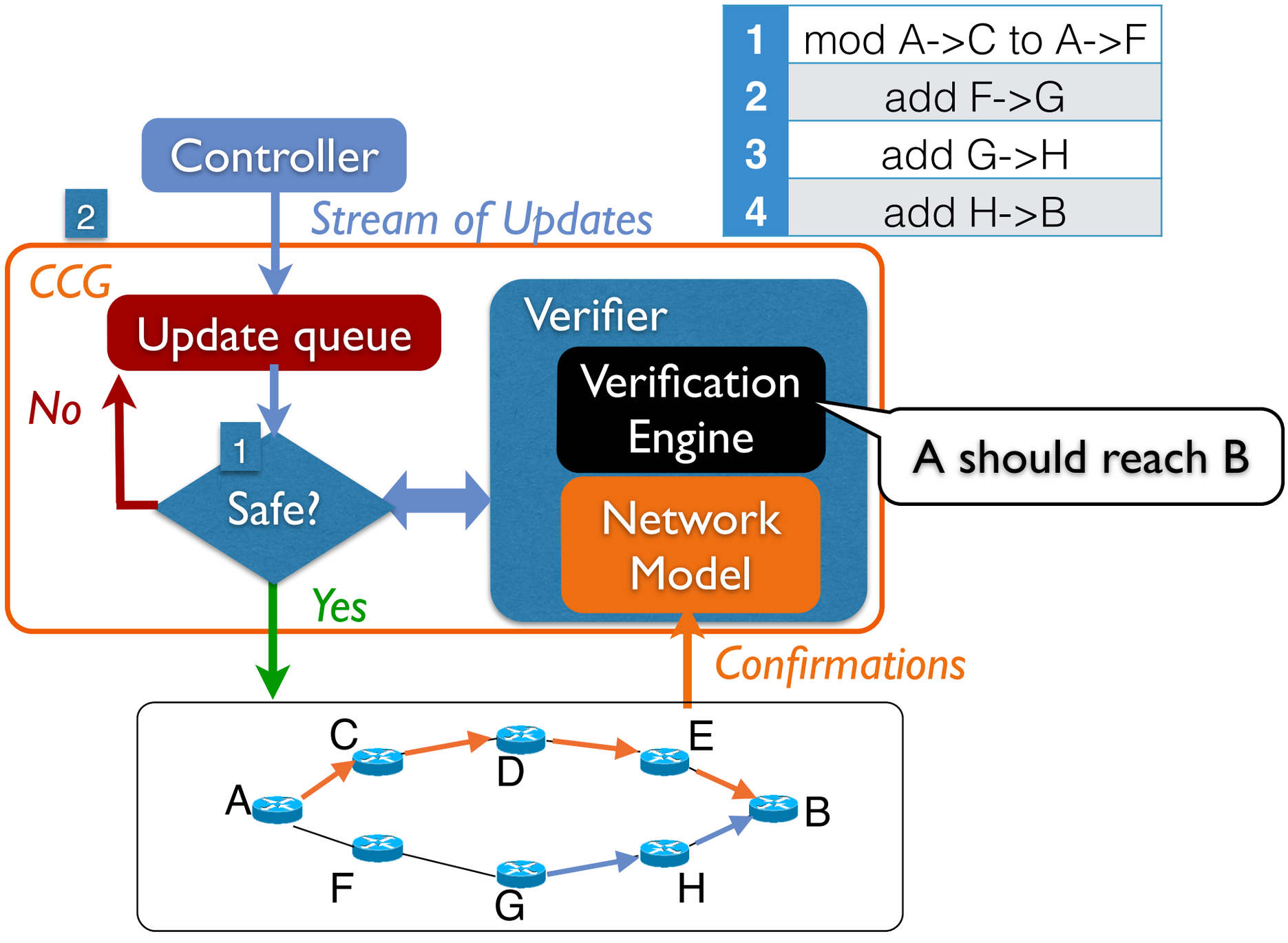
Our design: Customizable Consistency Generator



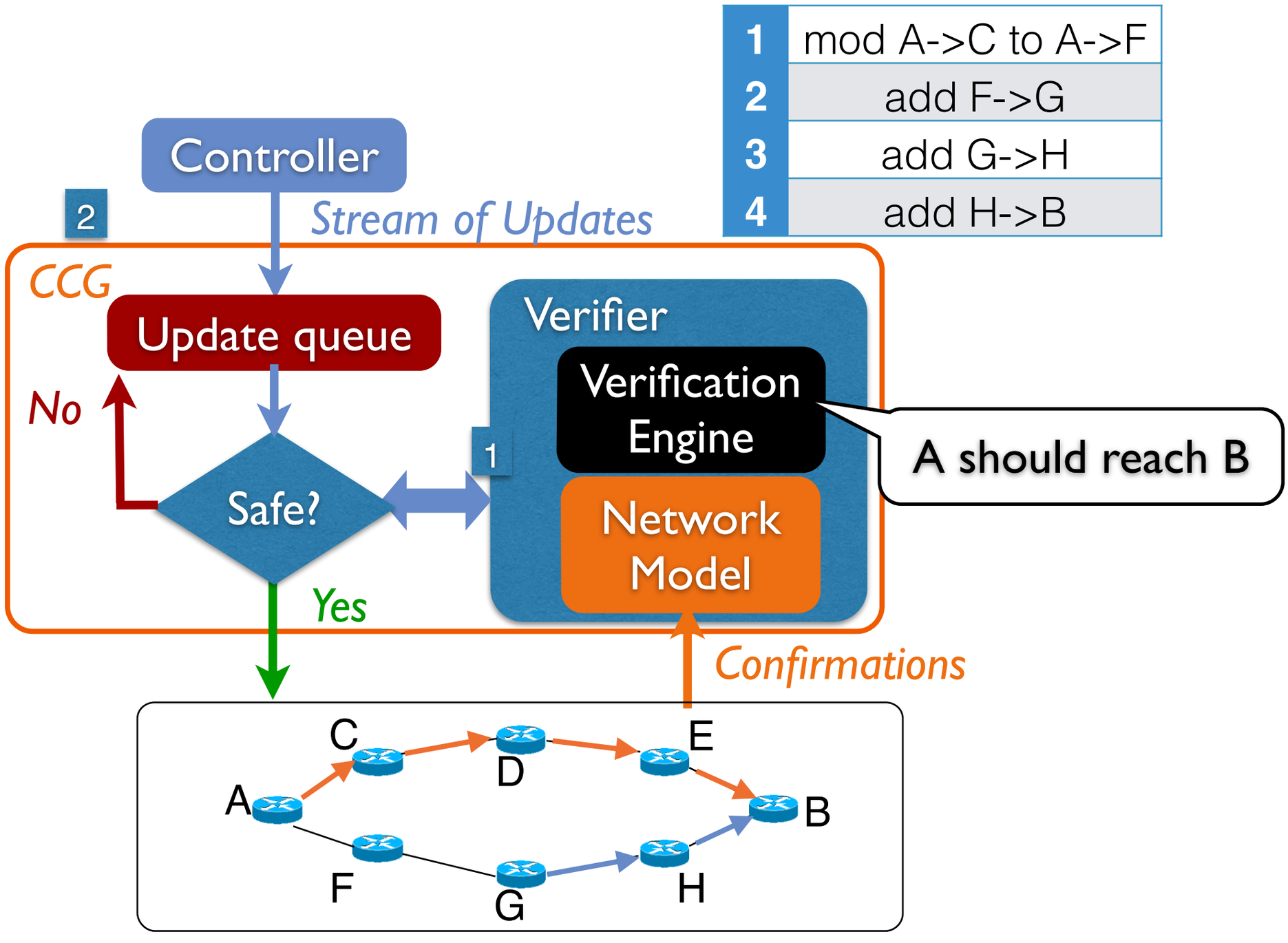
Our design: Customizable Consistency Generator



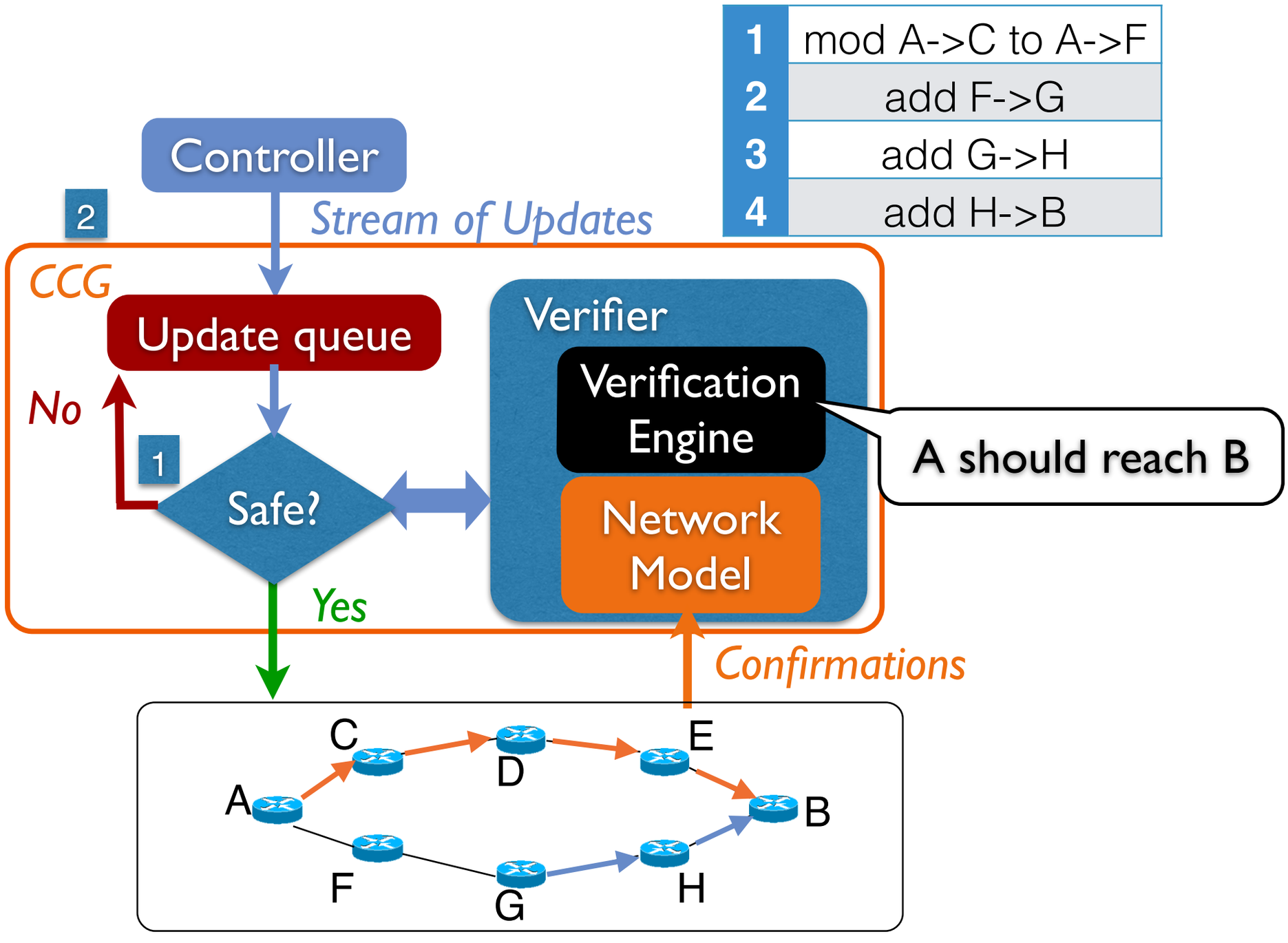
Our design: Customizable Consistency Generator



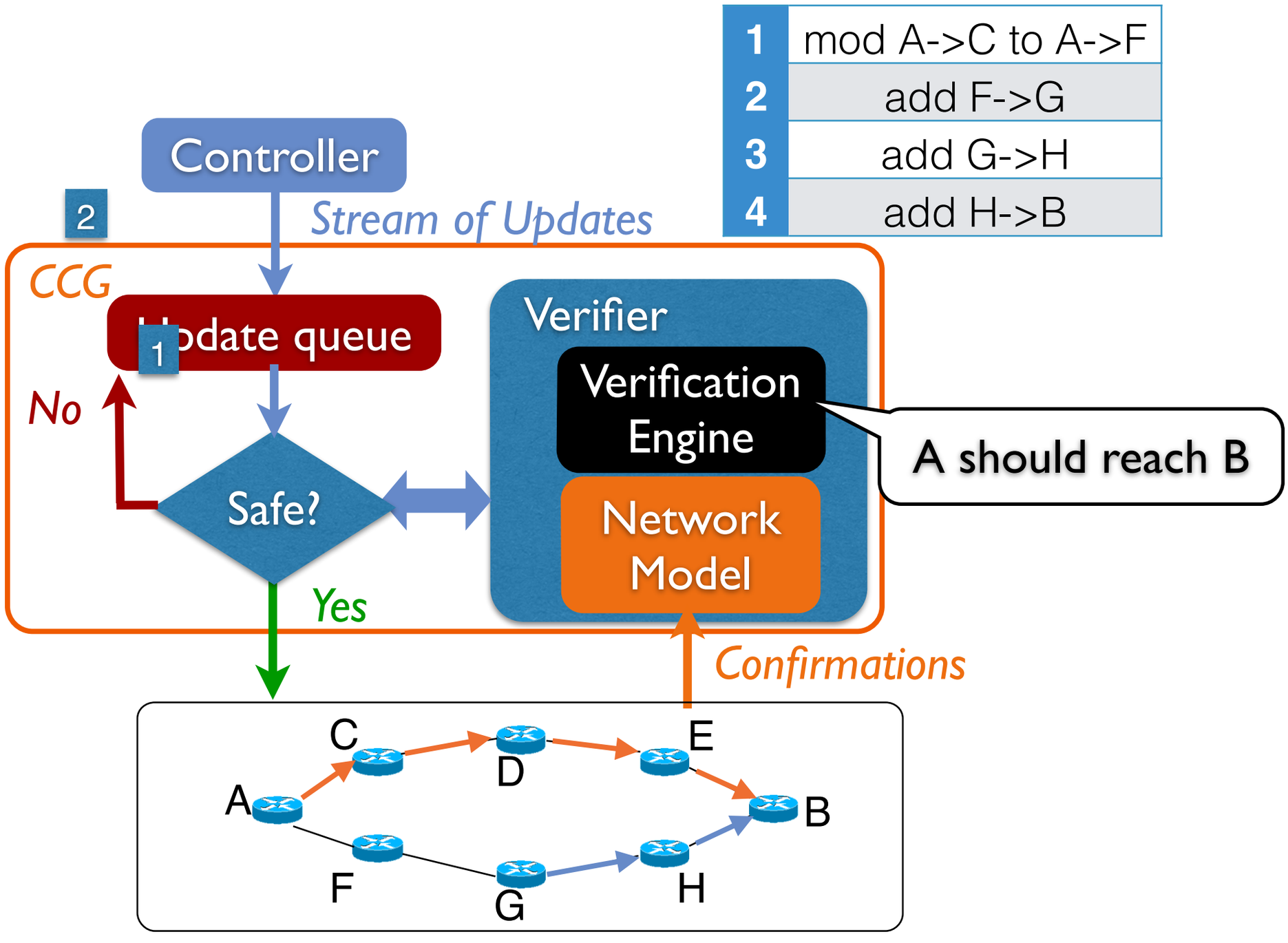
Our design: Customizable Consistency Generator



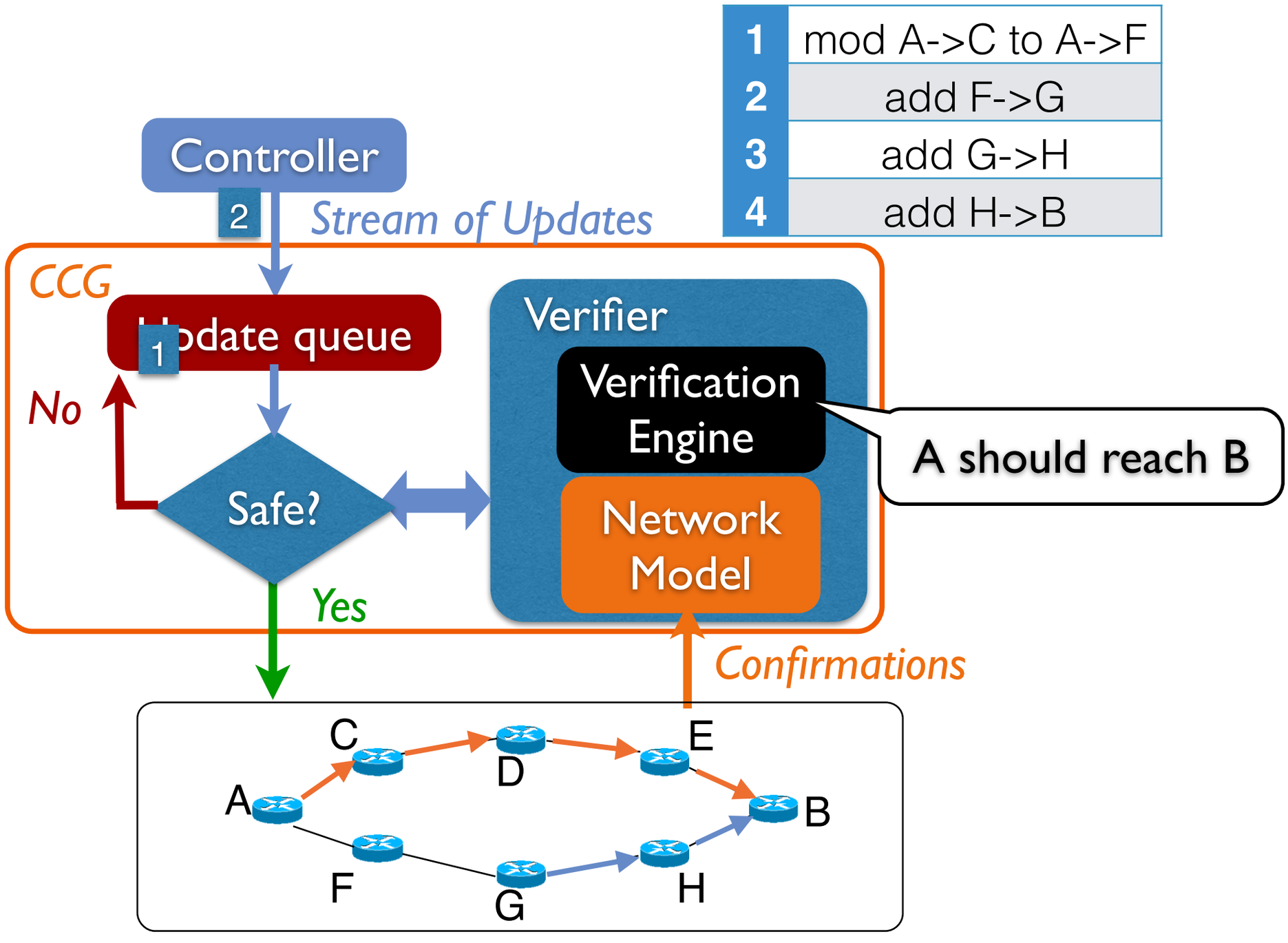
Our design: Customizable Consistency Generator



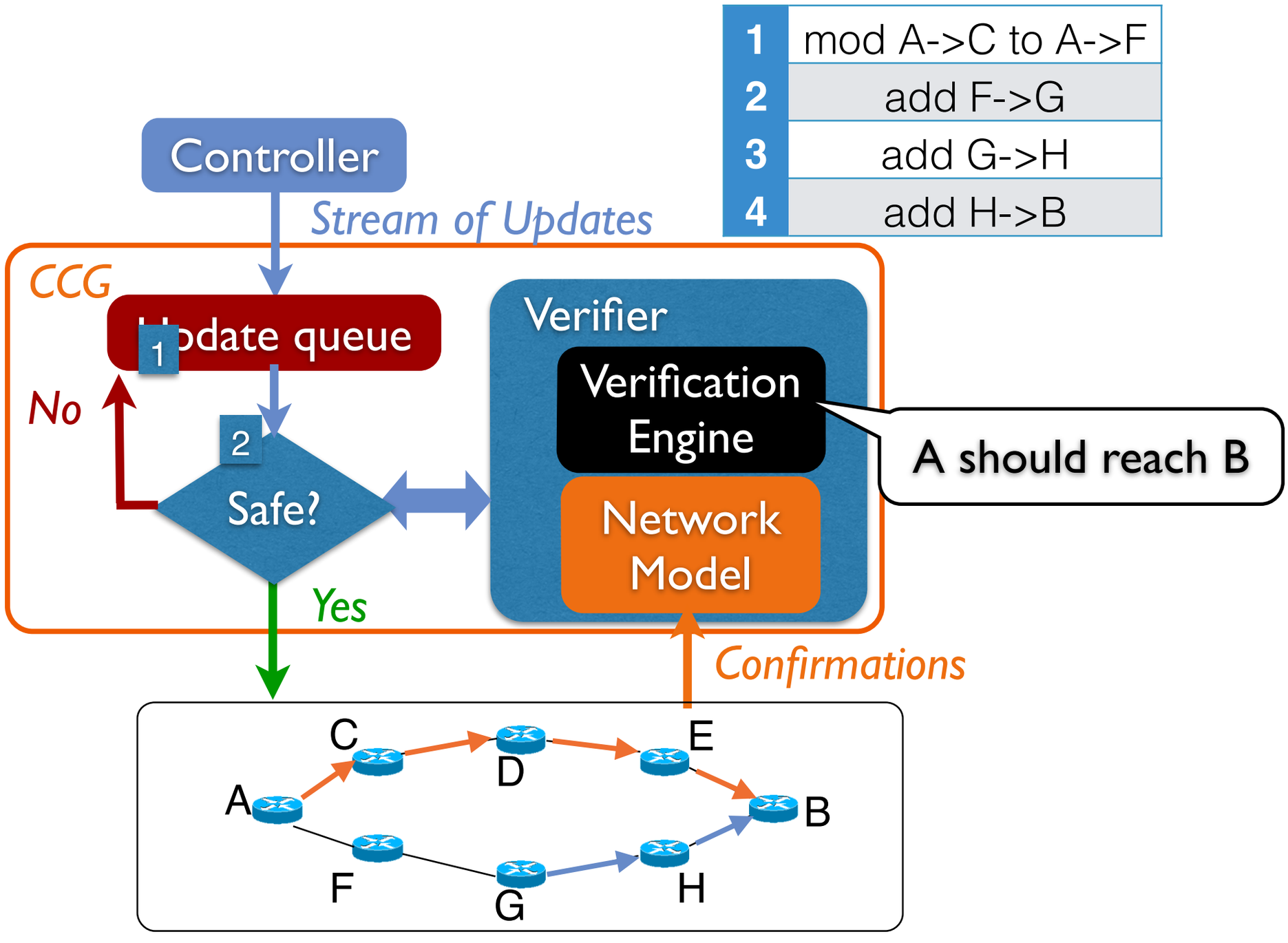
Our design: Customizable Consistency Generator



Our design: Customizable Consistency Generator

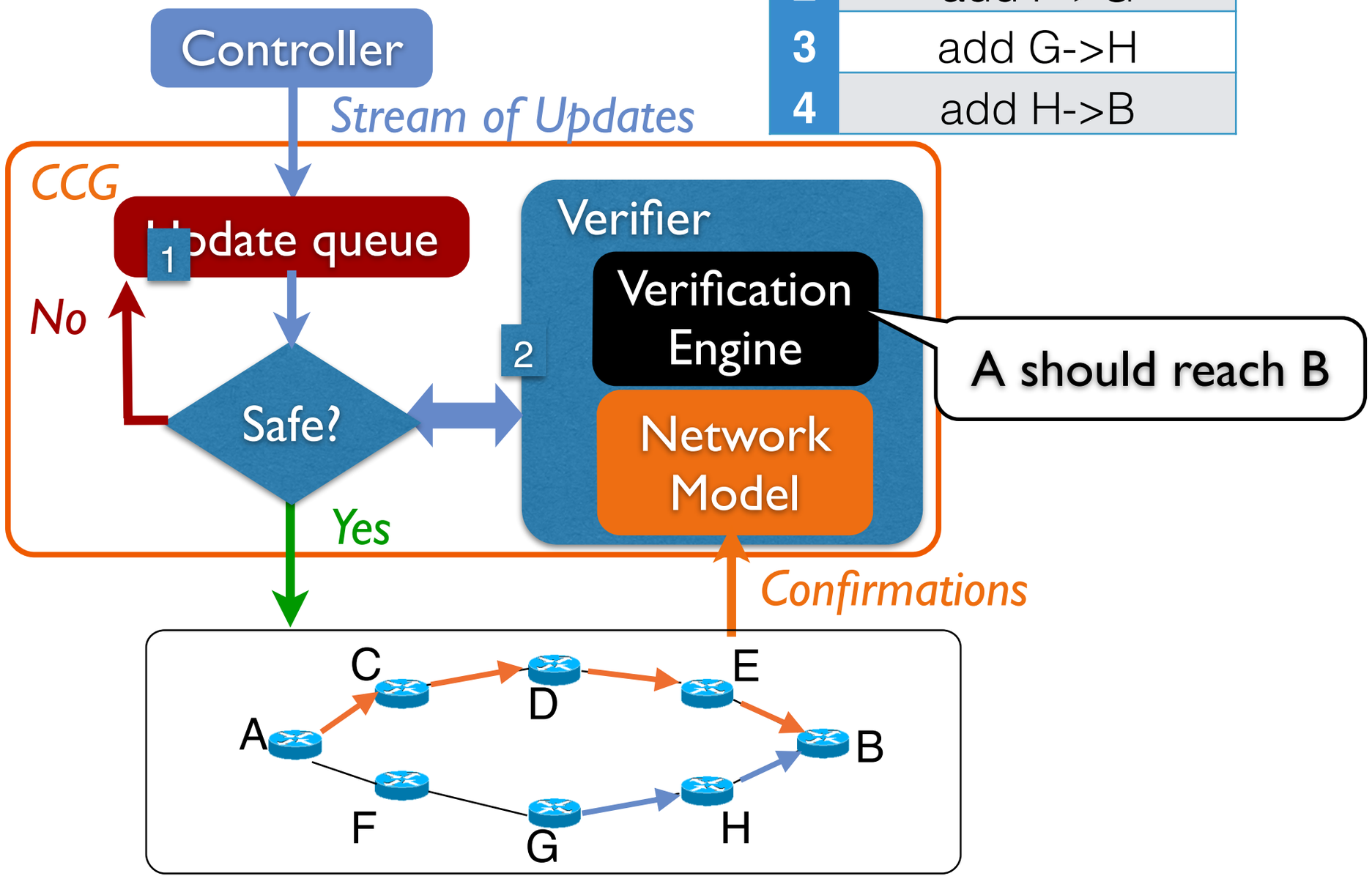


Our design: Customizable Consistency Generator



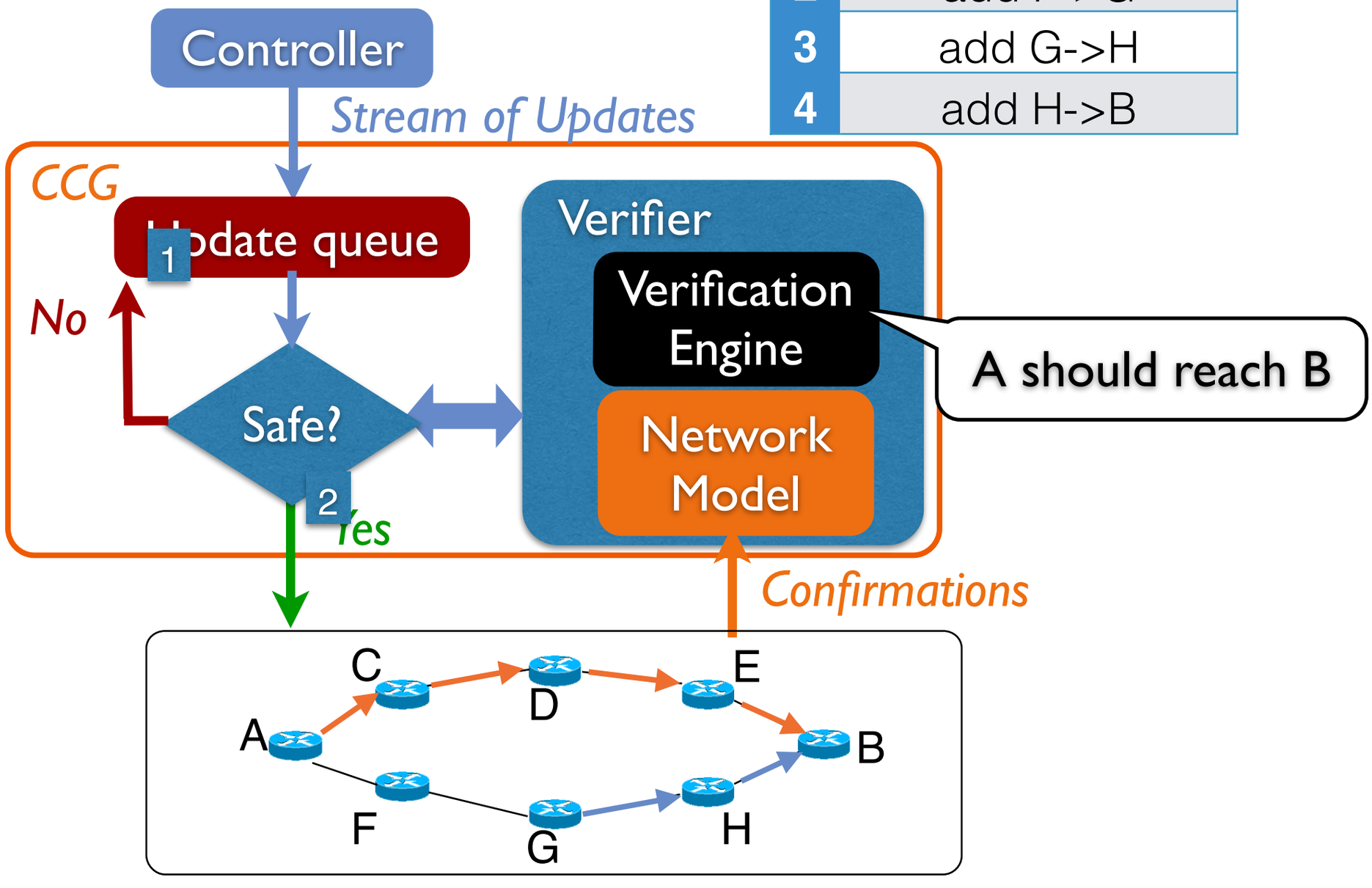
Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B



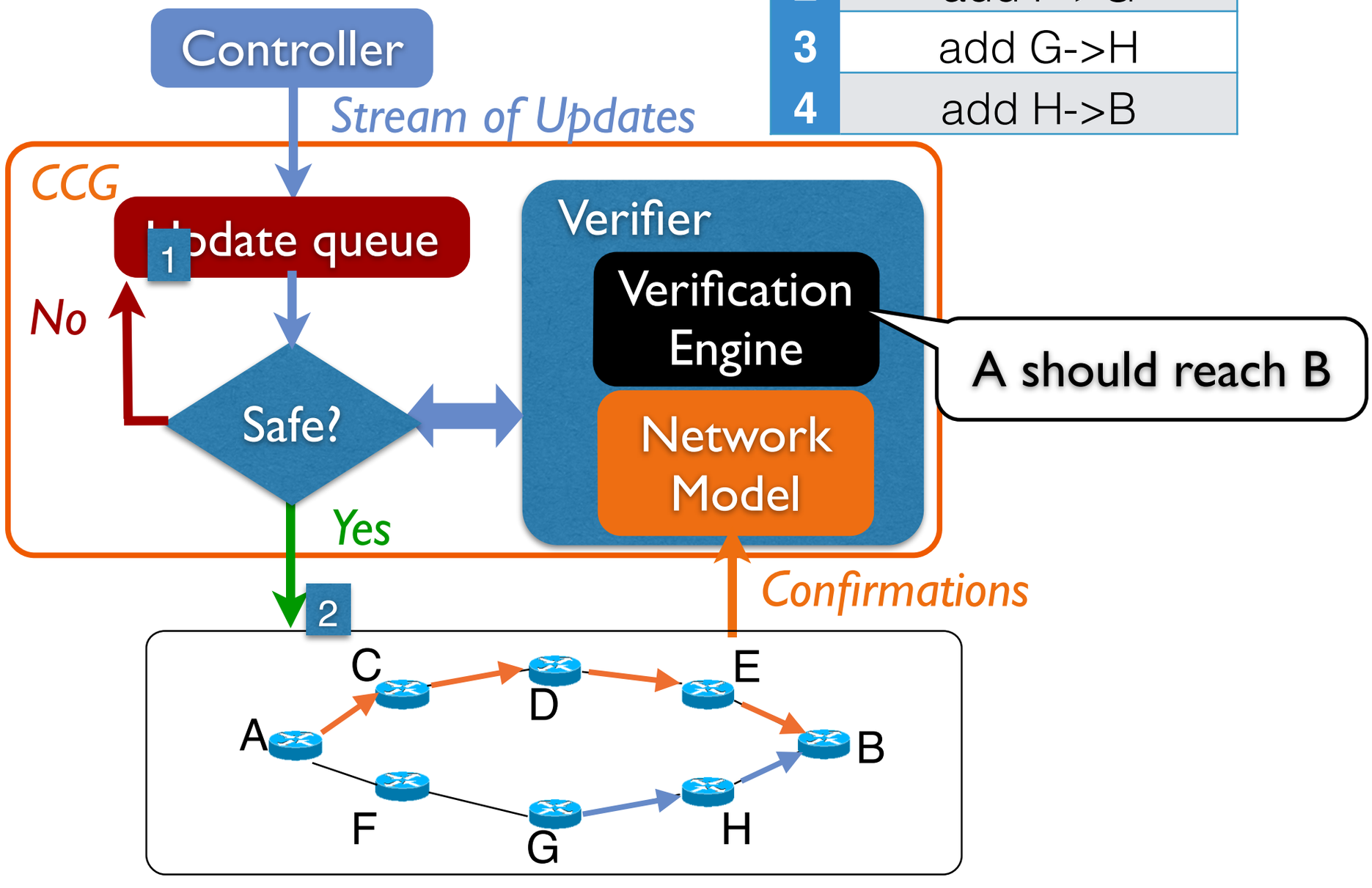
Our design: Customizable Consistency Generator

1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B

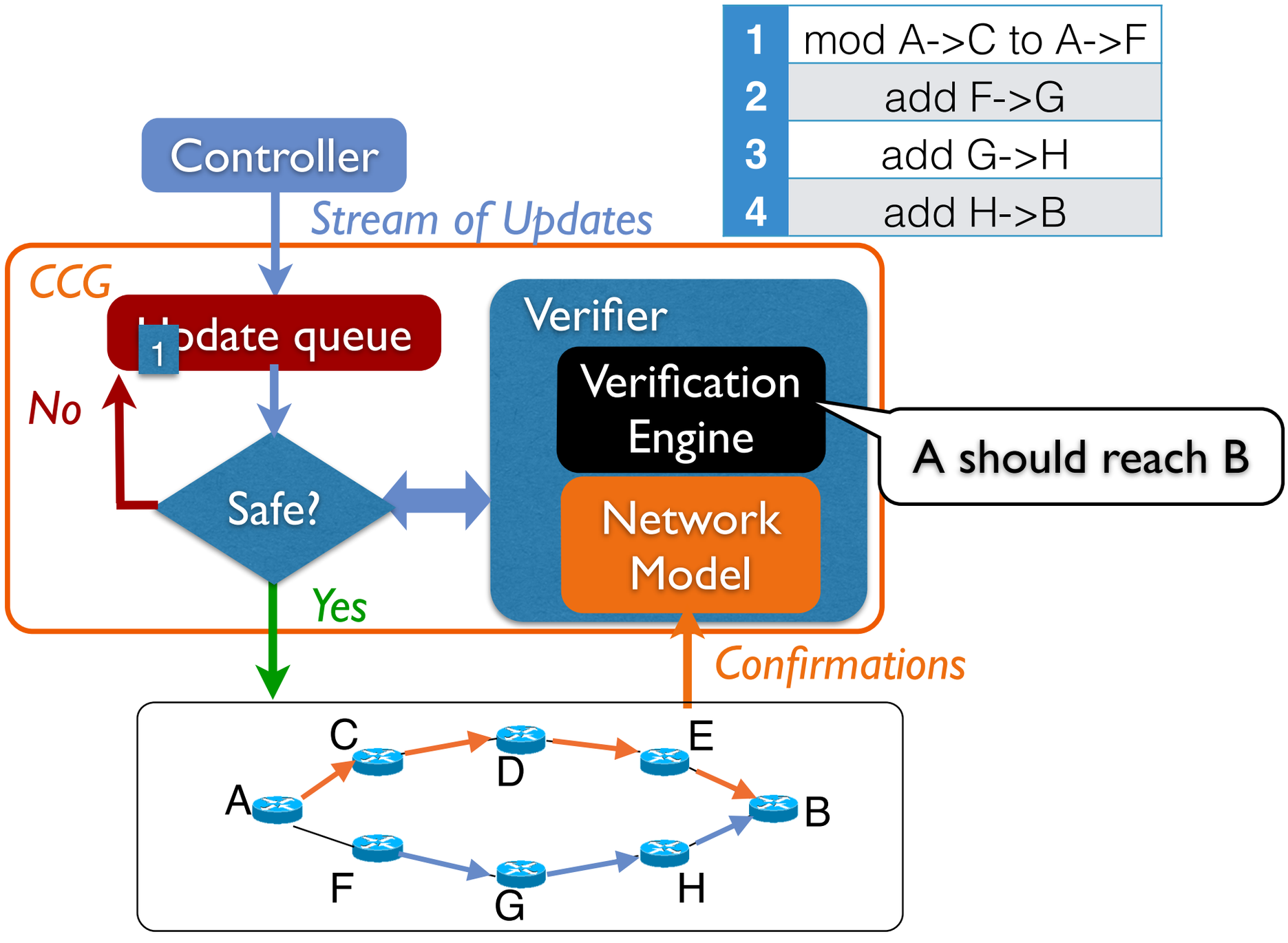


Our design: Customizable Consistency Generator

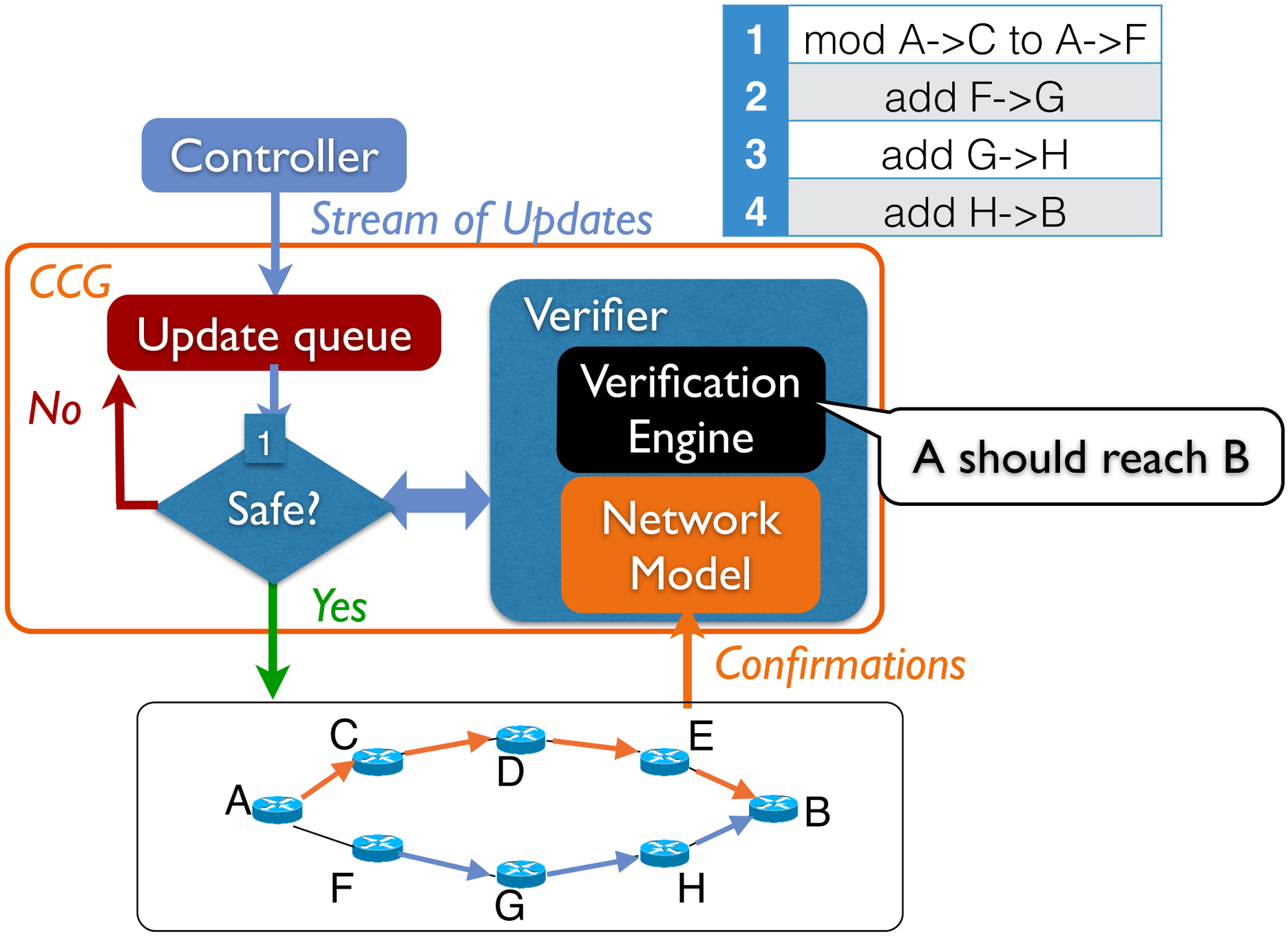
1	mod A->C to A->F
2	add F->G
3	add G->H
4	add H->B



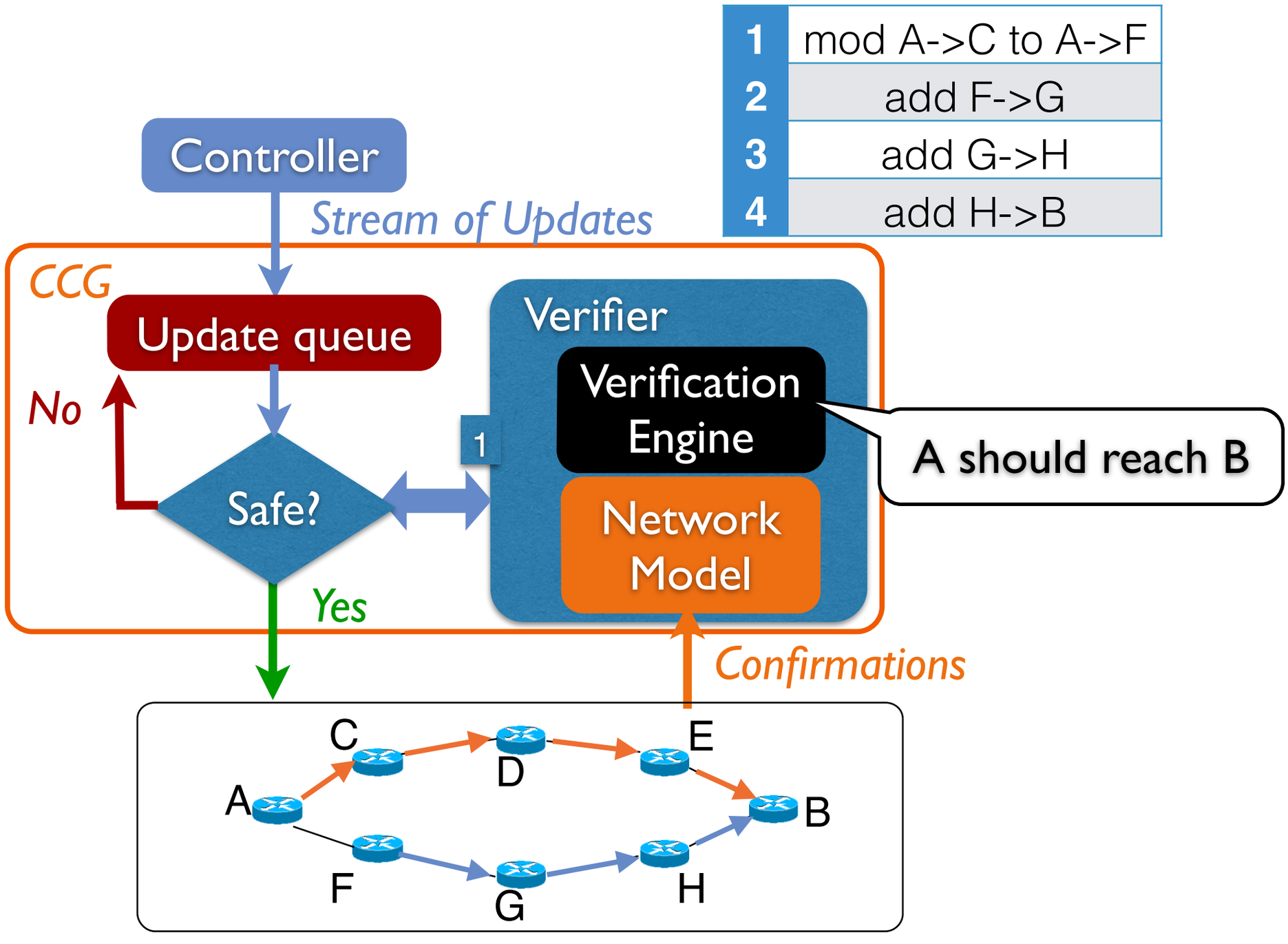
Our design: Customizable Consistency Generator



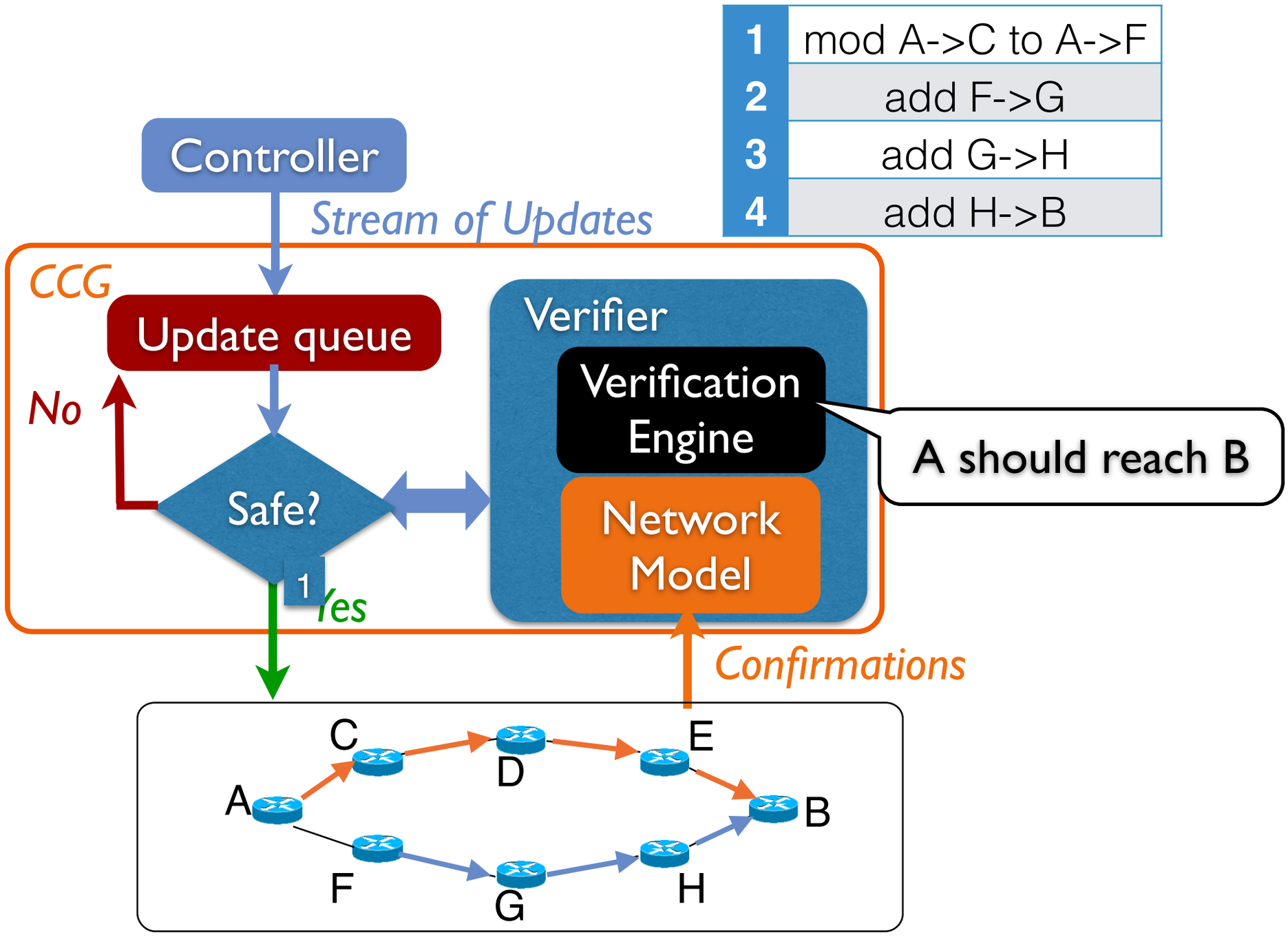
Our design: Customizable Consistency Generator



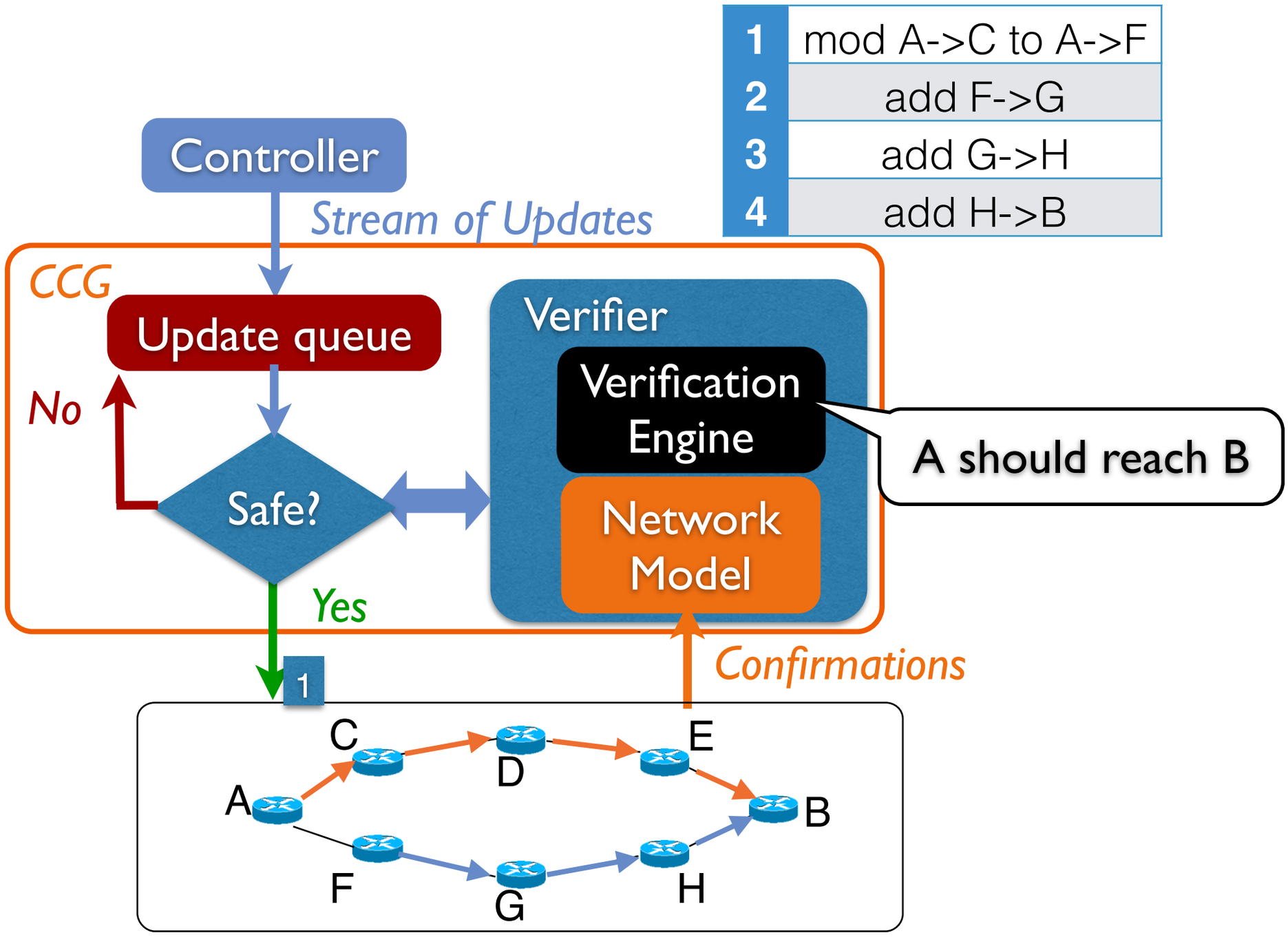
Our design: Customizable Consistency Generator



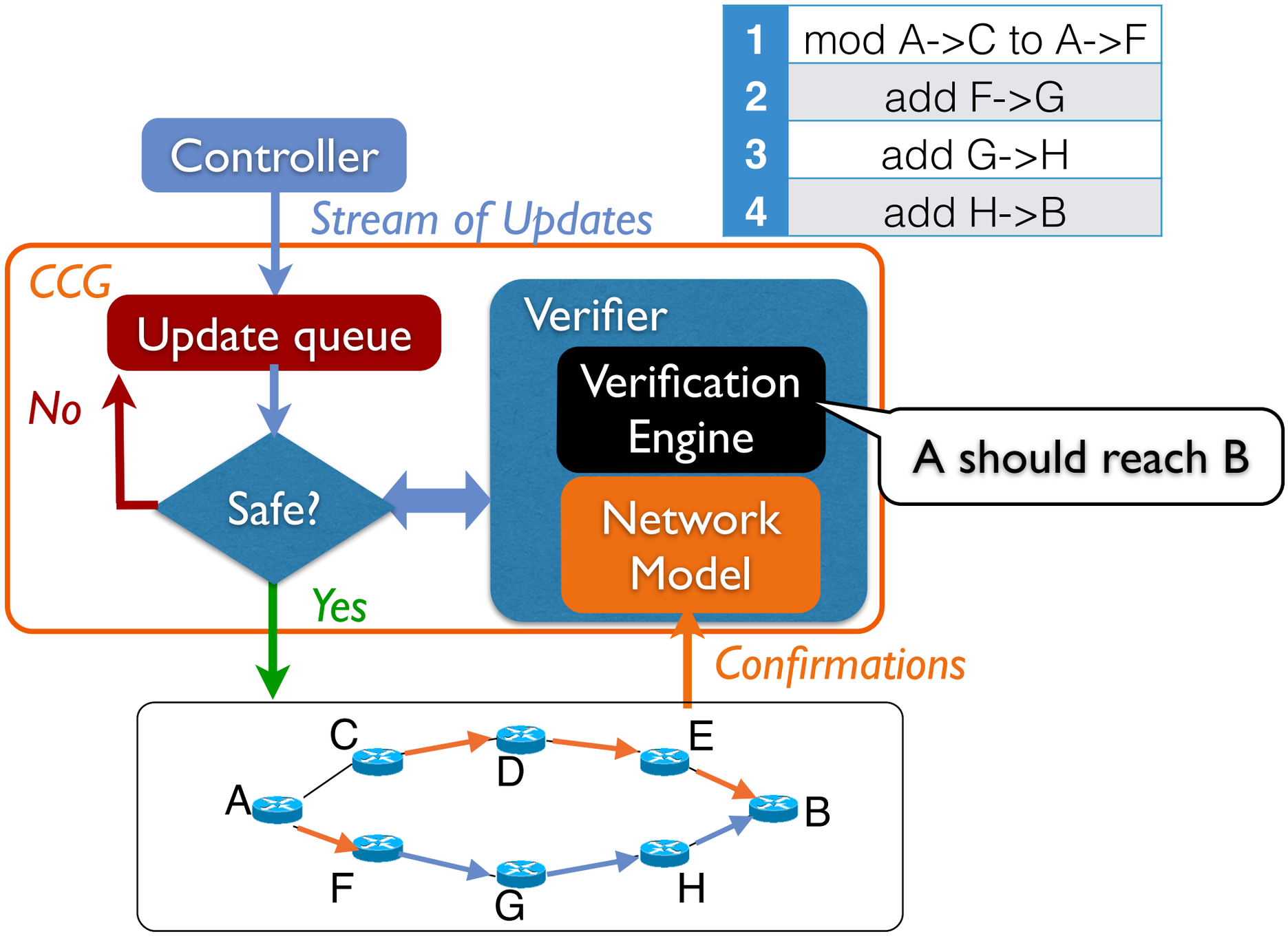
Our design: Customizable Consistency Generator



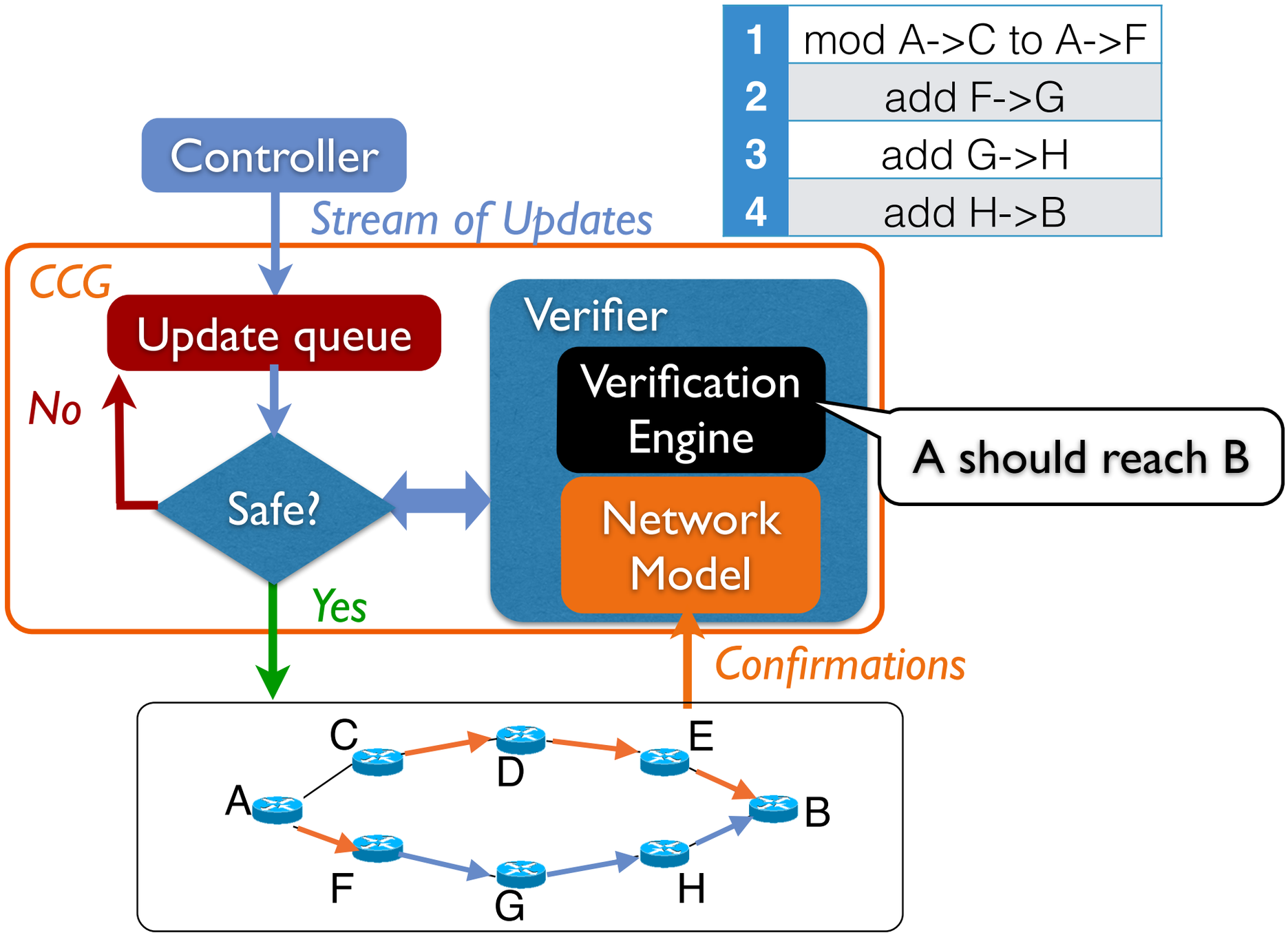
Our design: Customizable Consistency Generator



Our design: Customizable Consistency Generator

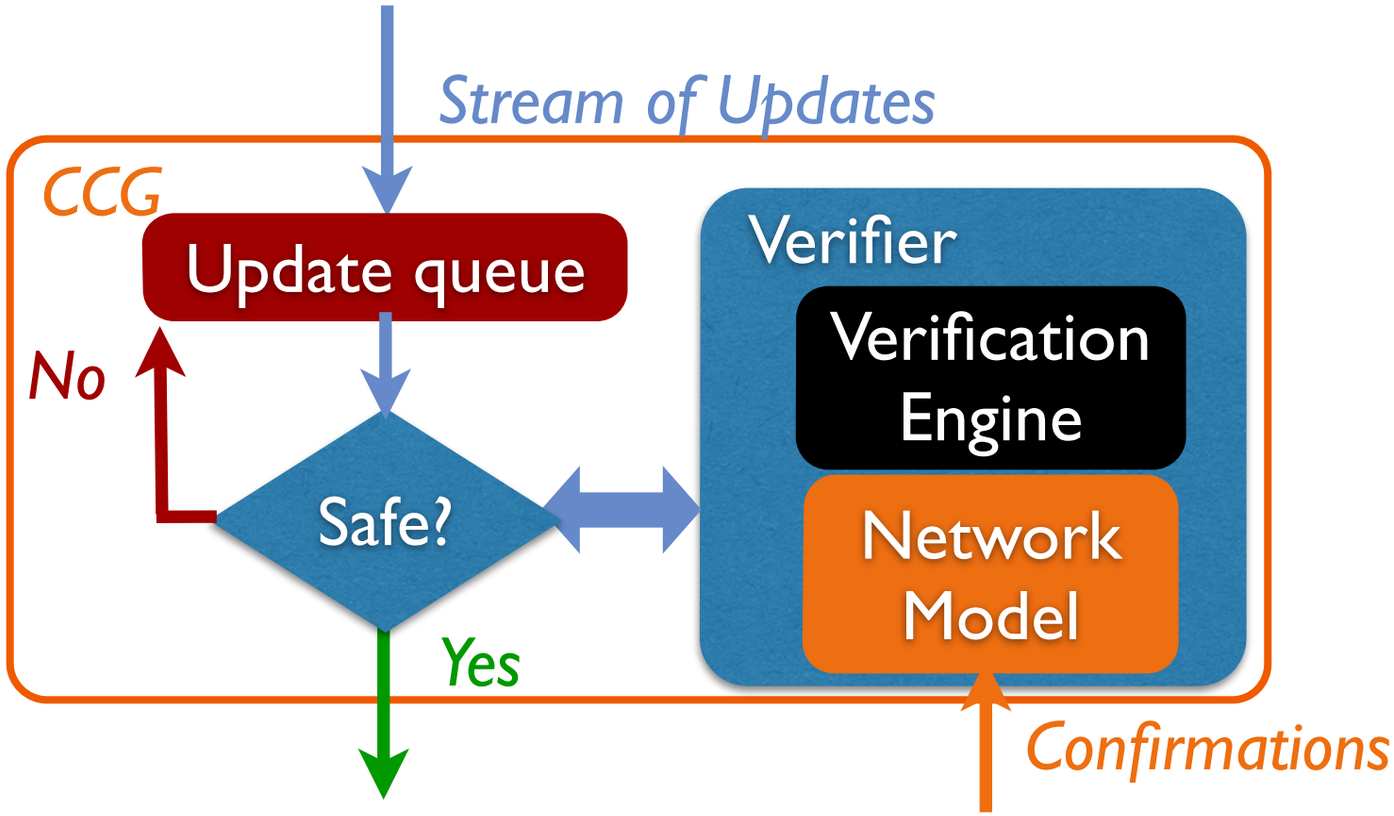


Our design: Customizable Consistency Generator

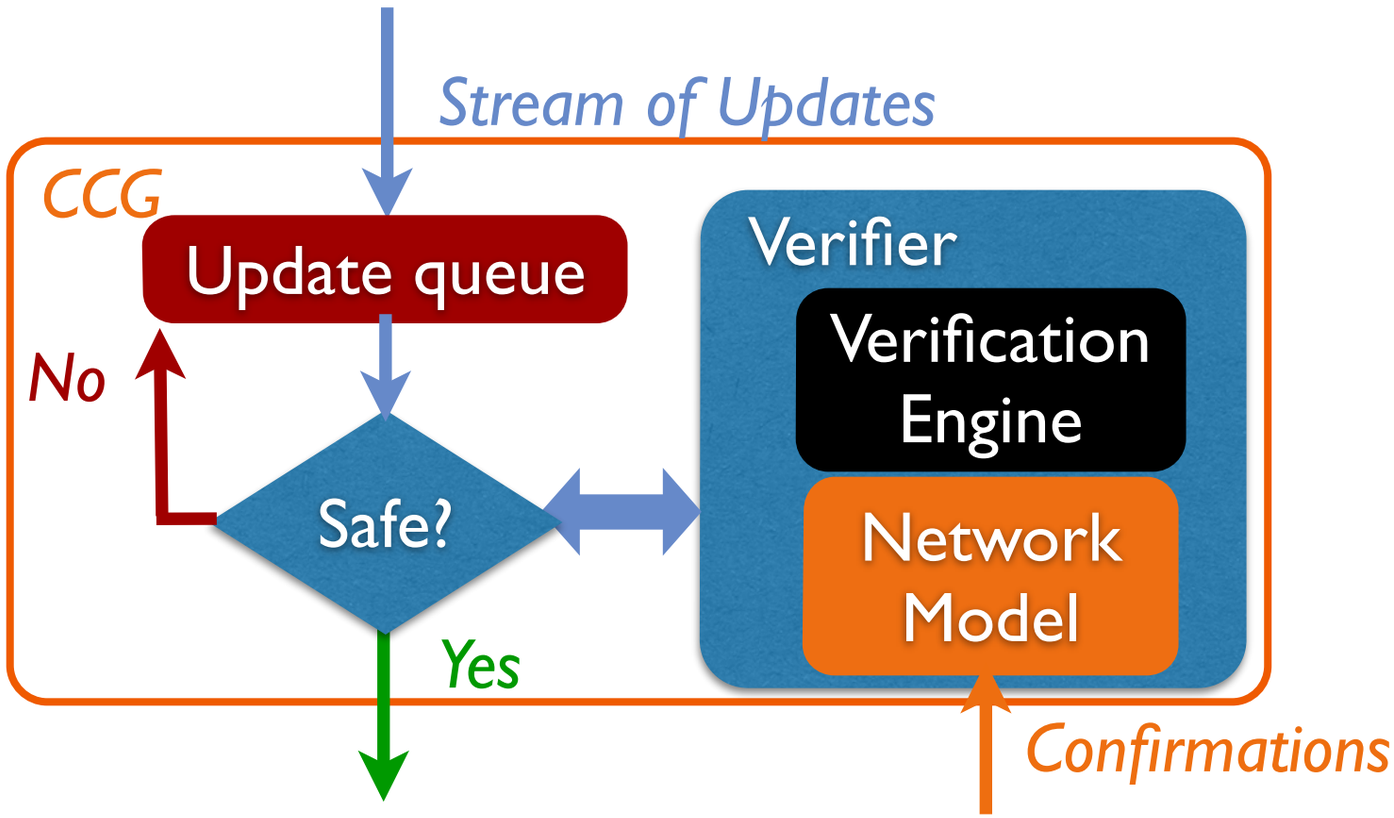


Enforcing consistency with heuristically maximized parallelism.

Challenges

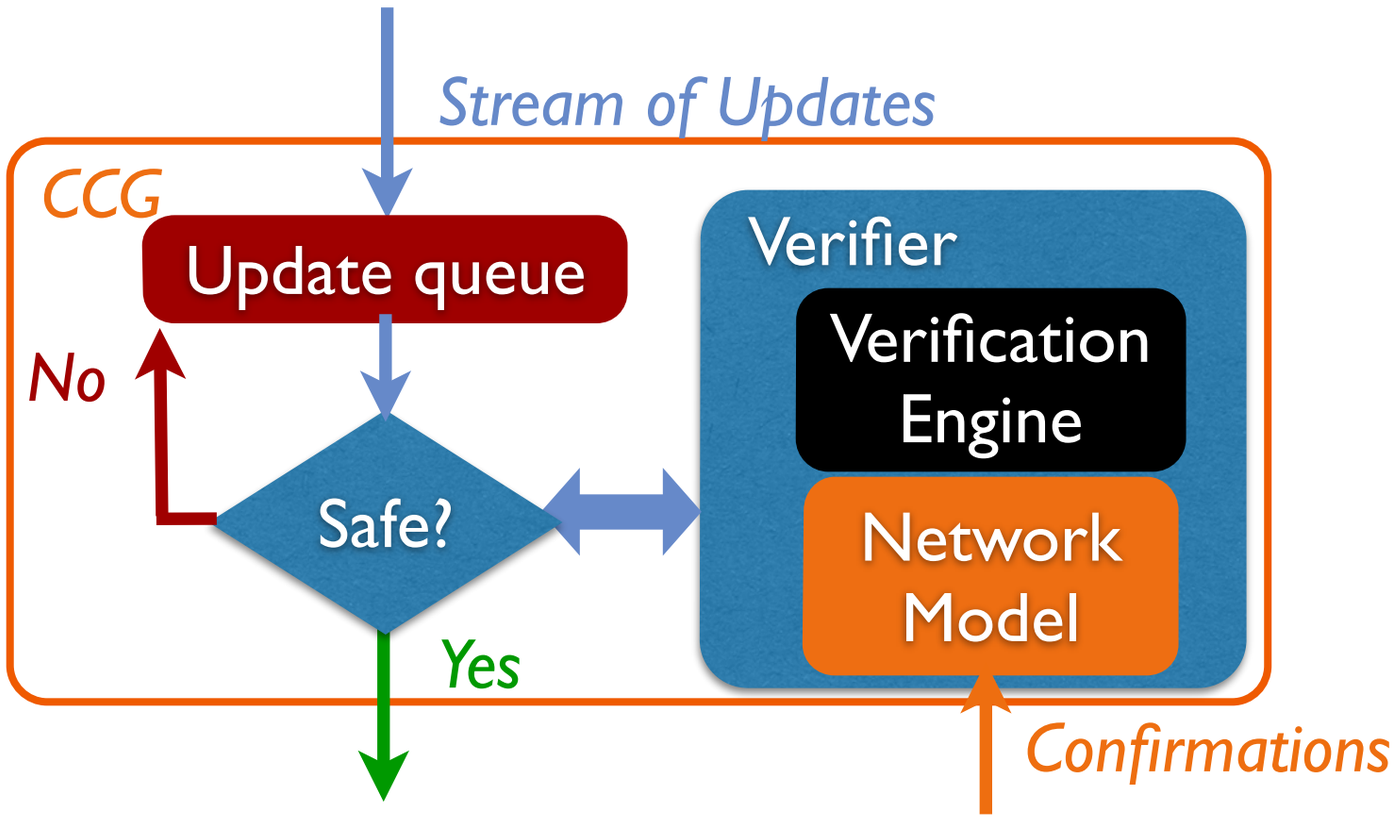

Stream of Updates
CCG
Update queue
No
Safe?
Yes
Verifier
Verification Engine
Network Model
Confirmations

Challenges



1. Distributed nature of networks (uncertainty)

Challenges



1. Distributed nature of networks (uncertainty)

2. Greedy algorithm may get stuck

Network Uncertainty

1

The “uncertainty” of an observation point tasked with instilling updates in knowing the current network state.

Network Uncertainty

1

The “uncertainty” of an observation point tasked with instilling updates in knowing the current network state.

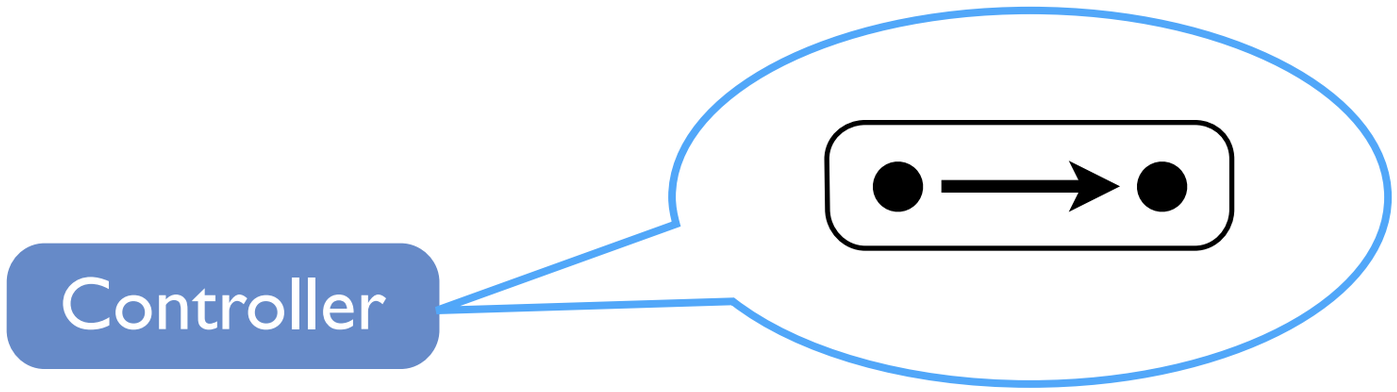
May deviate network behavior away from desired properties.

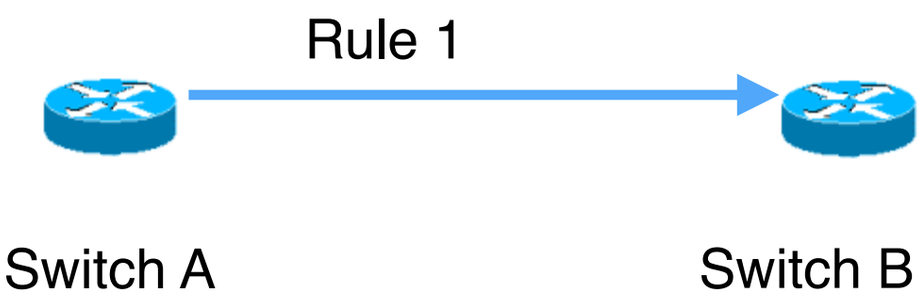
Network Uncertainty

1

The “uncertainty” of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.



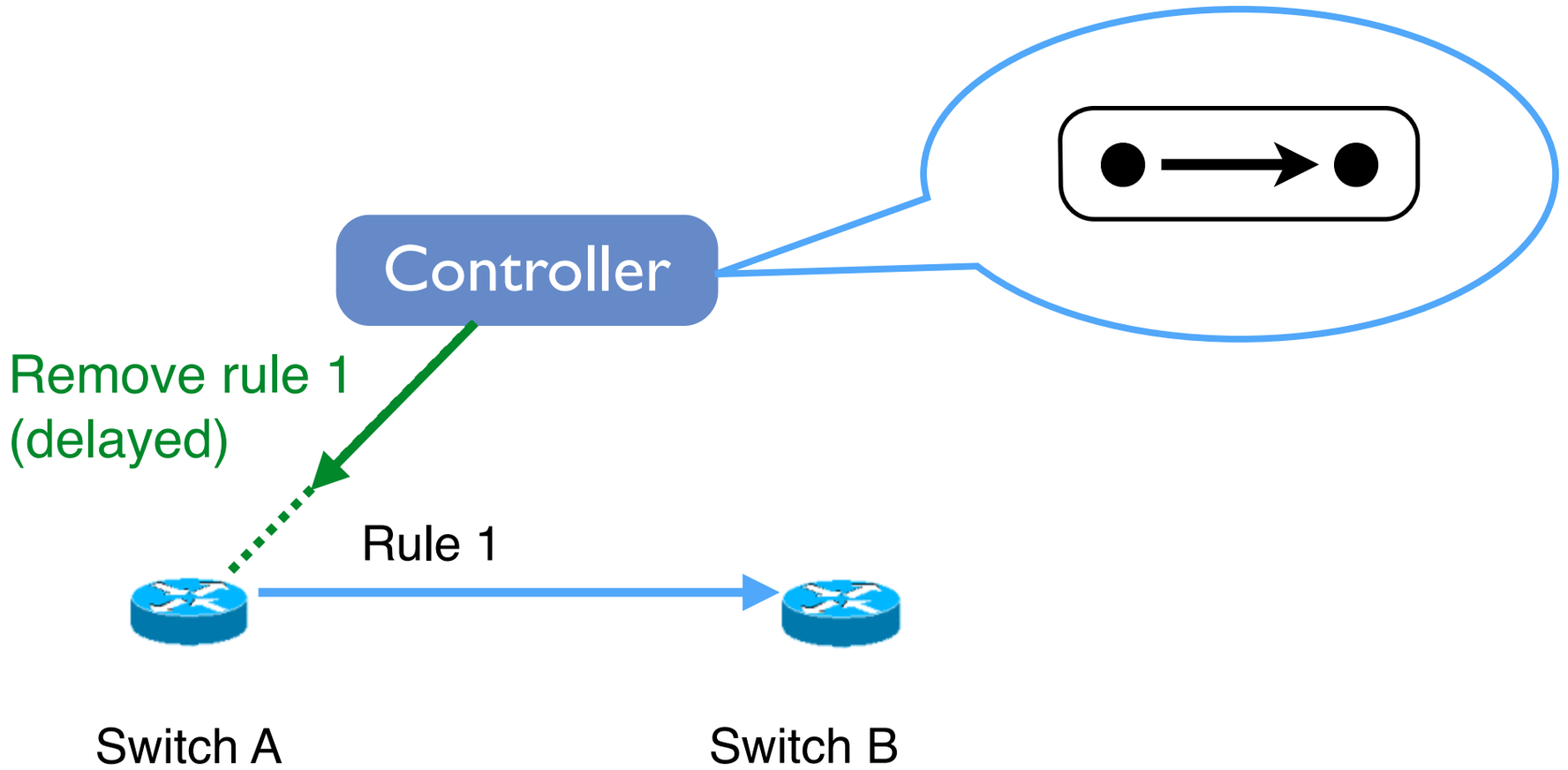


Network Uncertainty

1

The “uncertainty” of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.

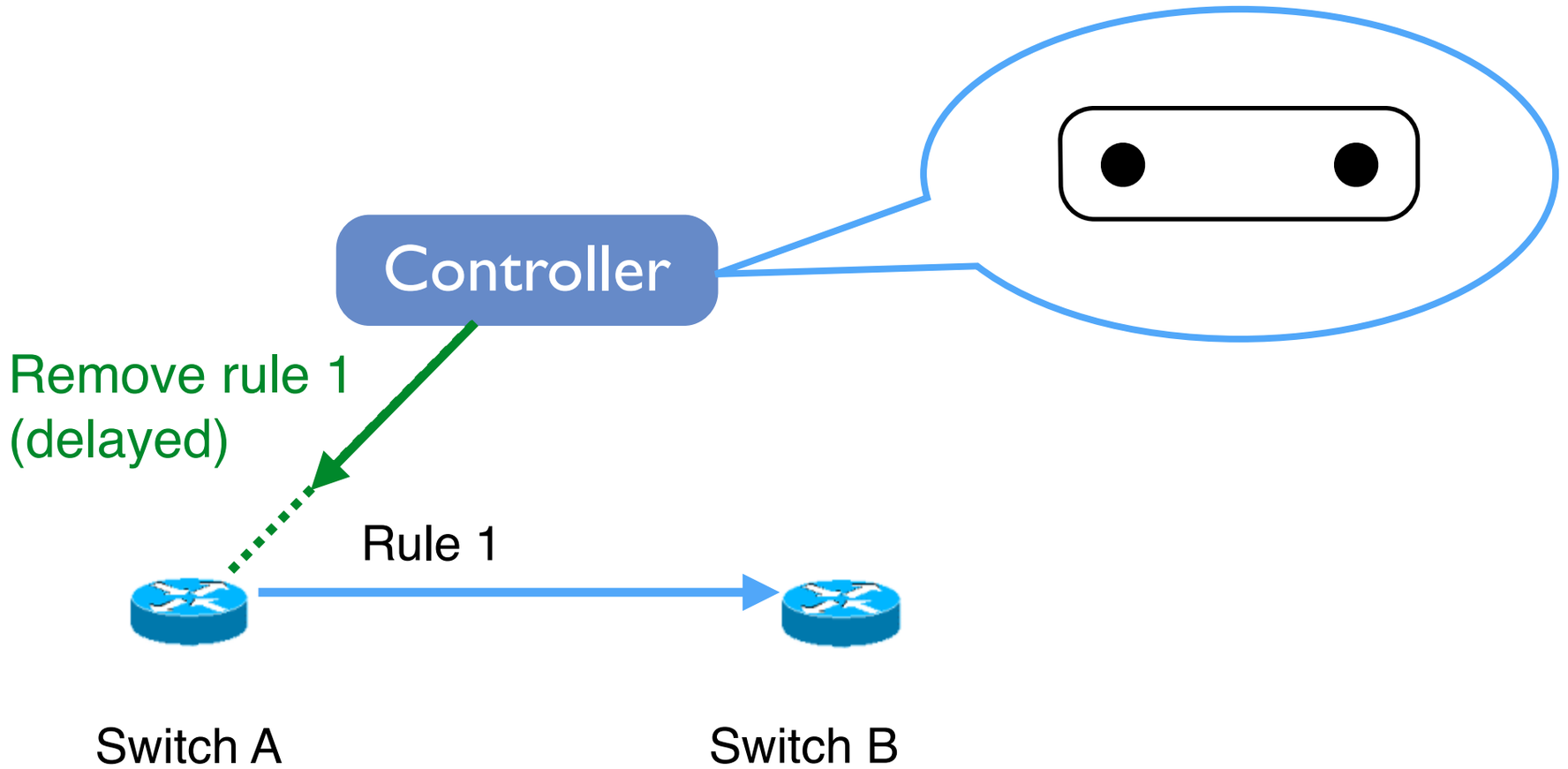


Network Uncertainty

1

The “uncertainty” of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.

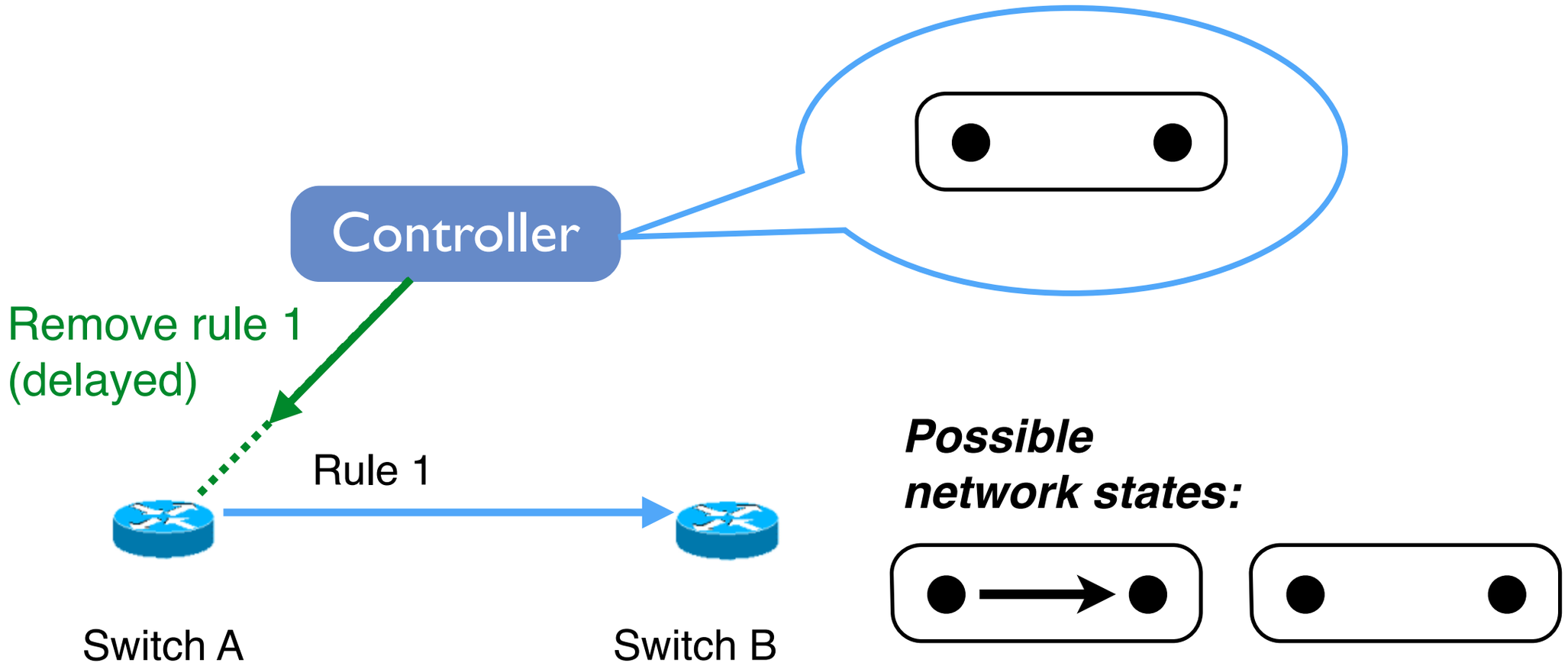


Network Uncertainty

1

The "uncertainty" of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.

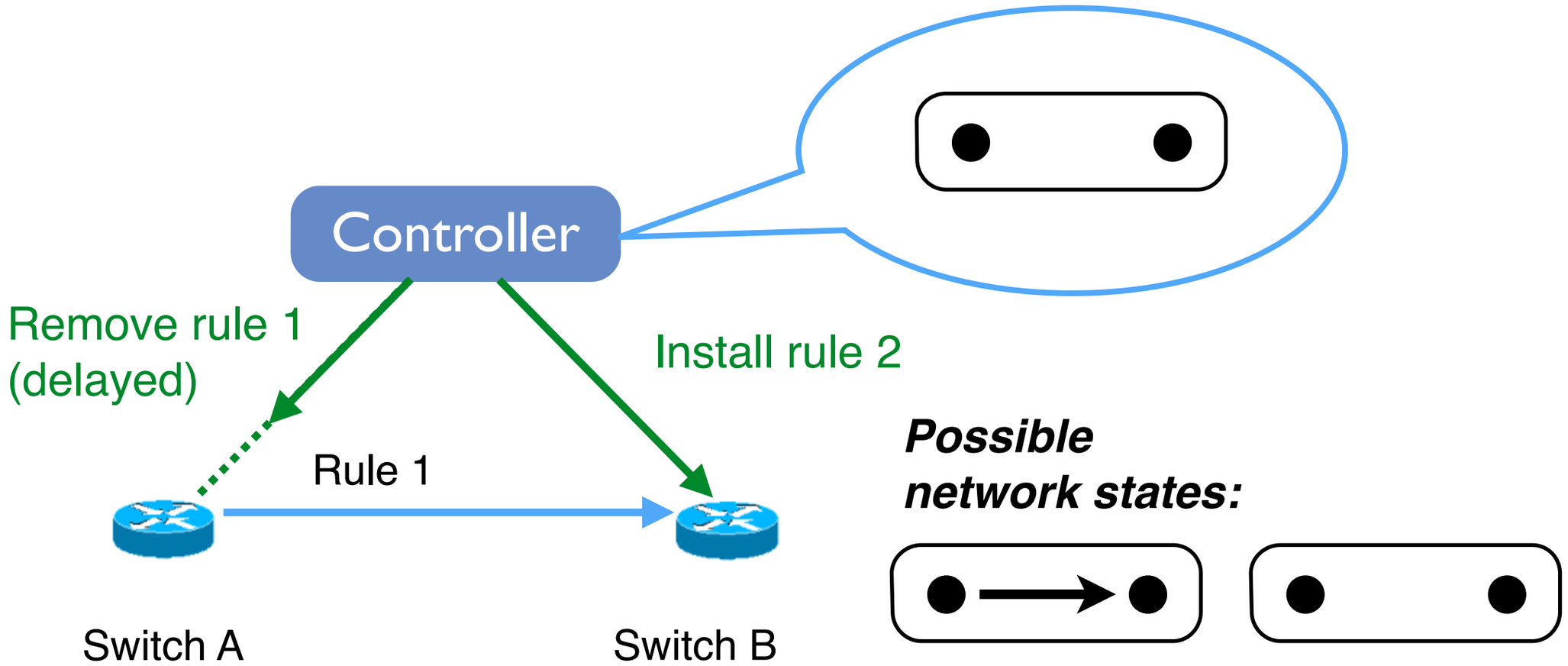


Network Uncertainty

1

The "uncertainty" of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.

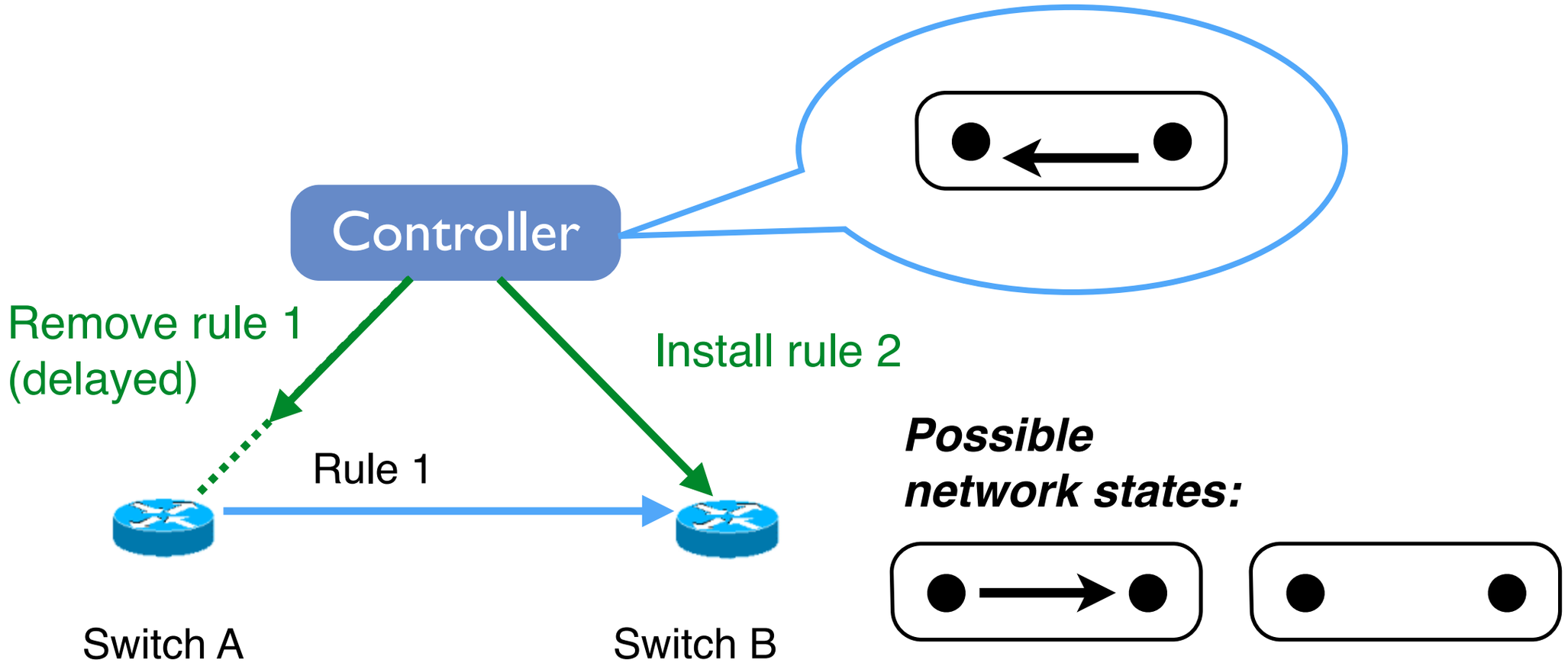


Network Uncertainty

1

The "uncertainty" of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.

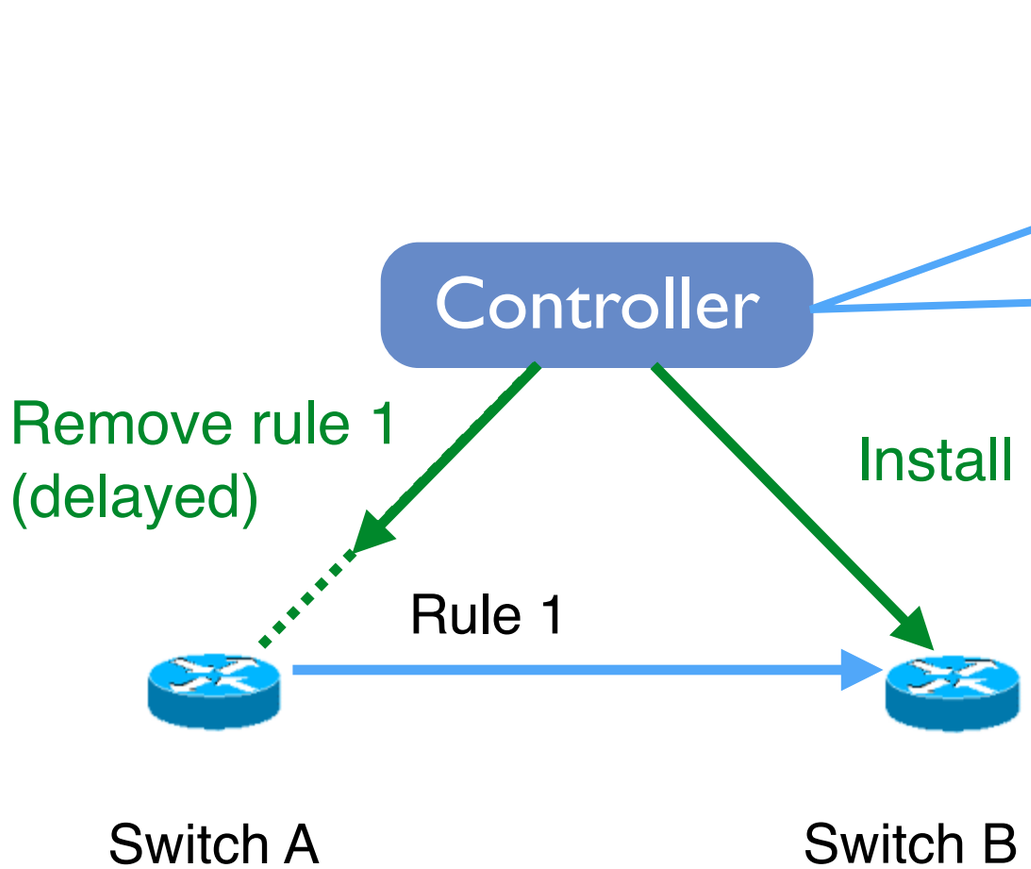


Network Uncertainty

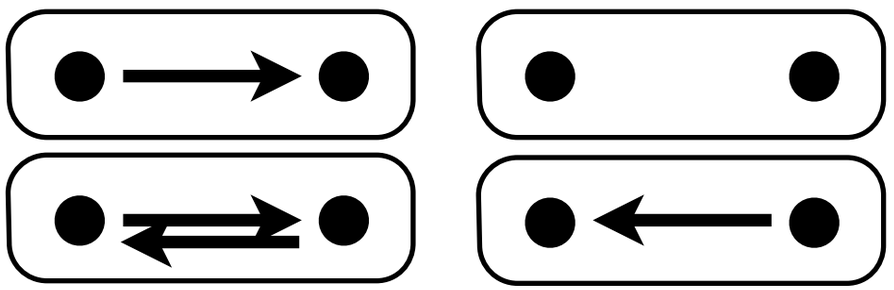
1

The "uncertainty" of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.



Possible network states:

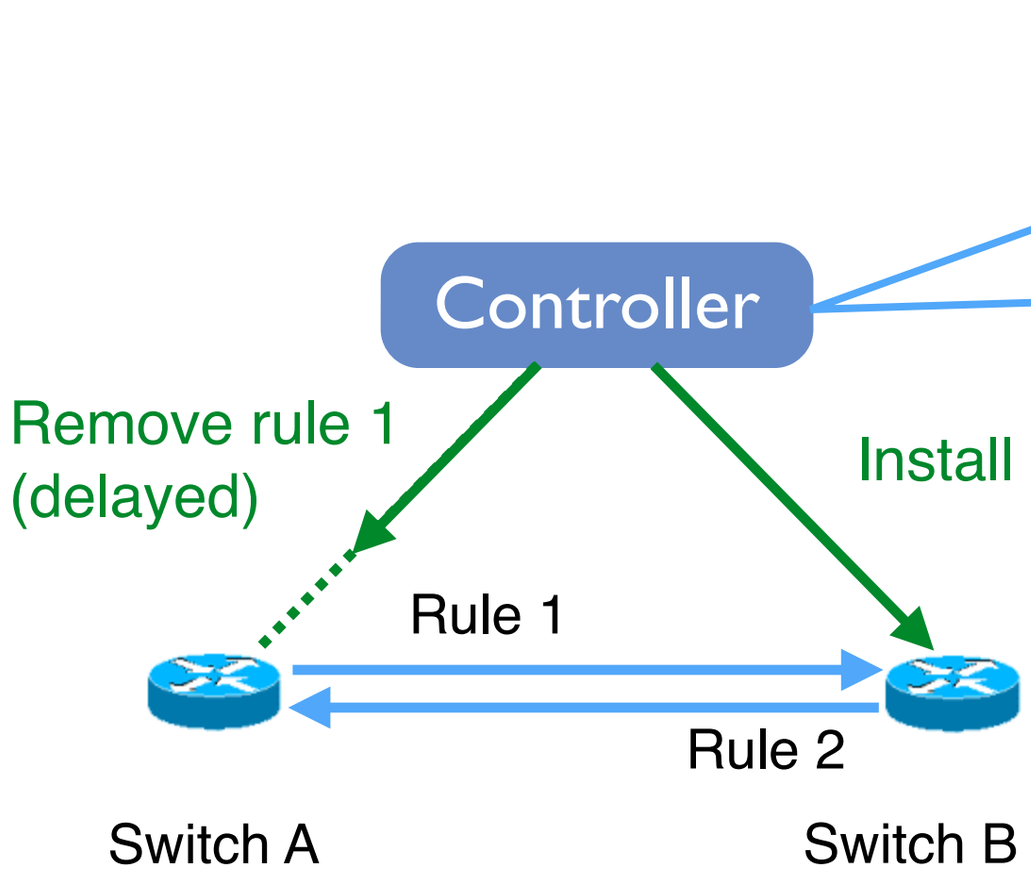


Network Uncertainty

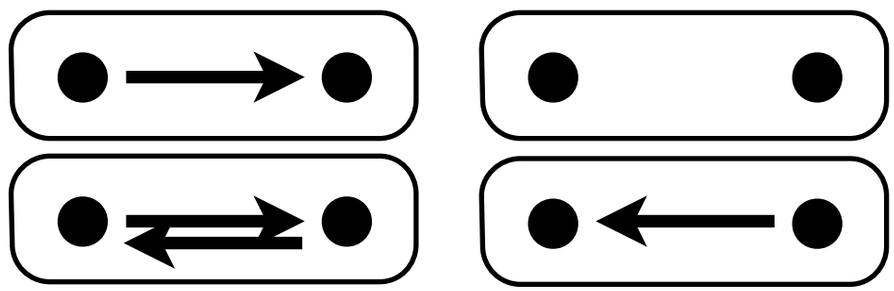
1

The "uncertainty" of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.



Possible network states:

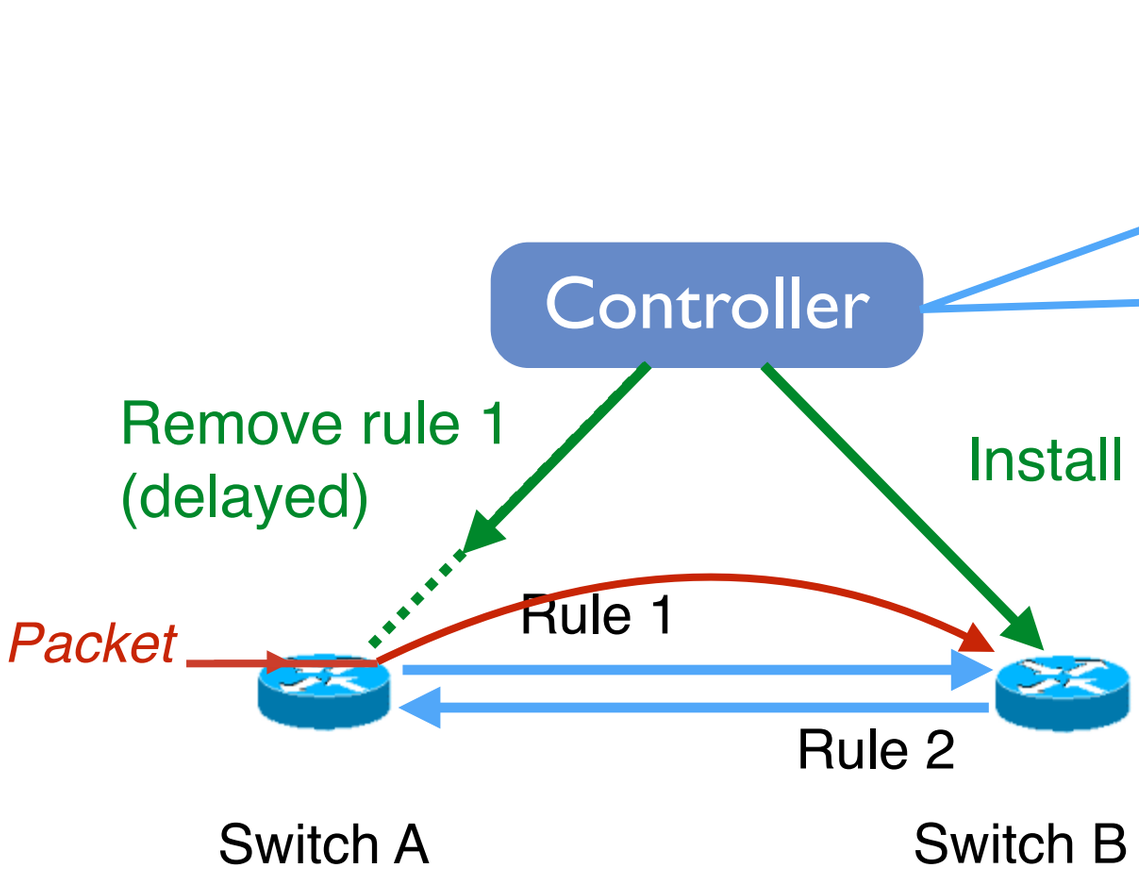


Network Uncertainty

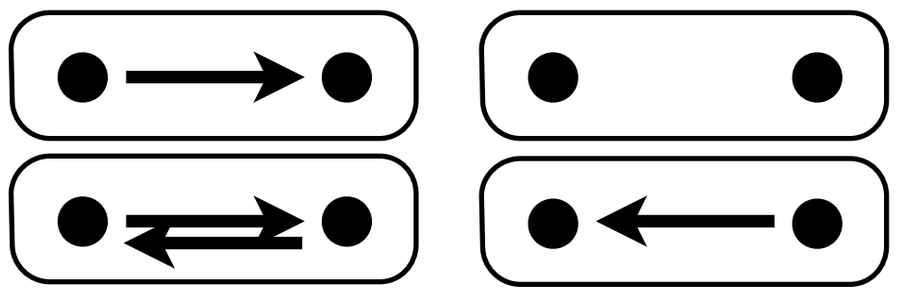
1

The "uncertainty" of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.



Possible network states:

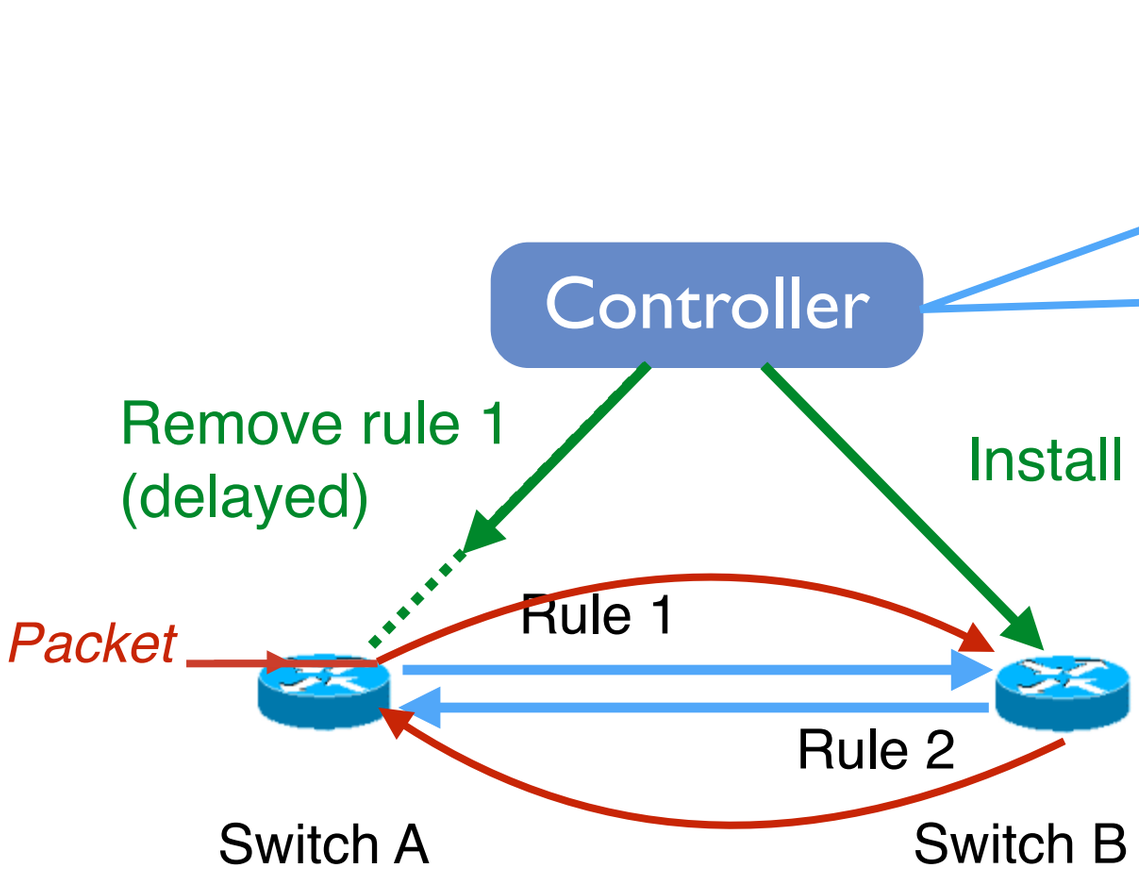


Network Uncertainty

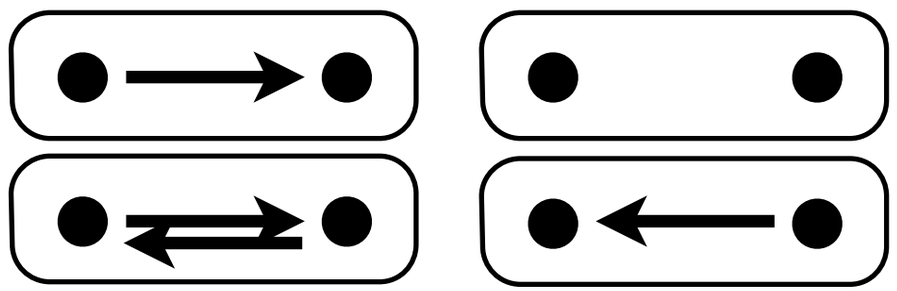
1

The "uncertainty" of an observation point tasked with instilling updates in knowing the current network state.

May deviate network behavior away from desired properties.



Possible network states:



Uncertainty-aware Modeling

1

Network states as forwarding graphs

Uncertainty-aware Modeling

1

Network states as forwarding graphs

Naively, represent every possible network state O(2^n)

Uncertainty-aware Modeling

1

Network states as forwarding graphs

Naively, represent every possible network state O(2^n)

Uncertain graph: Collapse all possible states onto one graph

Uncertainty-aware Modeling

1

Network states as forwarding graphs

Naively, represent every possible network state O(2^n)

Uncertain graph: Collapse all possible states onto one graph

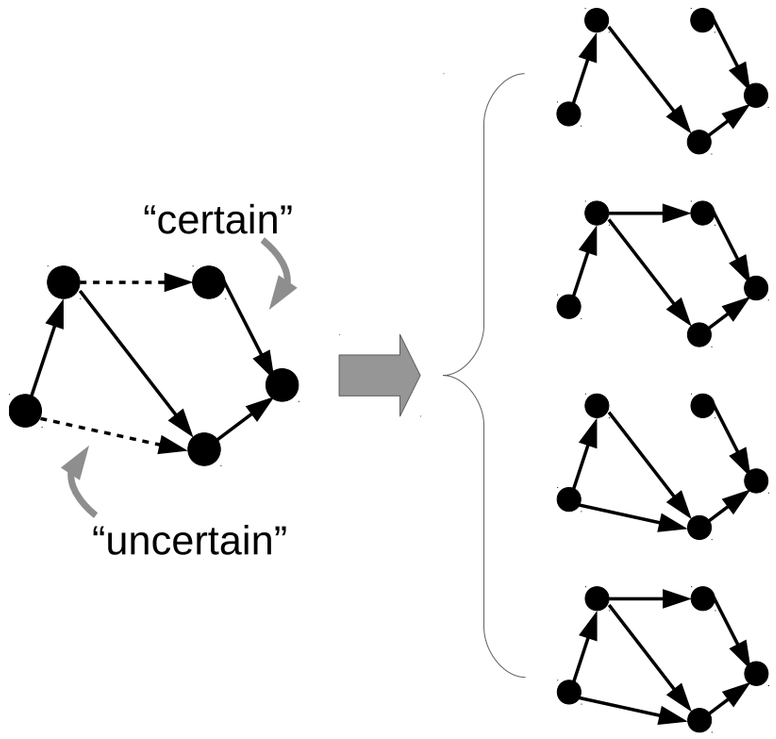
Uncertainty-aware Modeling

1

Network states as forwarding graphs

Naively, represent every possible network state O(2^n)

Uncertain graph: Collapse all possible states onto one graph



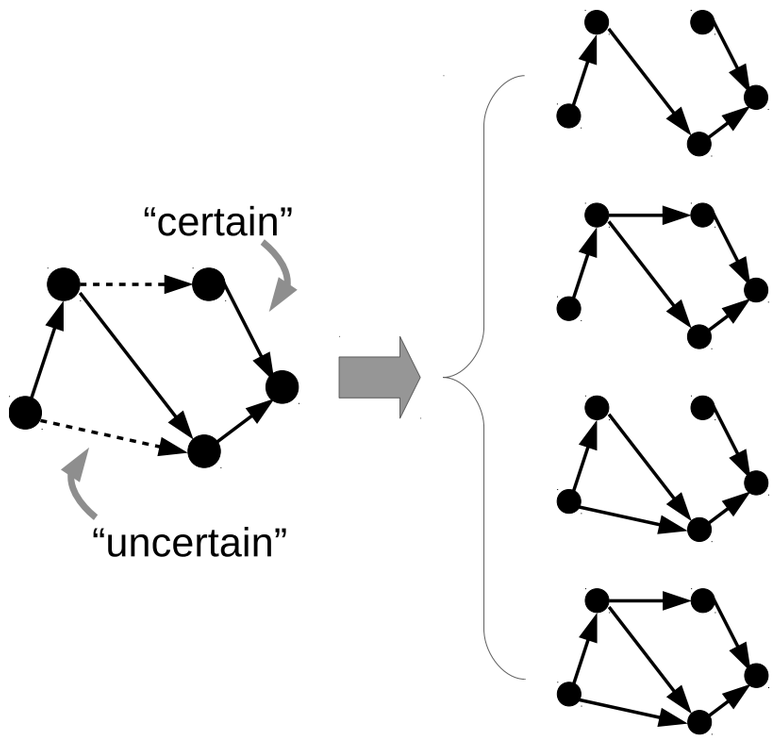
Uncertainty-aware Modeling

1

Network states as forwarding graphs

Naively, represent every possible network state O(2^n)

Uncertain graph: Collapse all possible states onto one graph



When to change “uncertain” to “certain”?

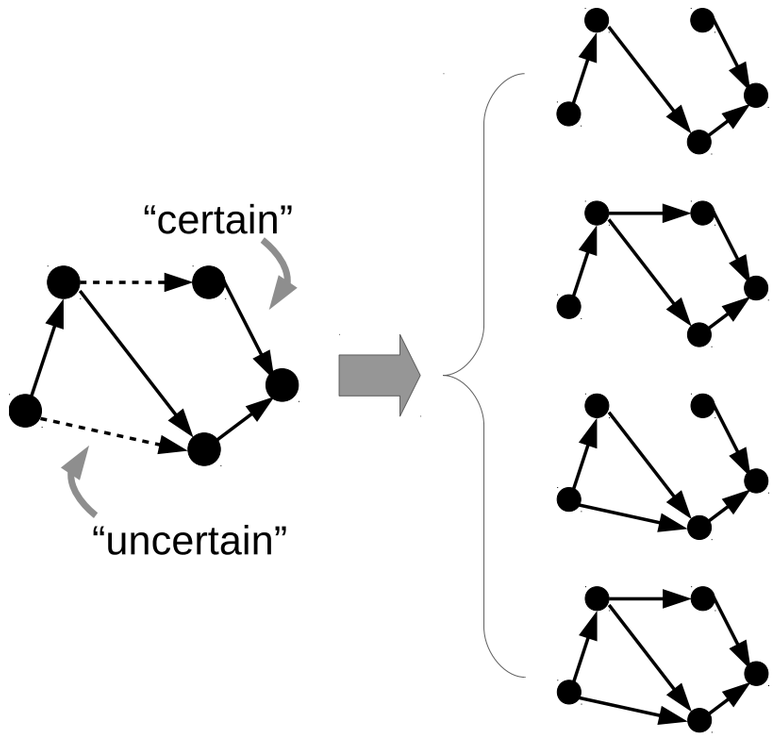
Uncertainty-aware Modeling

1

Network states as forwarding graphs

Naively, represent every possible network state O(2^n)

Uncertain graph: Collapse all possible states onto one graph



When to change “uncertain” to “certain”?

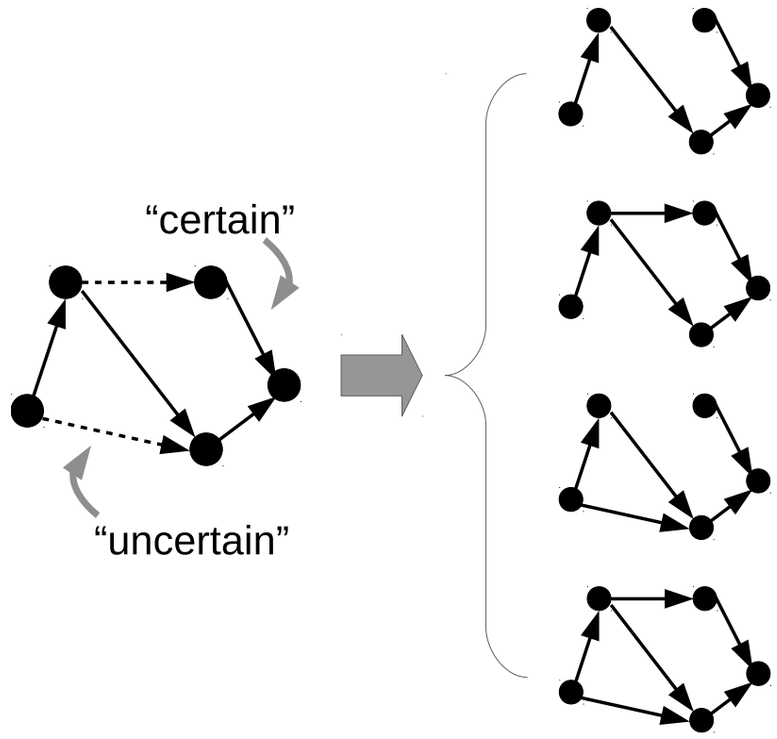
Uncertainty-aware Modeling

1

Network states as forwarding graphs

Naively, represent every possible network state O(2^n)

Uncertain graph: Collapse all possible states onto one graph



The model captures packets' view of the network, assuming controller initiates changes.

When to change "uncertain" to "certain"?

Verification under Uncertainty

1

Verification: Traversal on forwarding graphs

Verification under Uncertainty

1

Verification: Traversal on forwarding graphs

Take uncertainty into account:

Verification under Uncertainty

1

Verification: Traversal on forwarding graphs

Take uncertainty into account:

- No certain outgoing link → a possible black-hole

Verification under Uncertainty

1

Verification: Traversal on forwarding graphs

Take uncertainty into account:

- No certain outgoing link → a possible black-hole

By traversing the graph once,

Verification under Uncertainty

1

Verification: Traversal on forwarding graphs

Take uncertainty into account:

- No certain outgoing link → a possible black-hole

By traversing the graph once,

- perform a simultaneous traversal of all possibilities, and thus

Verification under Uncertainty

1

Verification: Traversal on forwarding graphs

Take uncertainty into account:

- No certain outgoing link → a possible black-hole

By traversing the graph once,

- perform a simultaneous traversal of all possibilities, and thus
- reason about the network state correctly in the presence of uncertainty

Verification under Uncertainty

1

Verification: Traversal on forwarding graphs

Take uncertainty into account:

- No certain outgoing link → a possible black-hole

By traversing the graph once,

- perform a simultaneous traversal of all possibilities, and thus
- reason about the network state correctly in the presence of uncertainty

Implementation Optimizations

Verification under Uncertainty

1

Verification: Traversal on forwarding graphs

Take uncertainty into account:

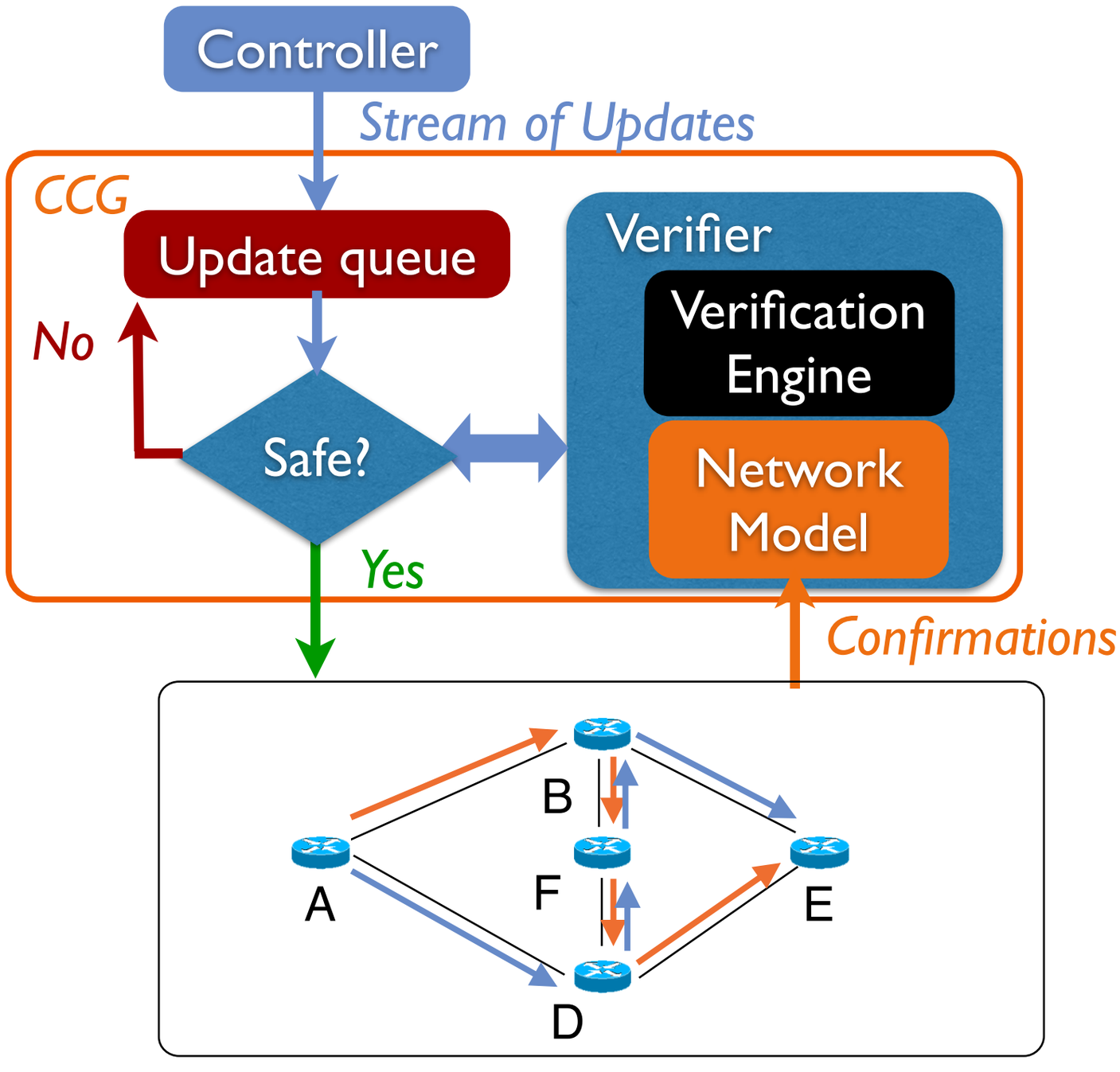
- No certain outgoing link → a possible black-hole

By traversing the graph once,

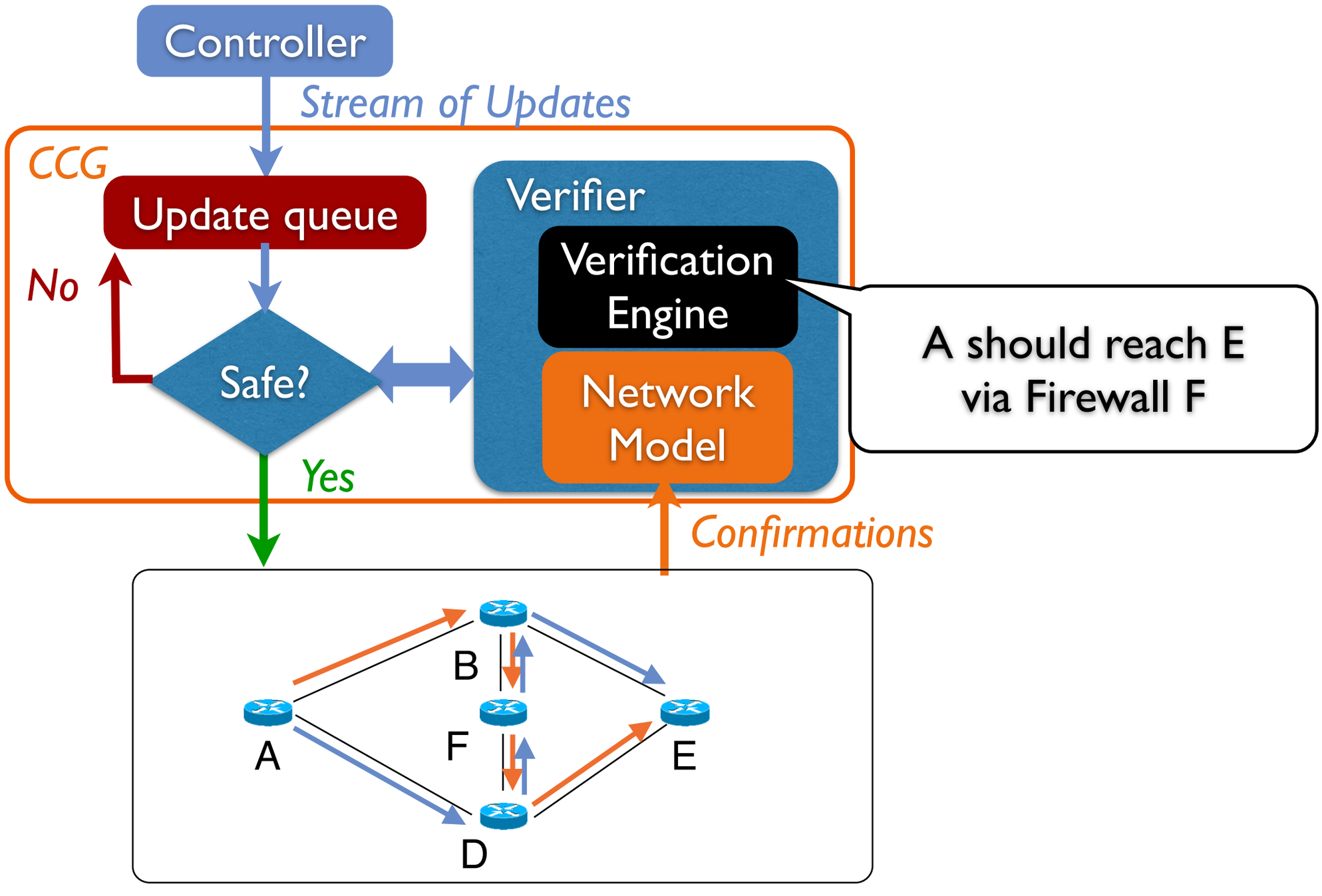
- perform a simultaneous traversal of all possibilities, and thus
- reason about the network state correctly in the presence of uncertainty

Implementation Optimizations

Consistency under Uncertainty — Problem

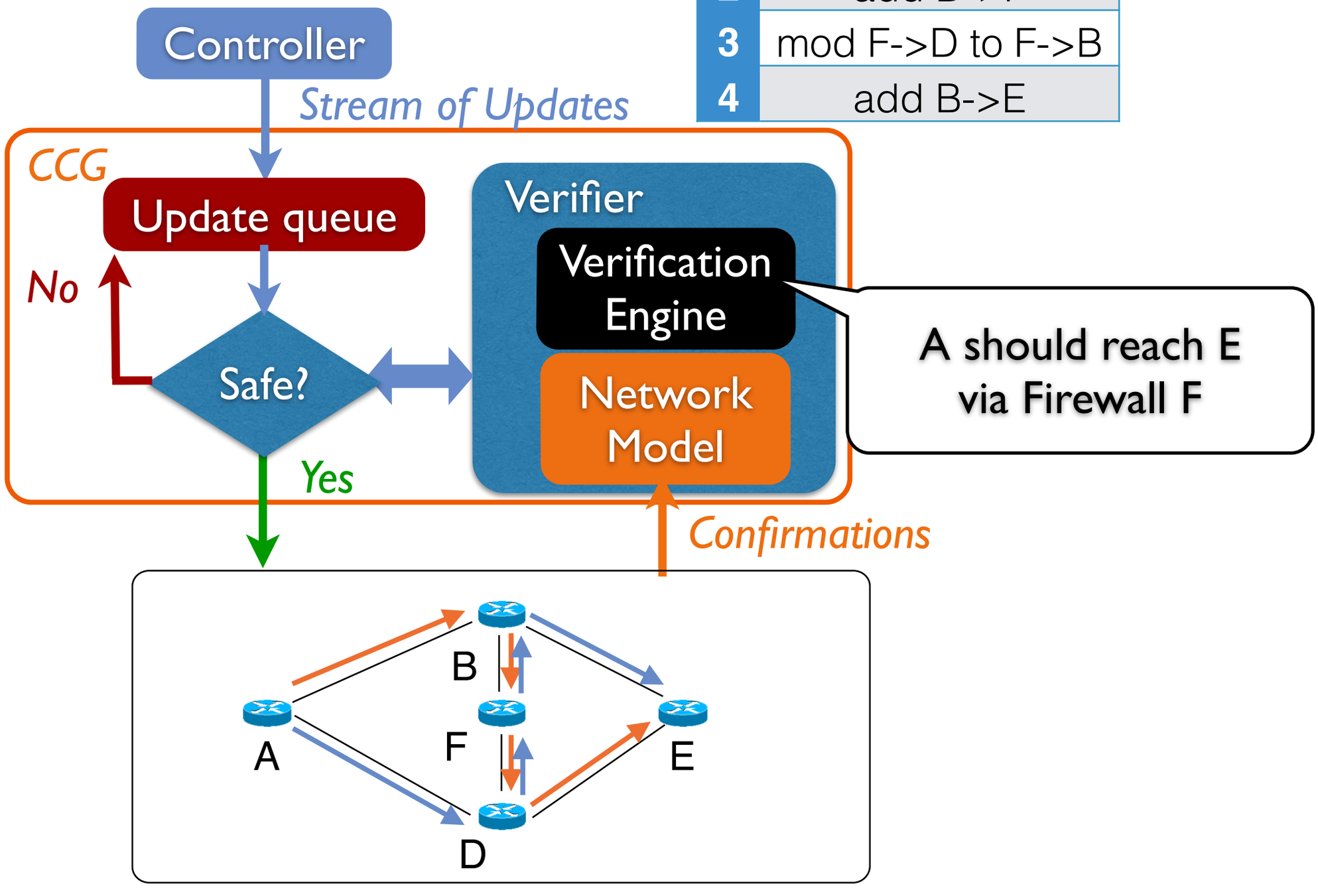


Consistency under Uncertainty — Problem



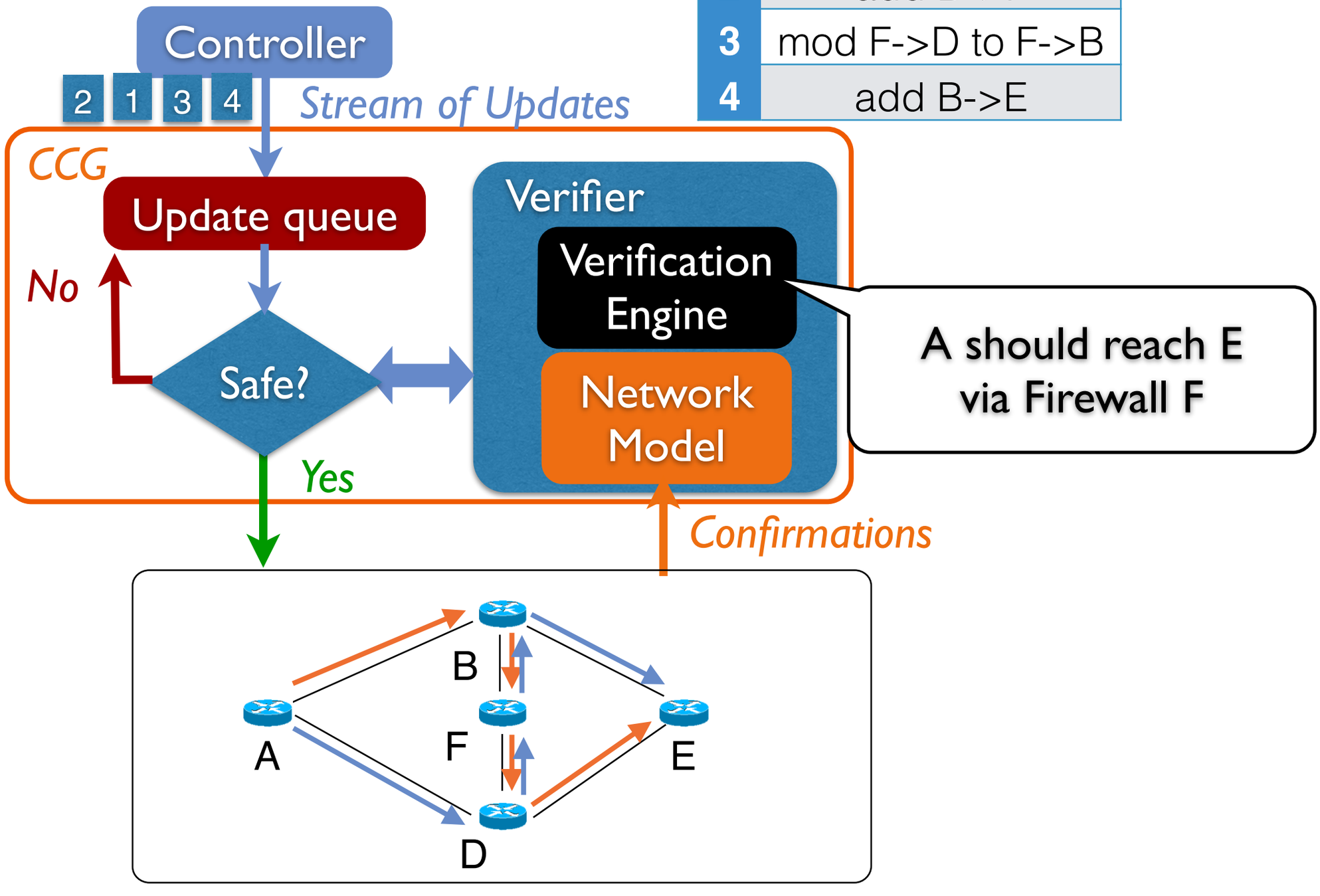
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



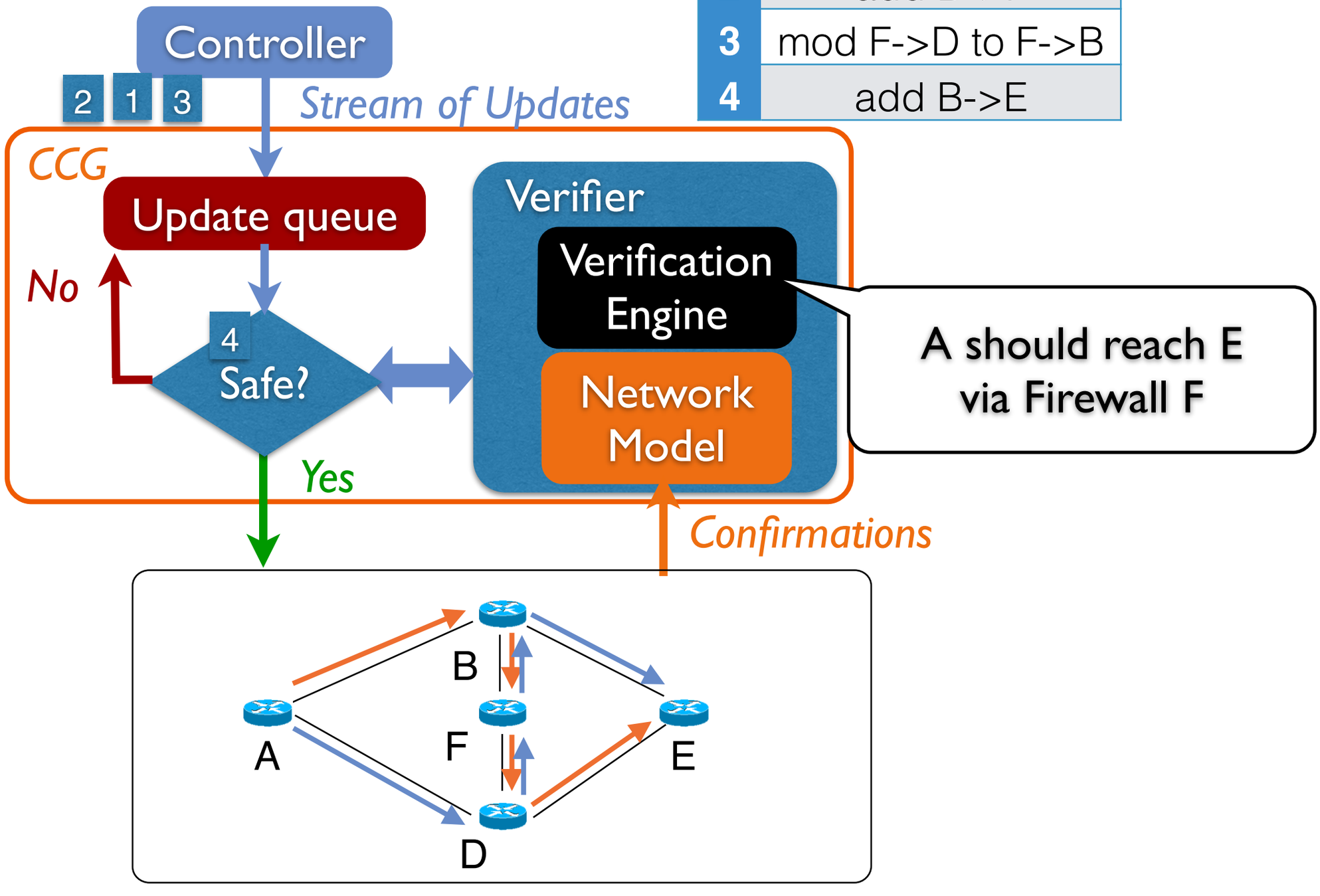
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



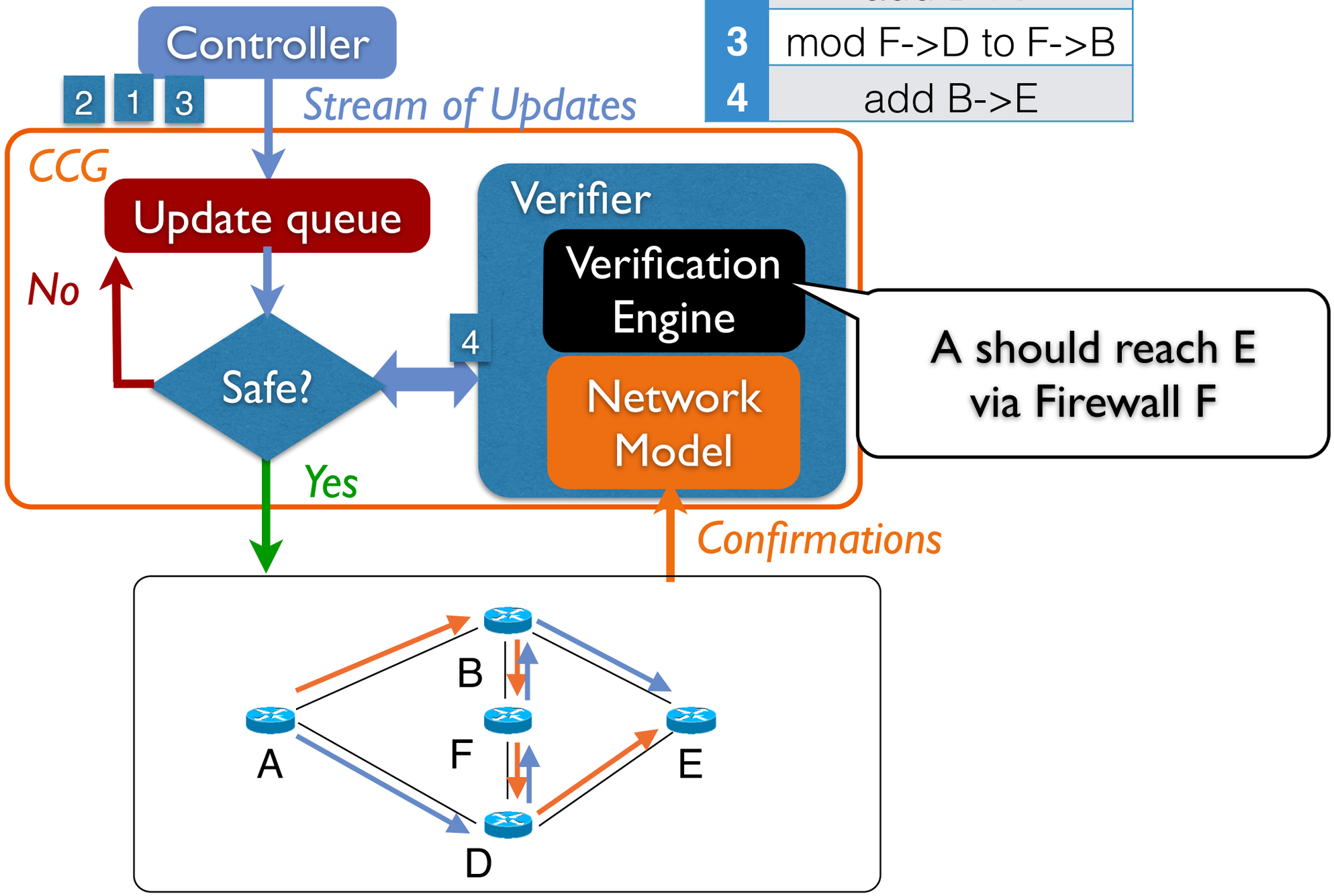
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



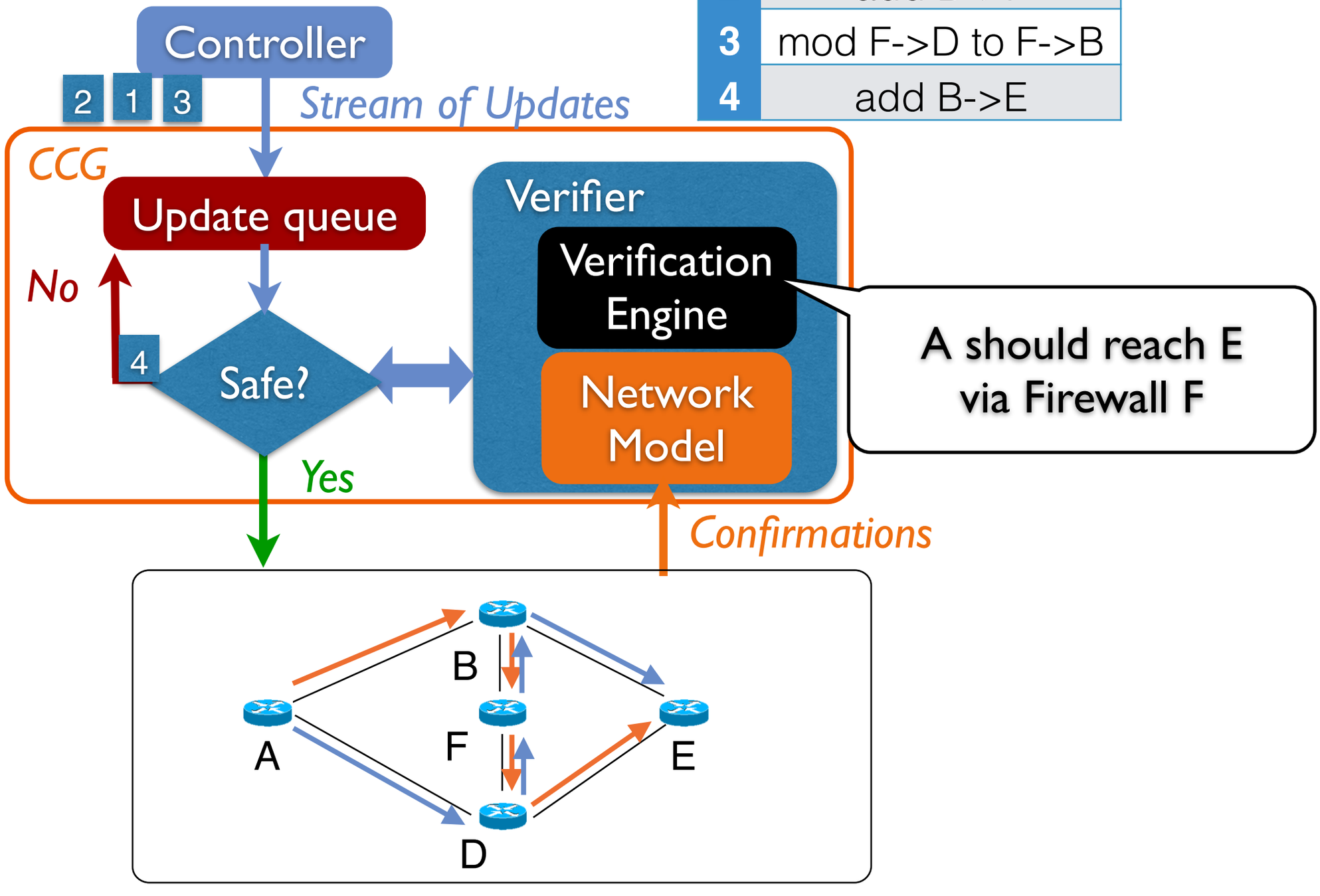
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



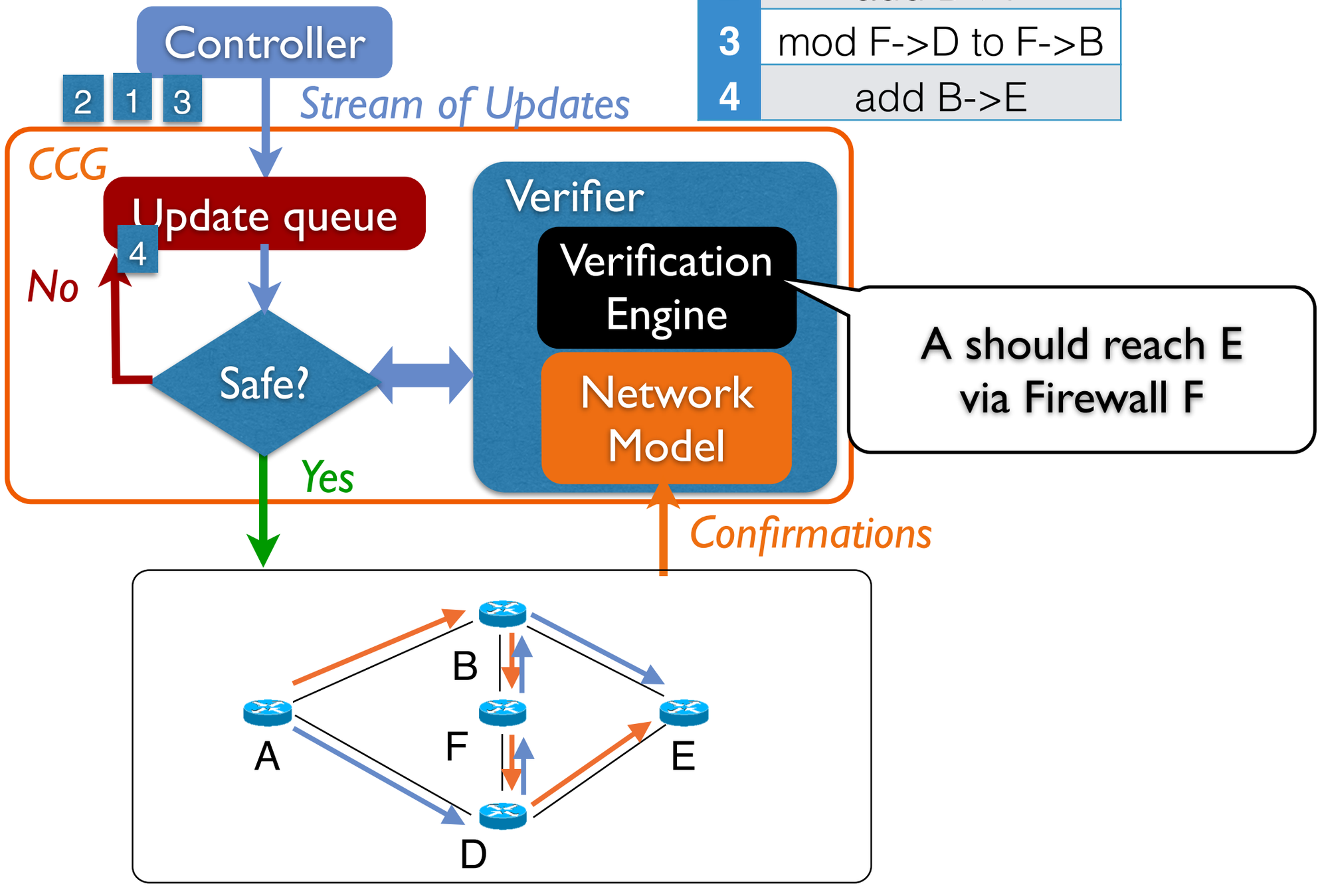
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



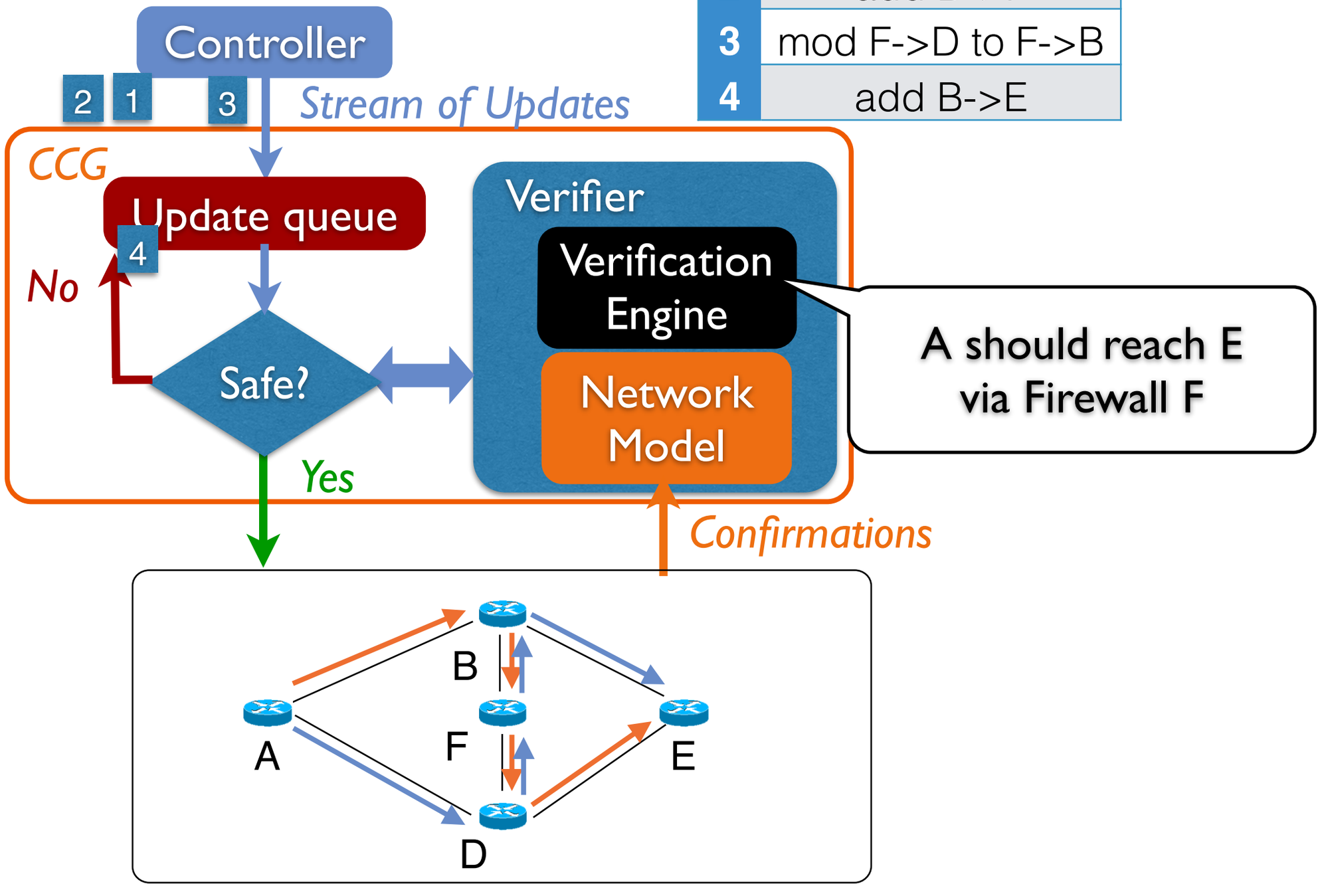
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



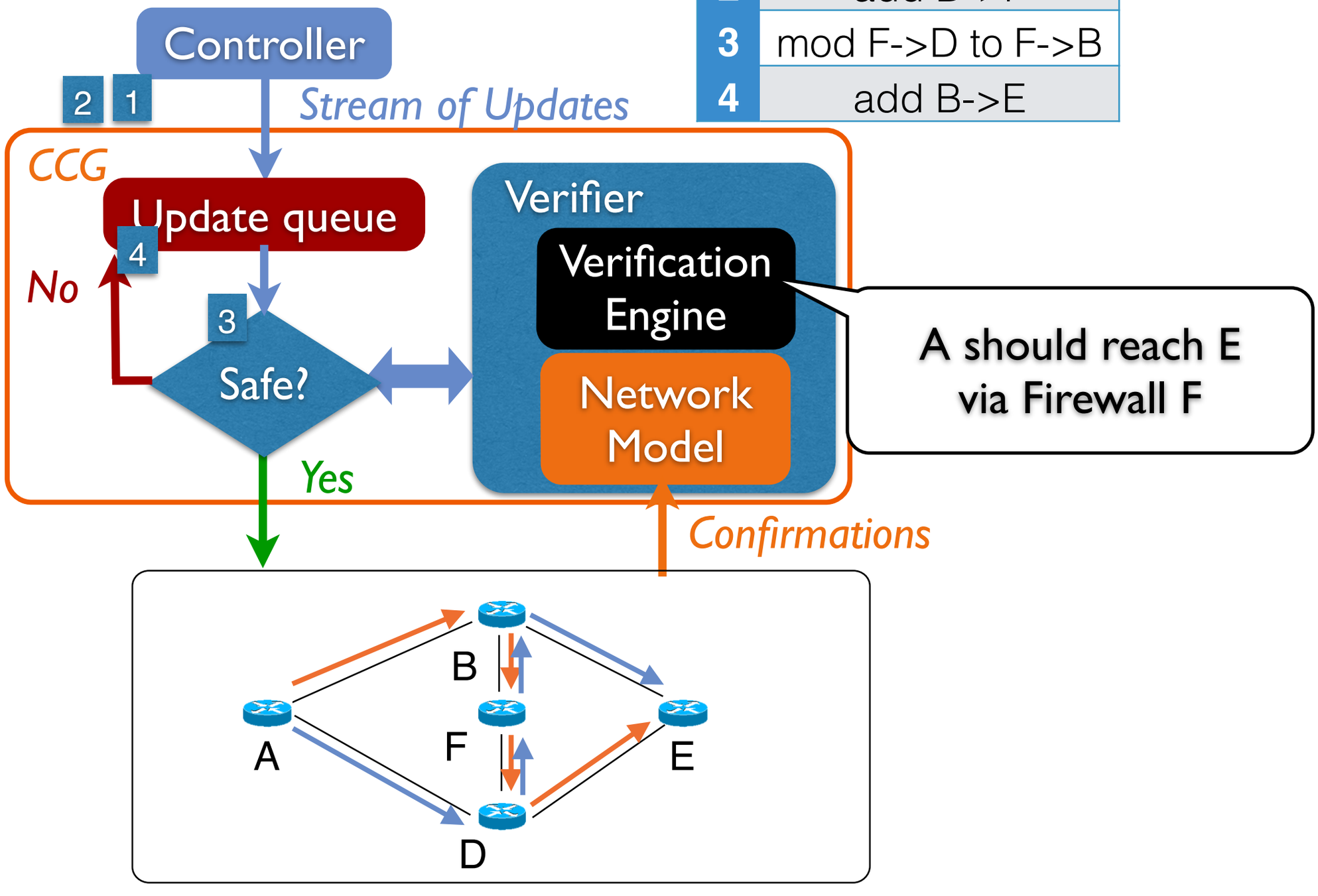
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



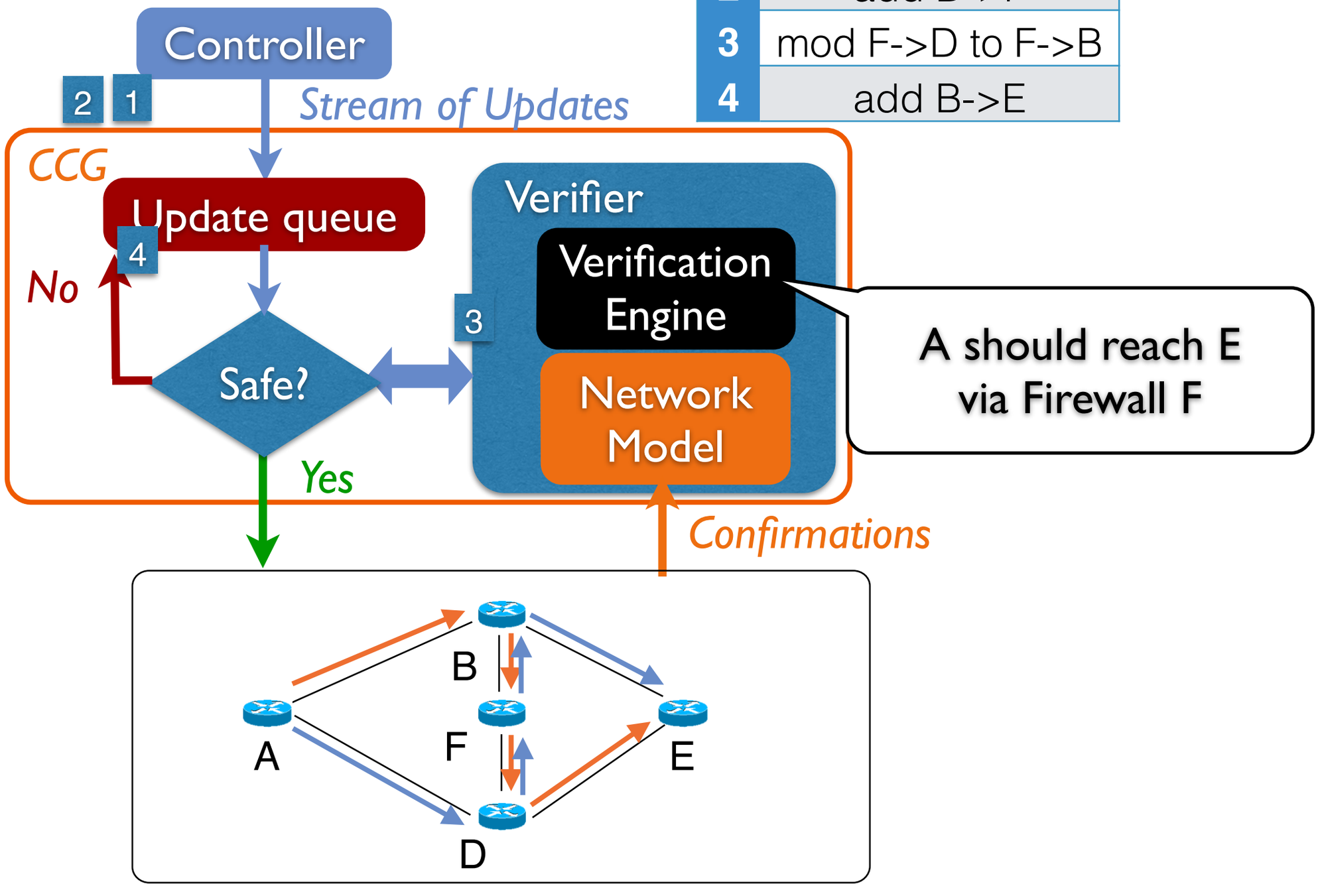
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



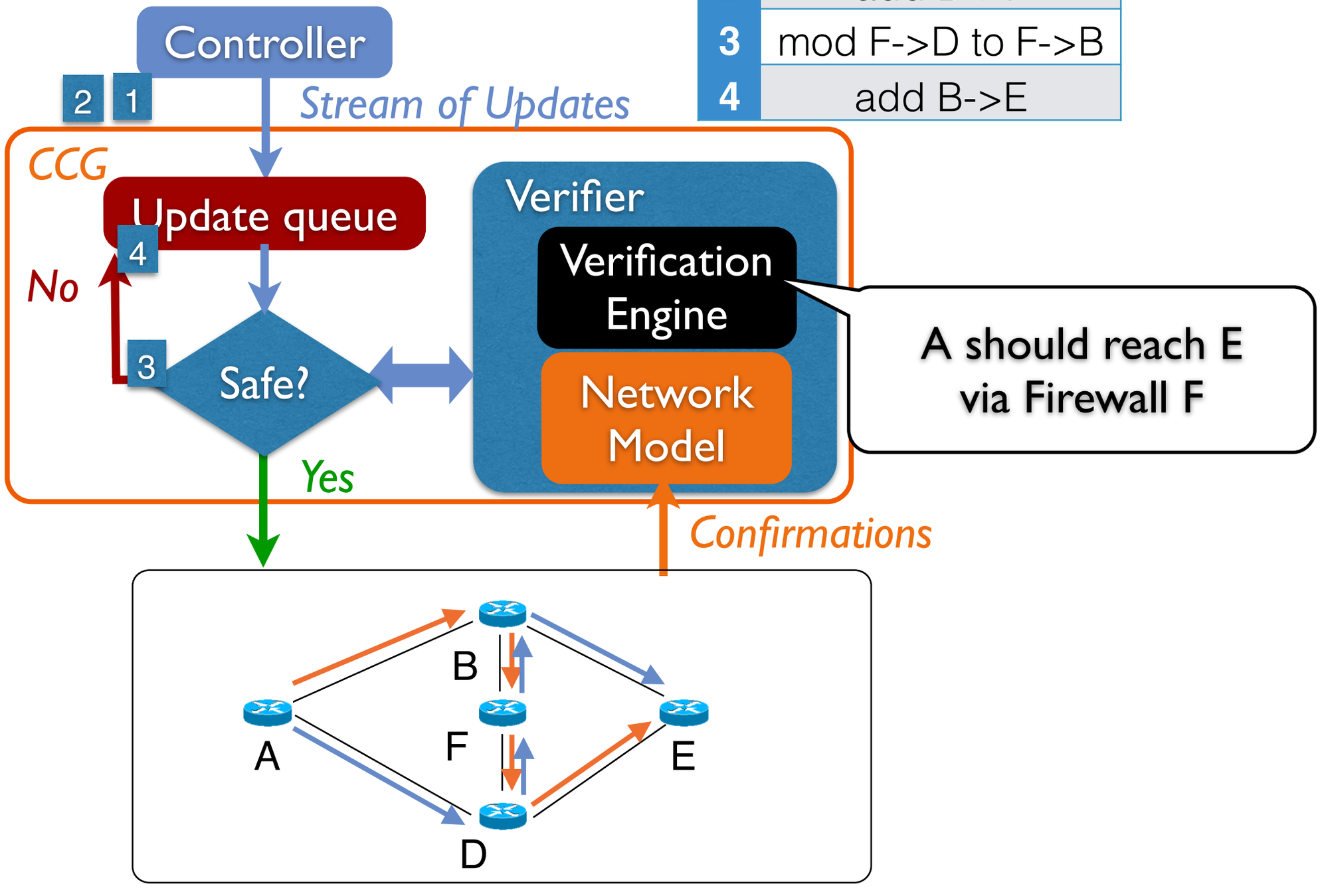
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



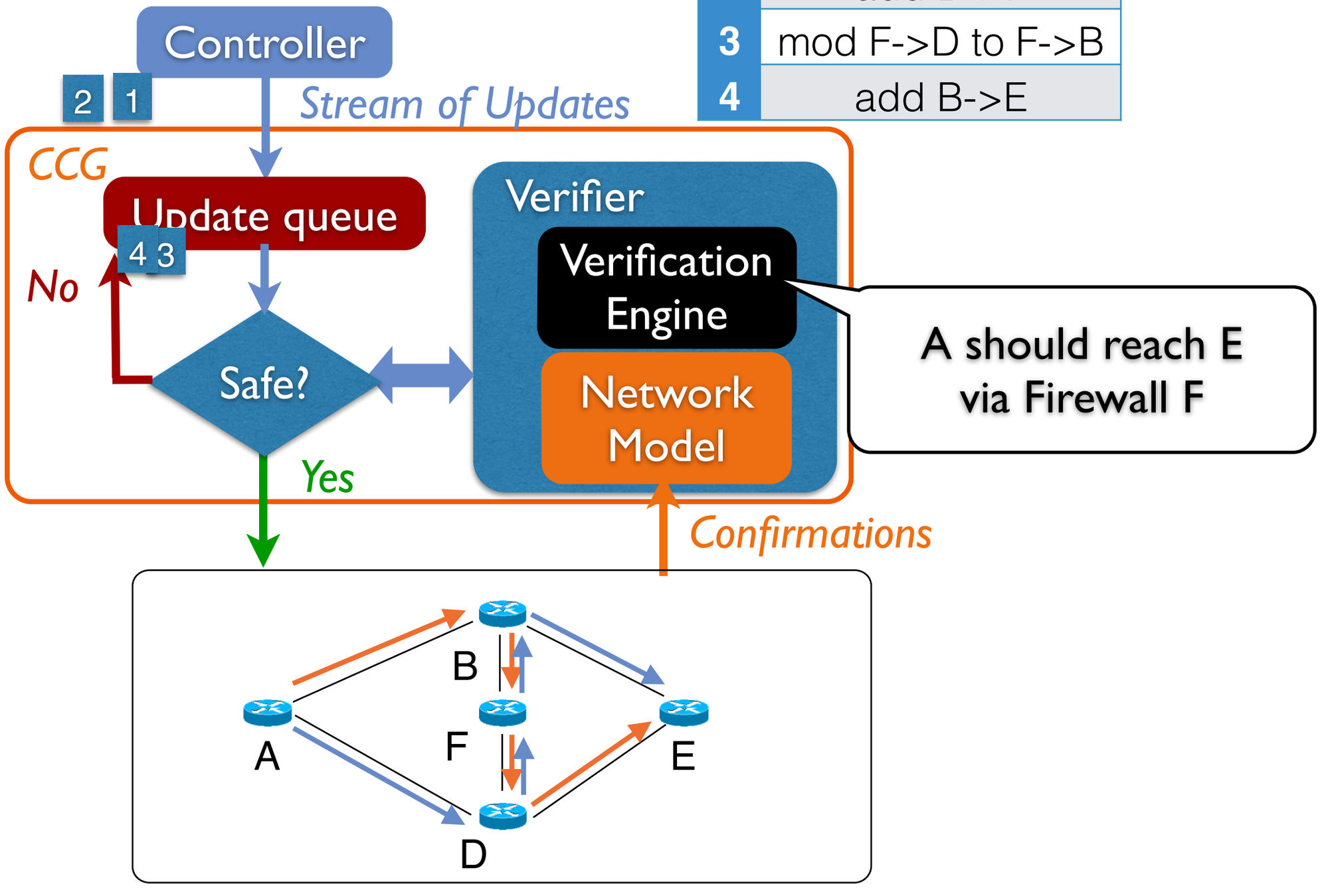
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



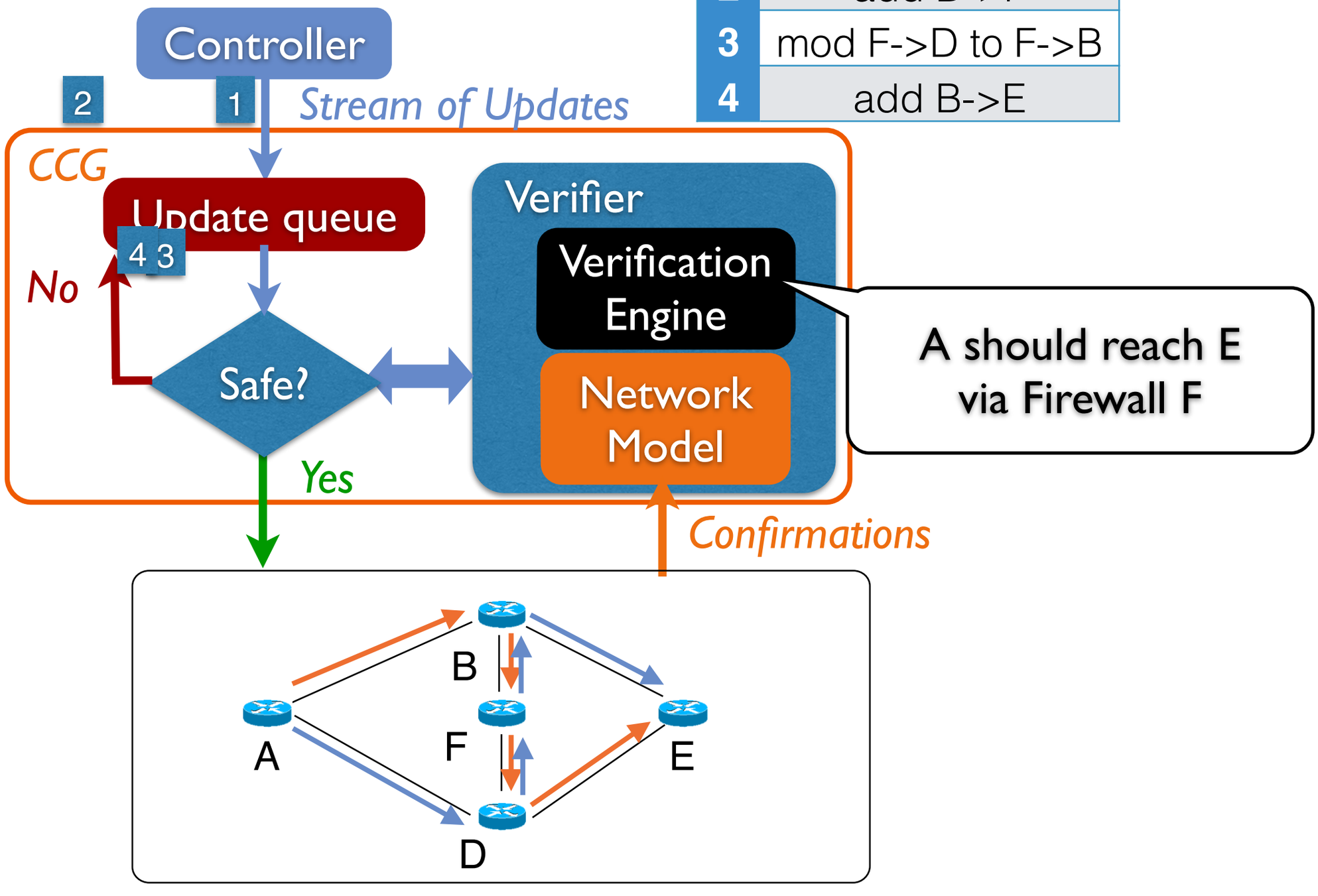
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



Consistency under Uncertainty — Problem

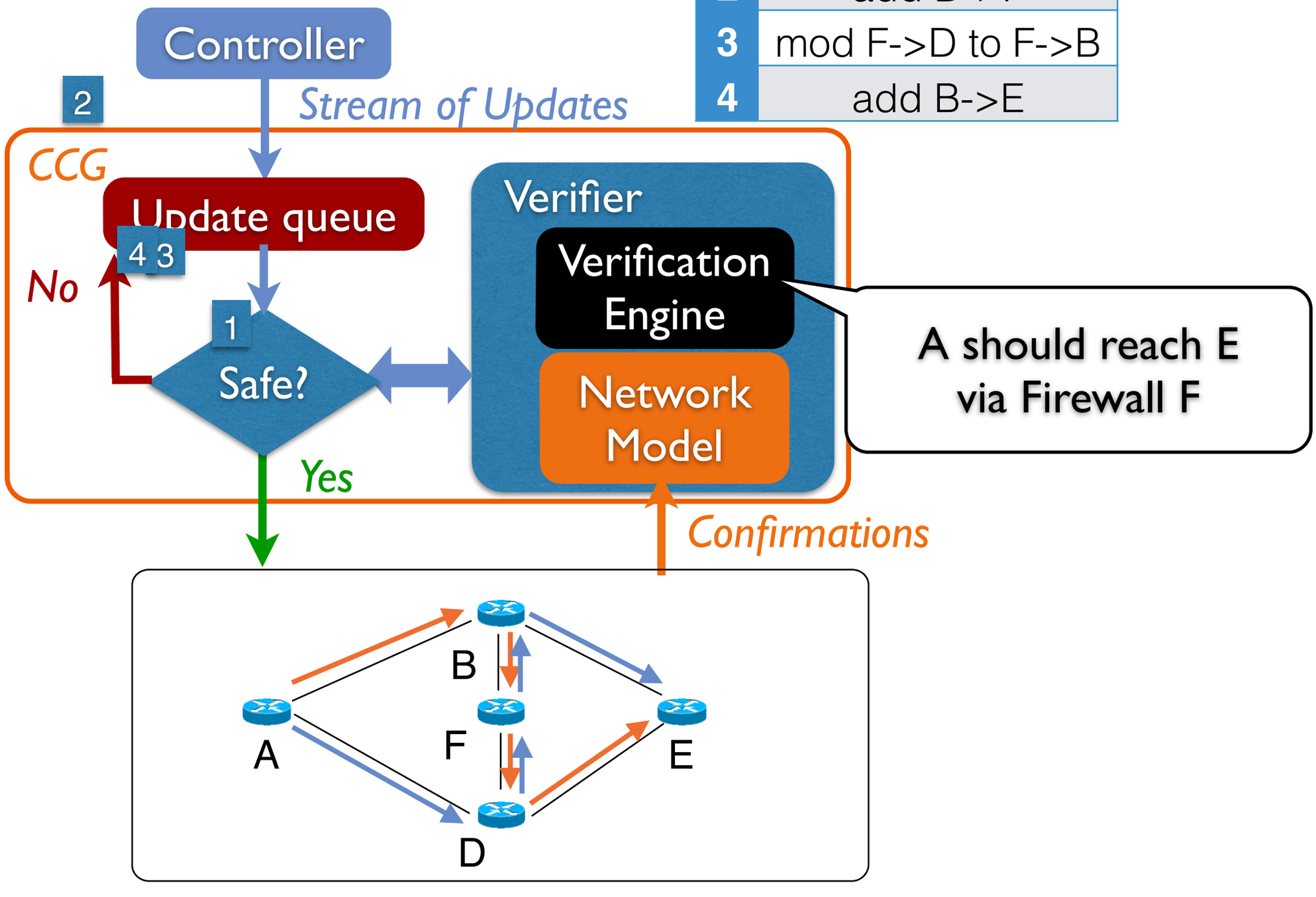
1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



Consistency under Uncertainty — Problem

2

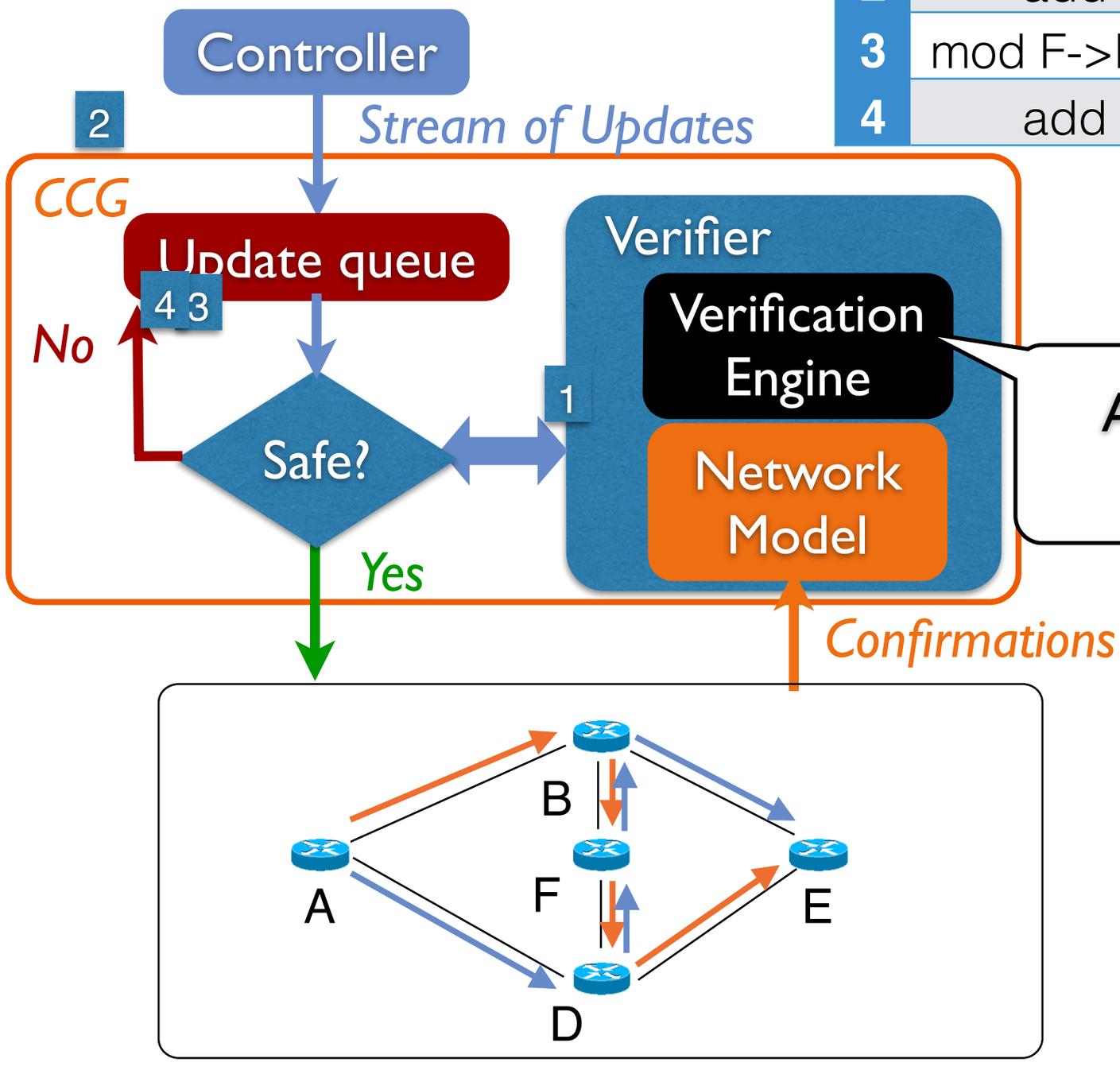
1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



Consistency under Uncertainty — Problem

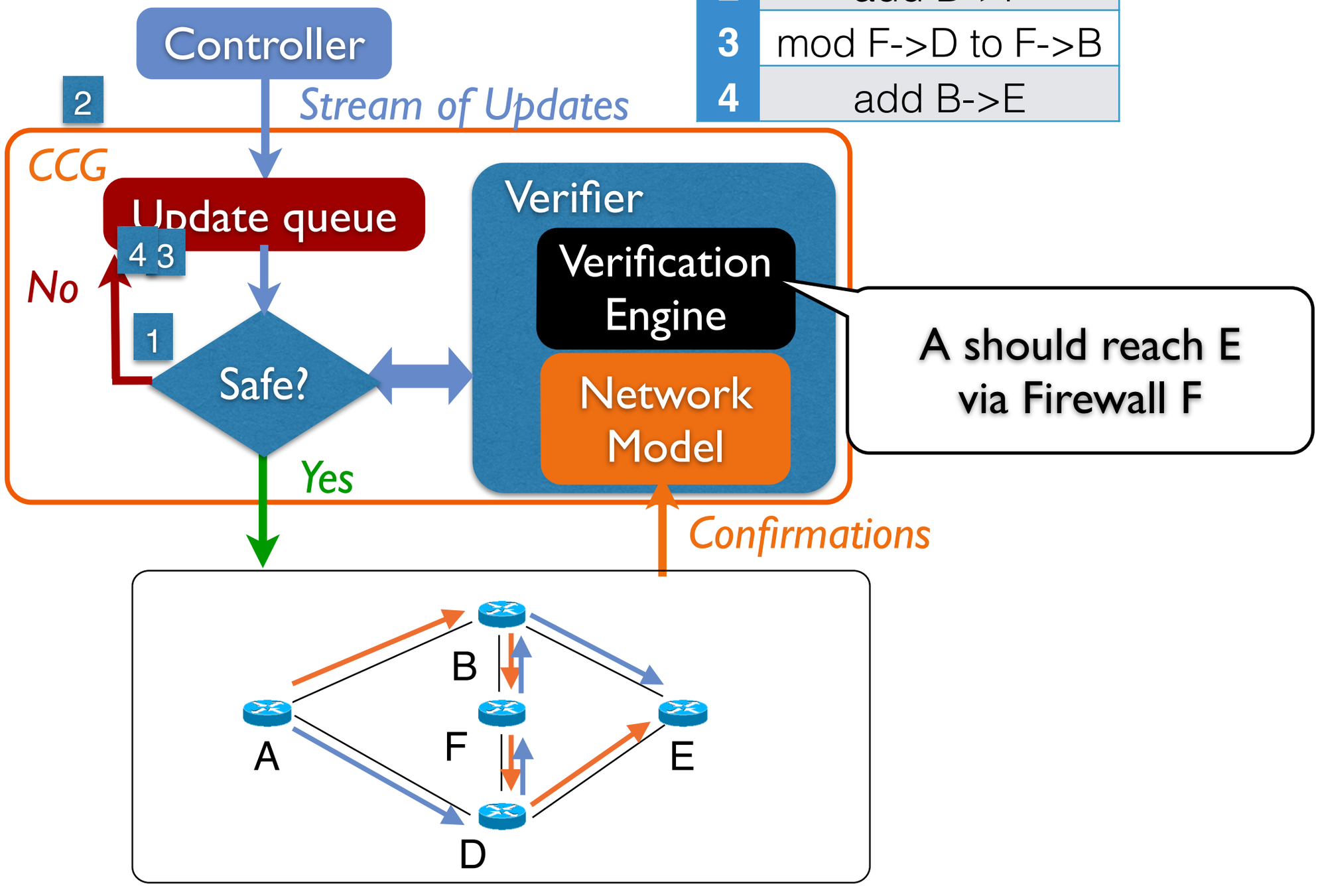
2

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



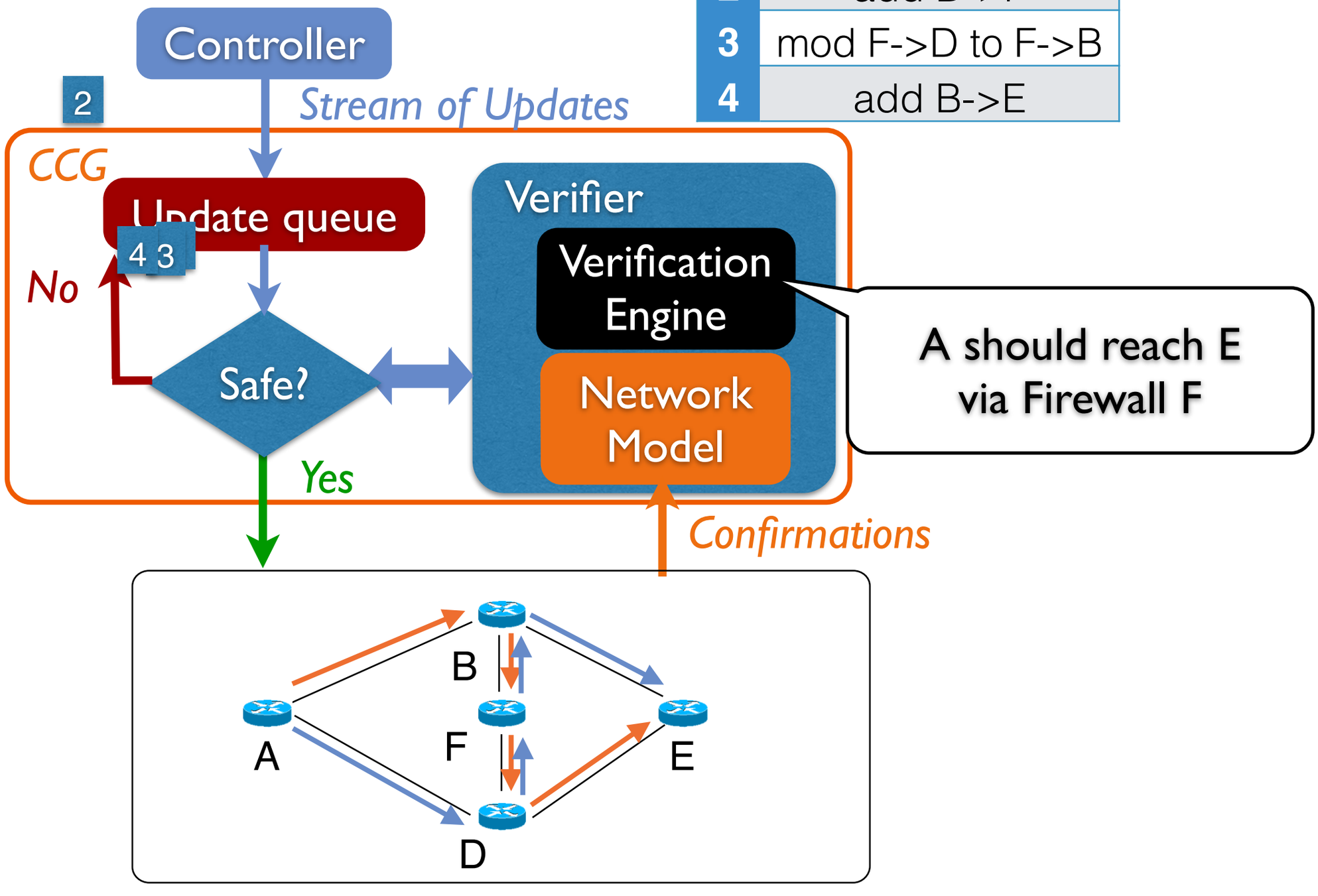
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



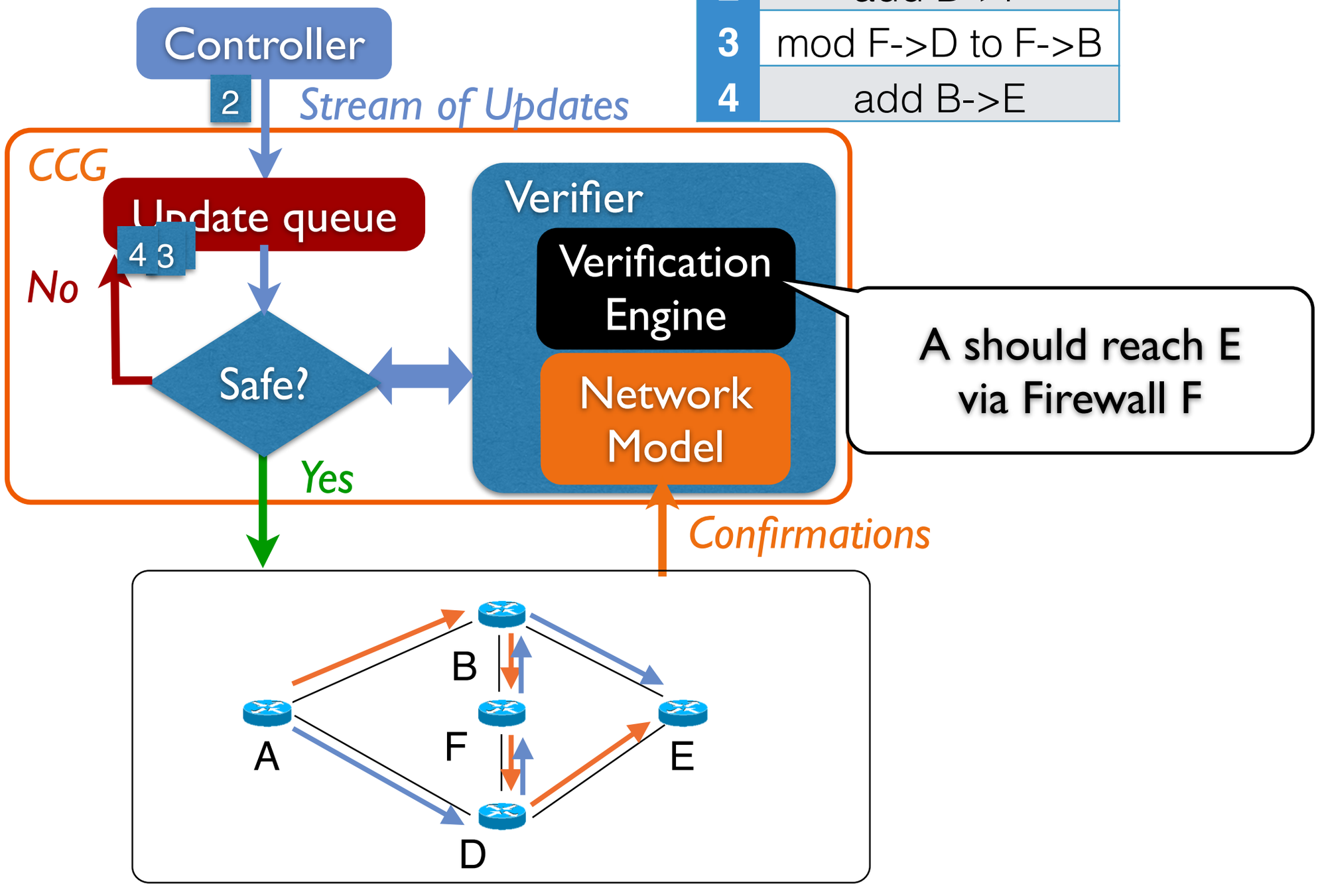
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



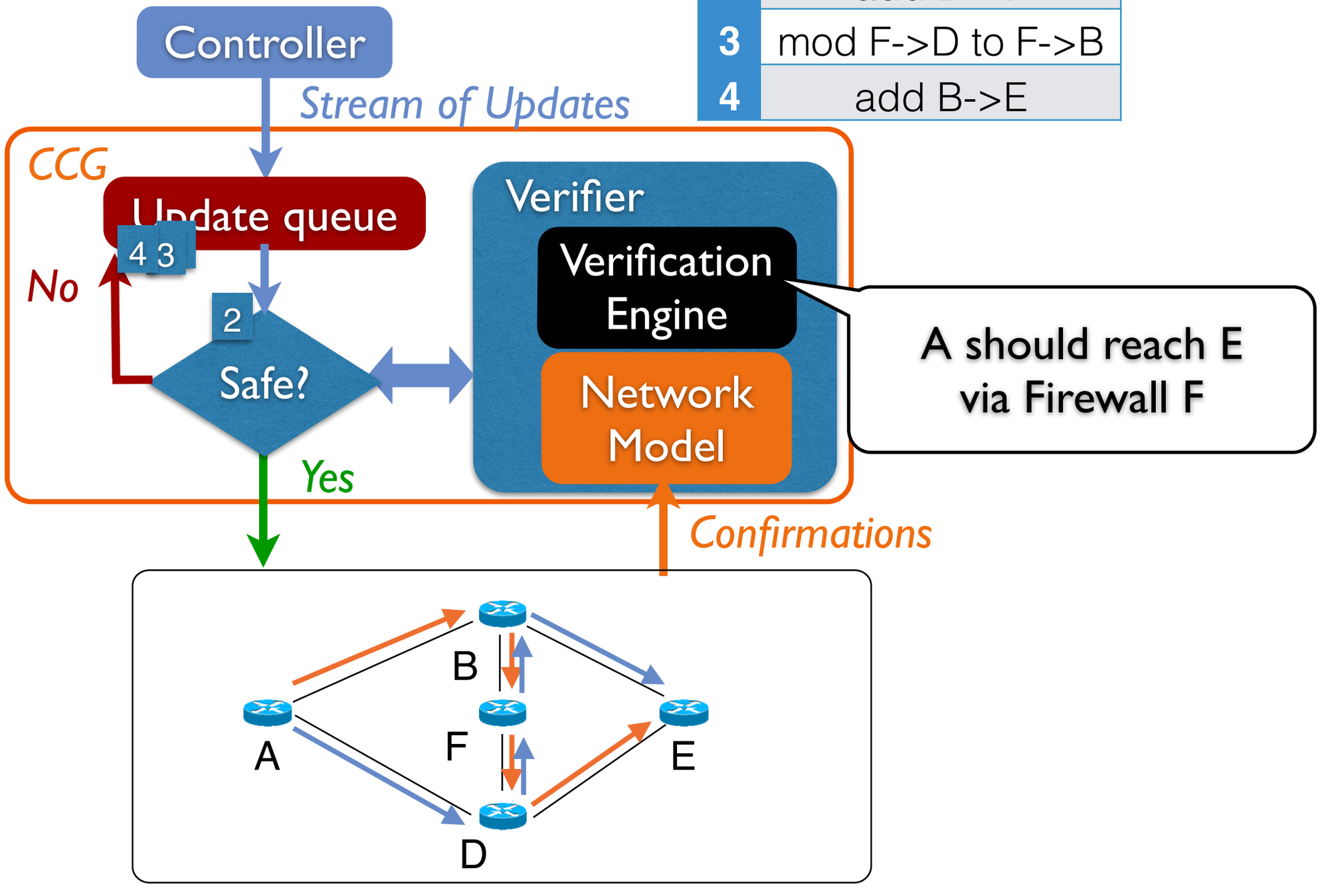
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



Consistency under Uncertainty — Problem

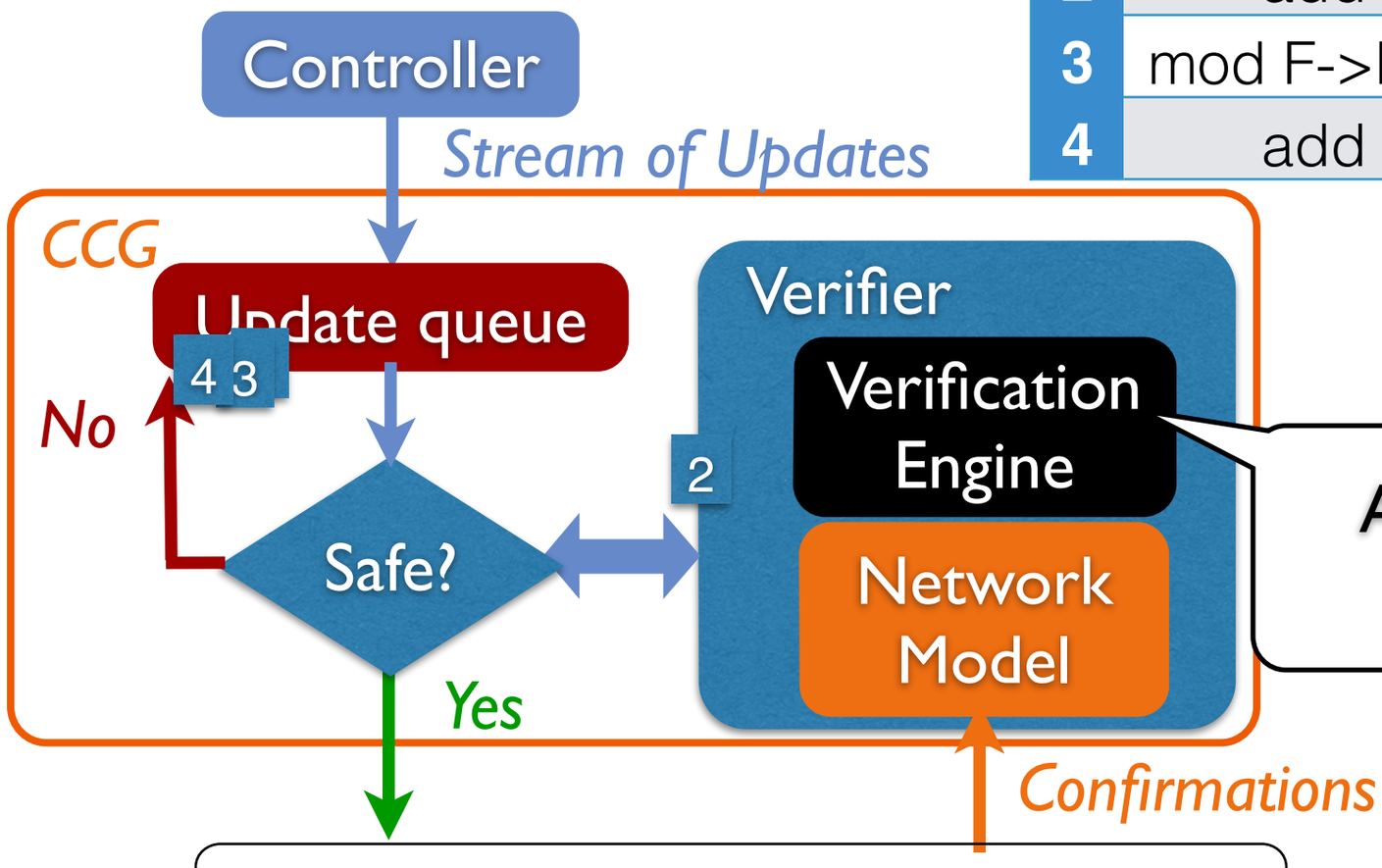
1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E

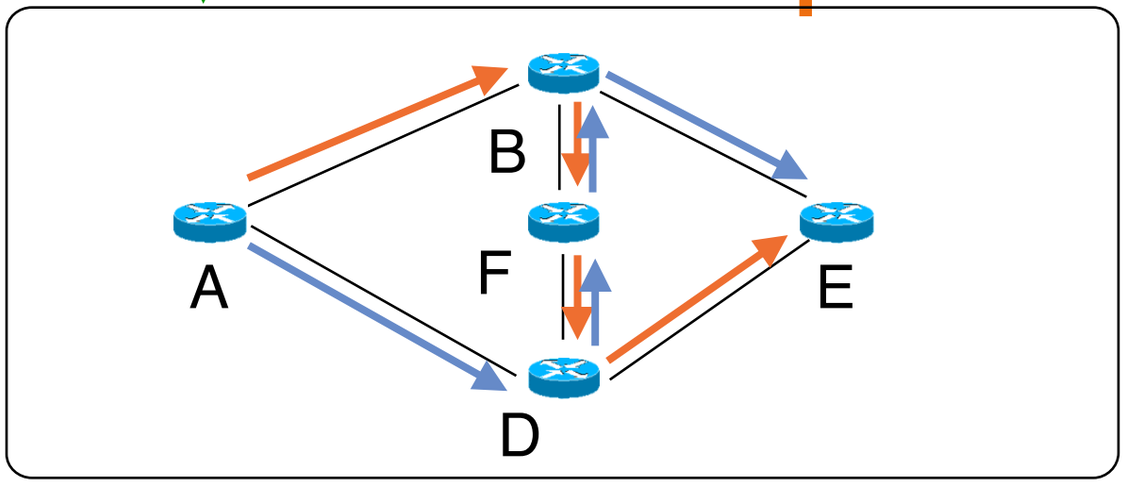


Consistency under Uncertainty — Problem

2

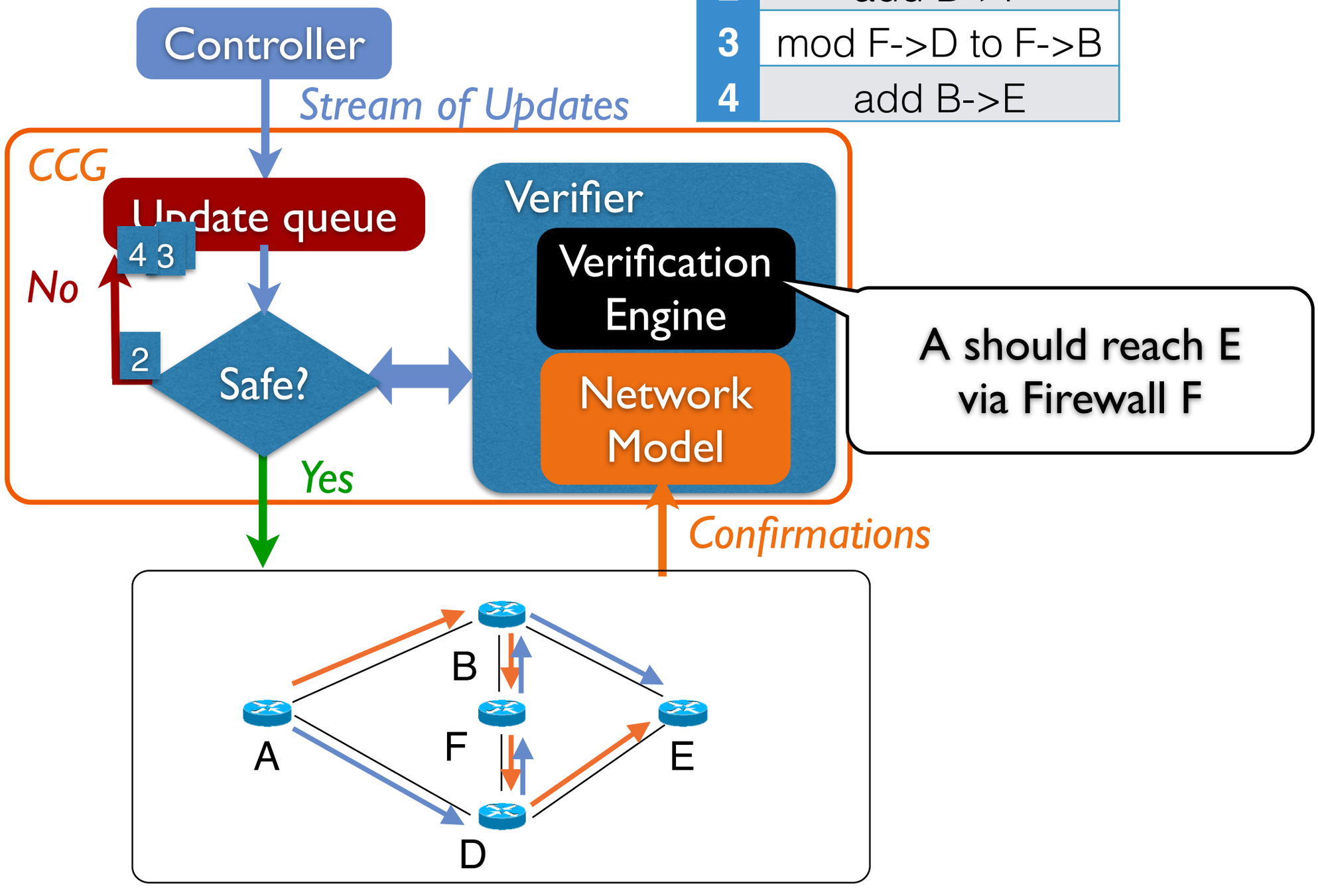
1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E





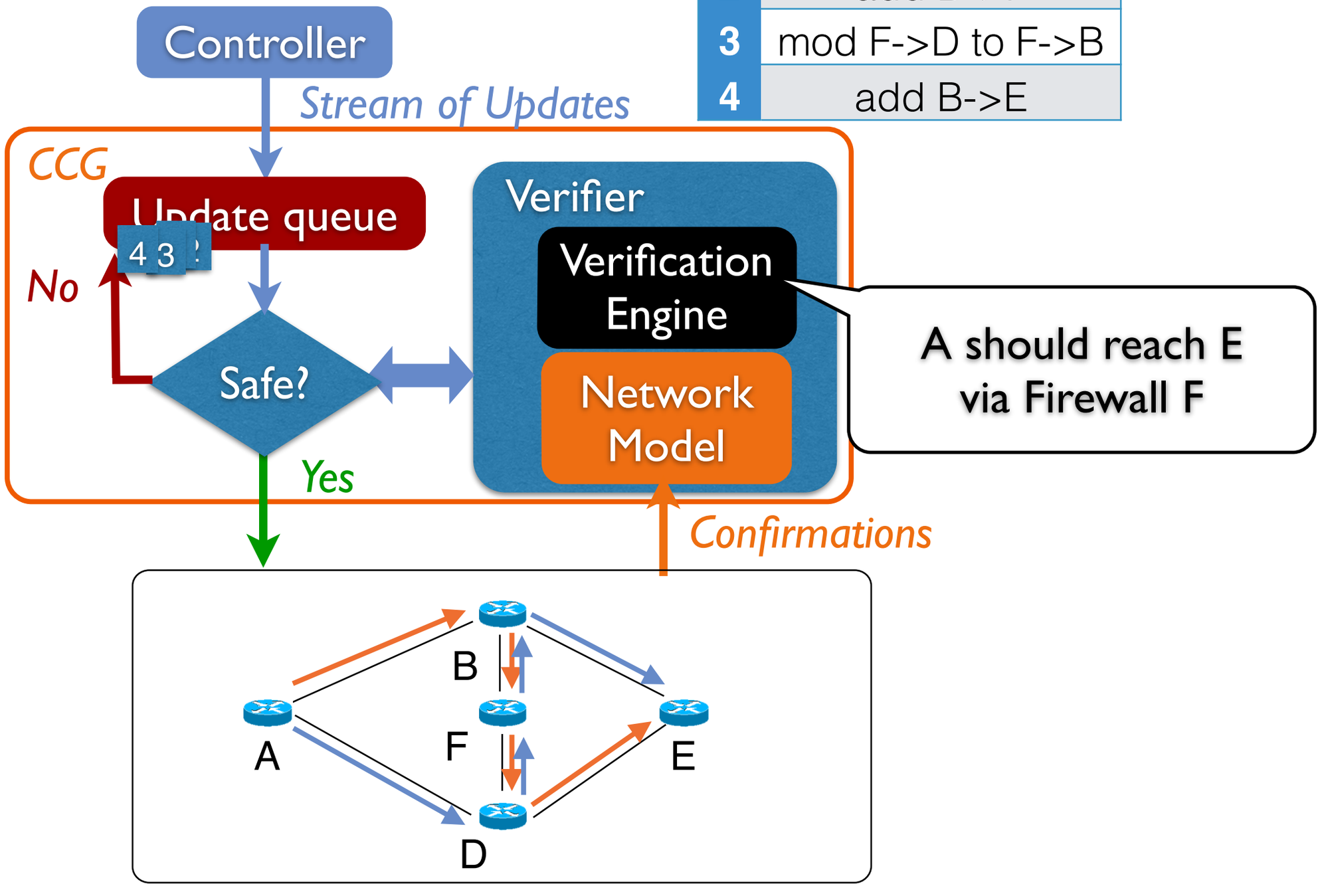
Consistency under Uncertainty — Problem

1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E

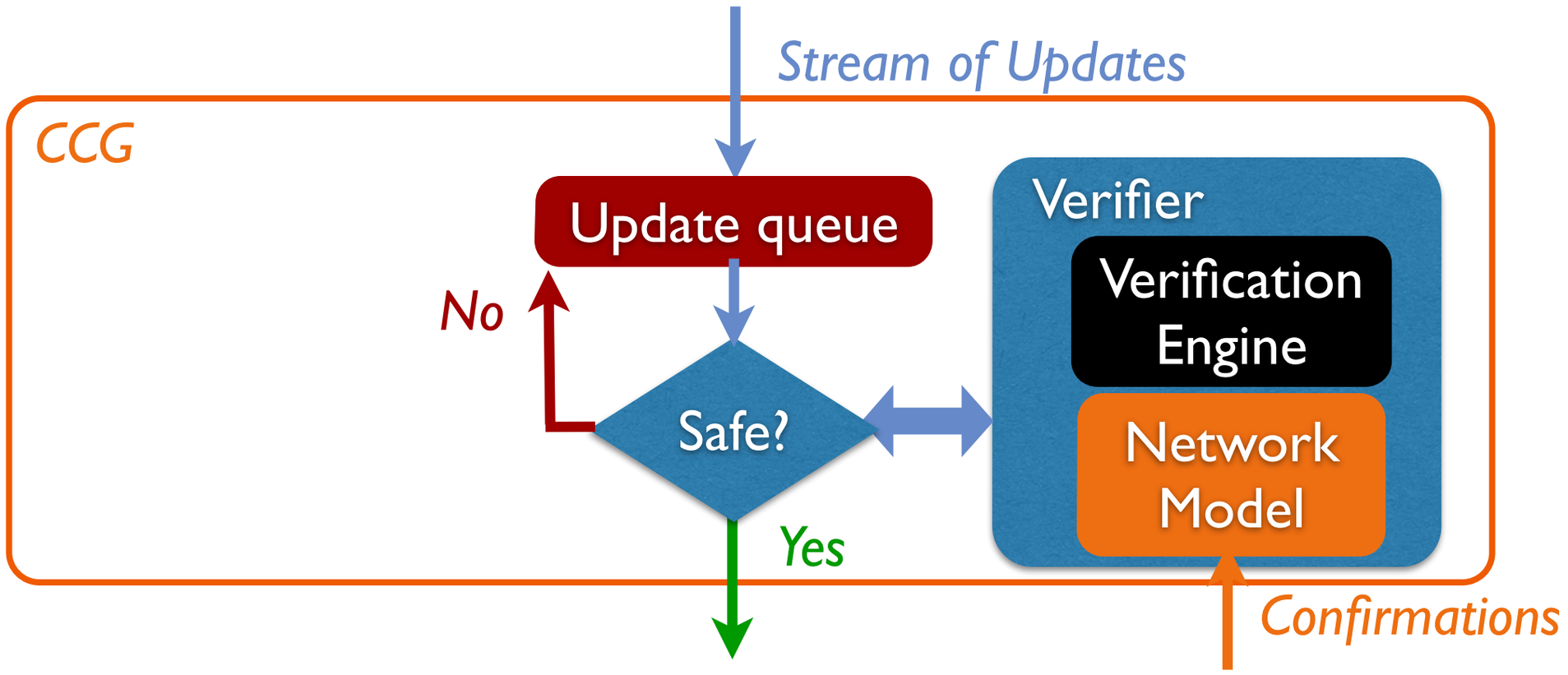


Consistency under Uncertainty — Problem

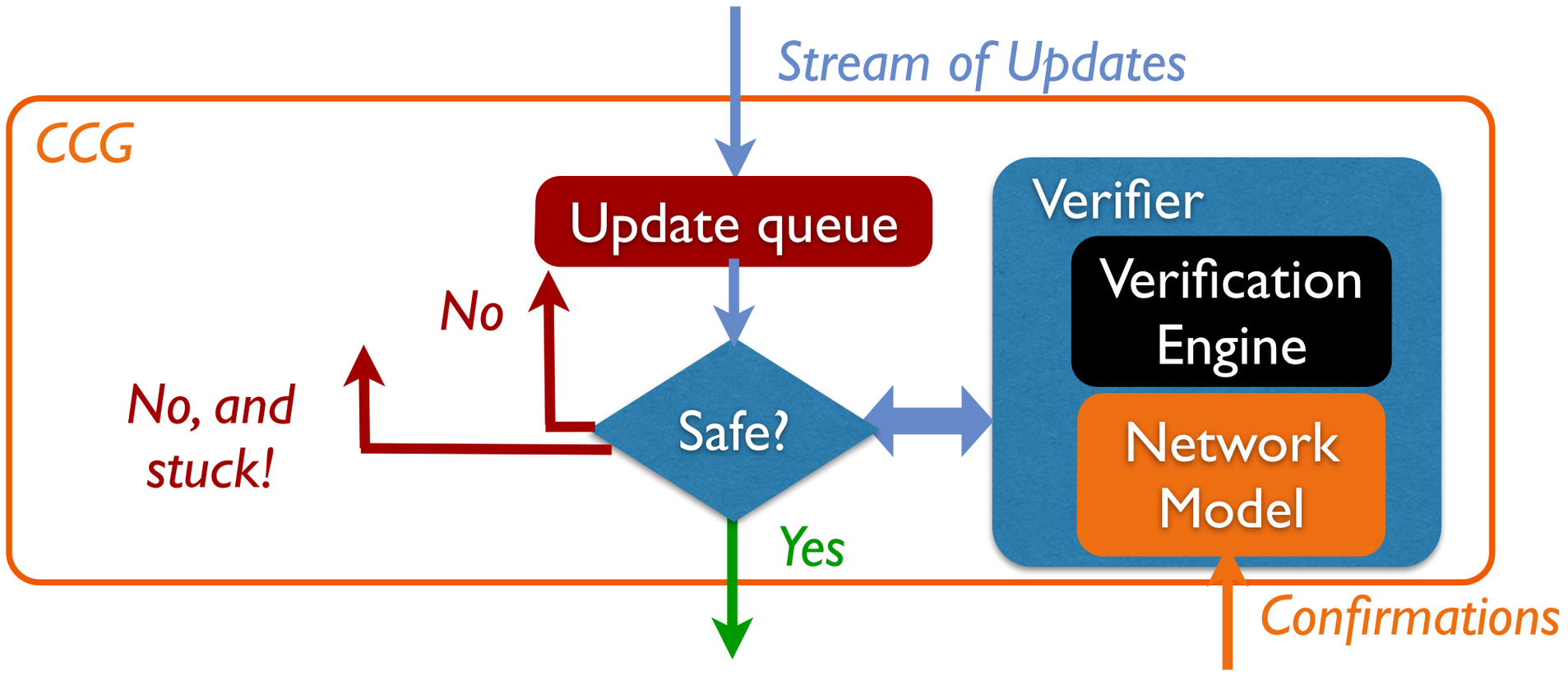
1	mod A->D to A->B
2	add D->F
3	mod F->D to F->B
4	add B->E



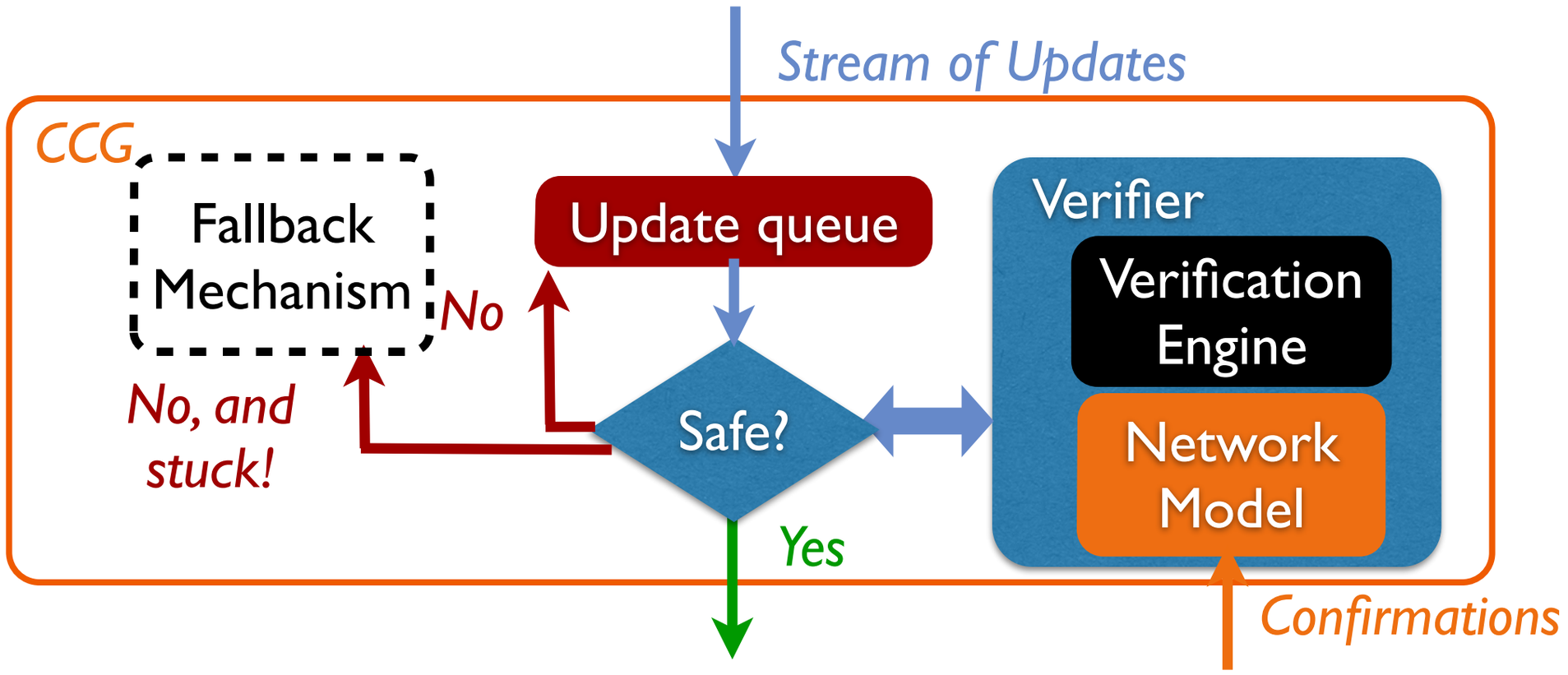
Consistency under Uncertainty — Solution



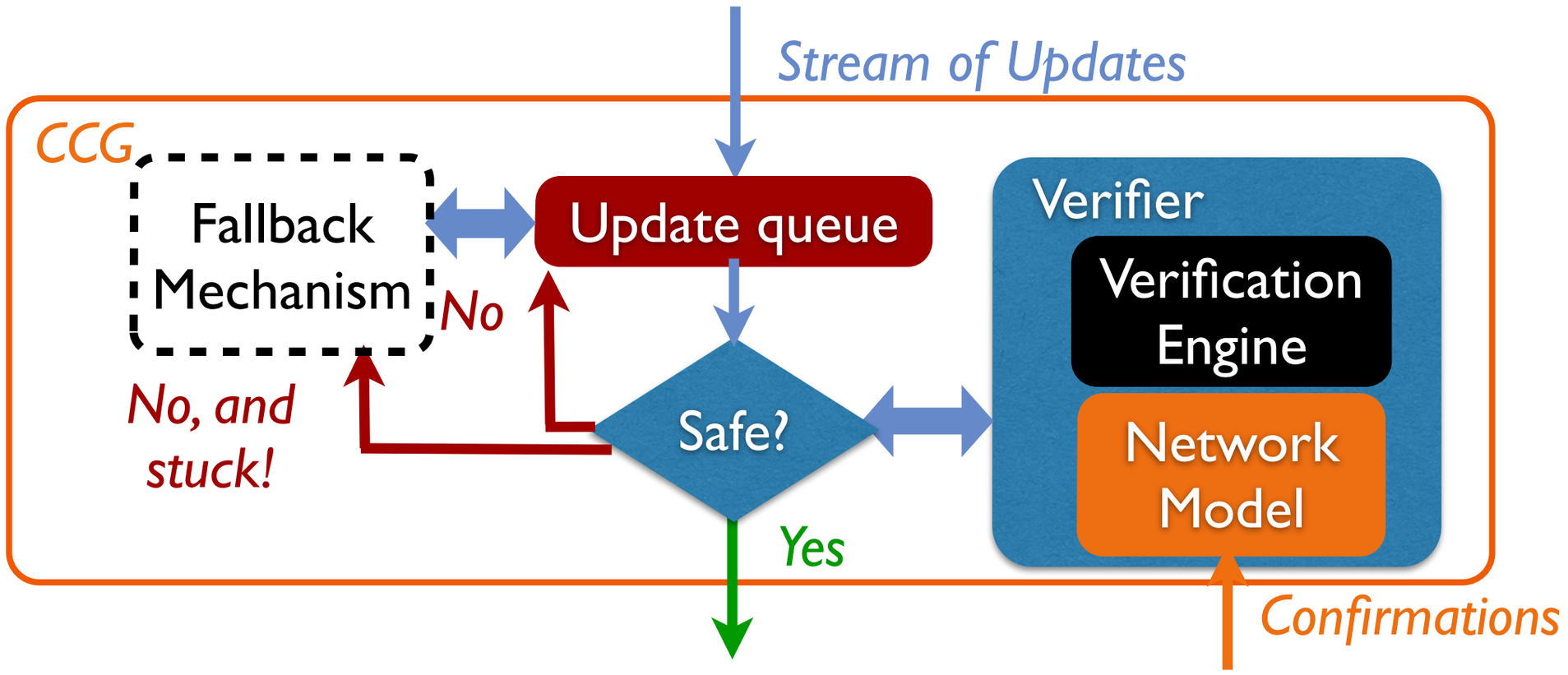
Consistency under Uncertainty — Solution



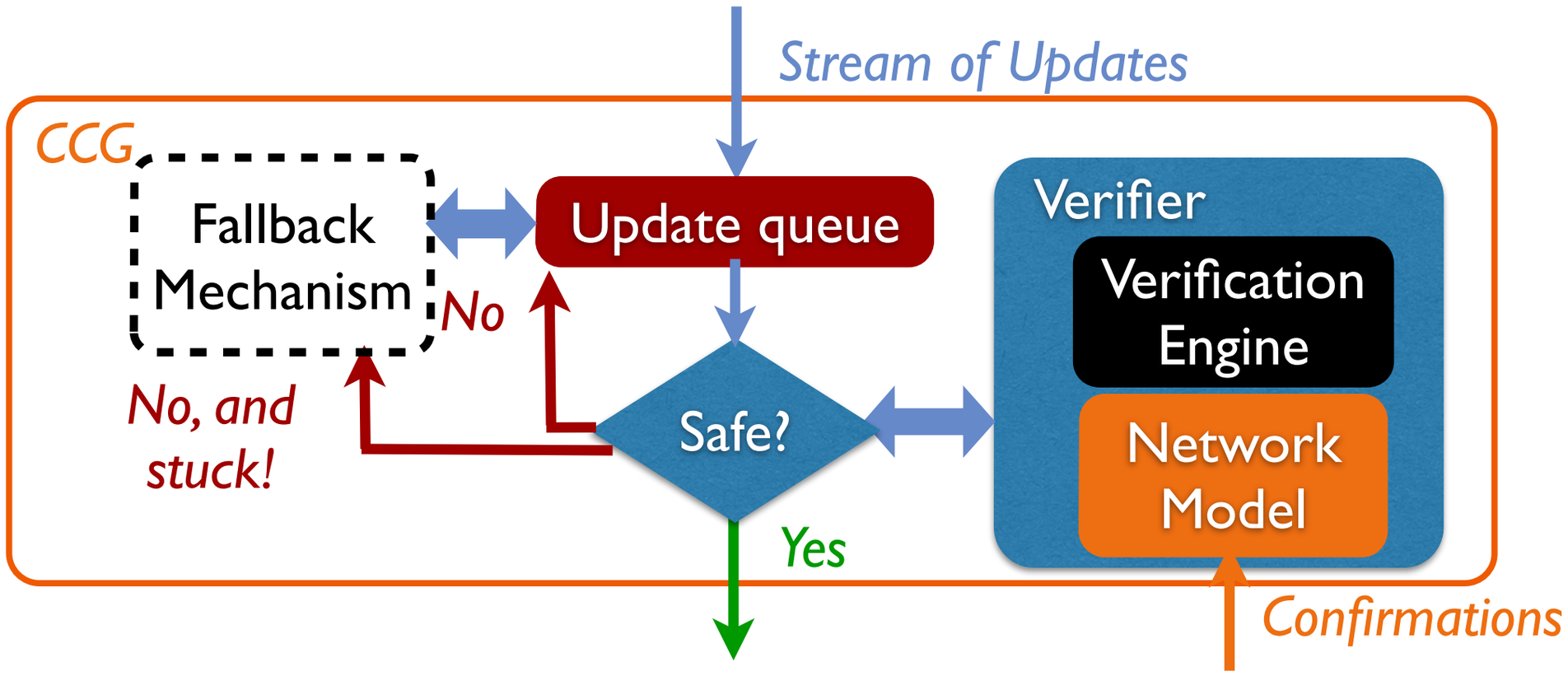
Consistency under Uncertainty — Solution



Consistency under Uncertainty — Solution

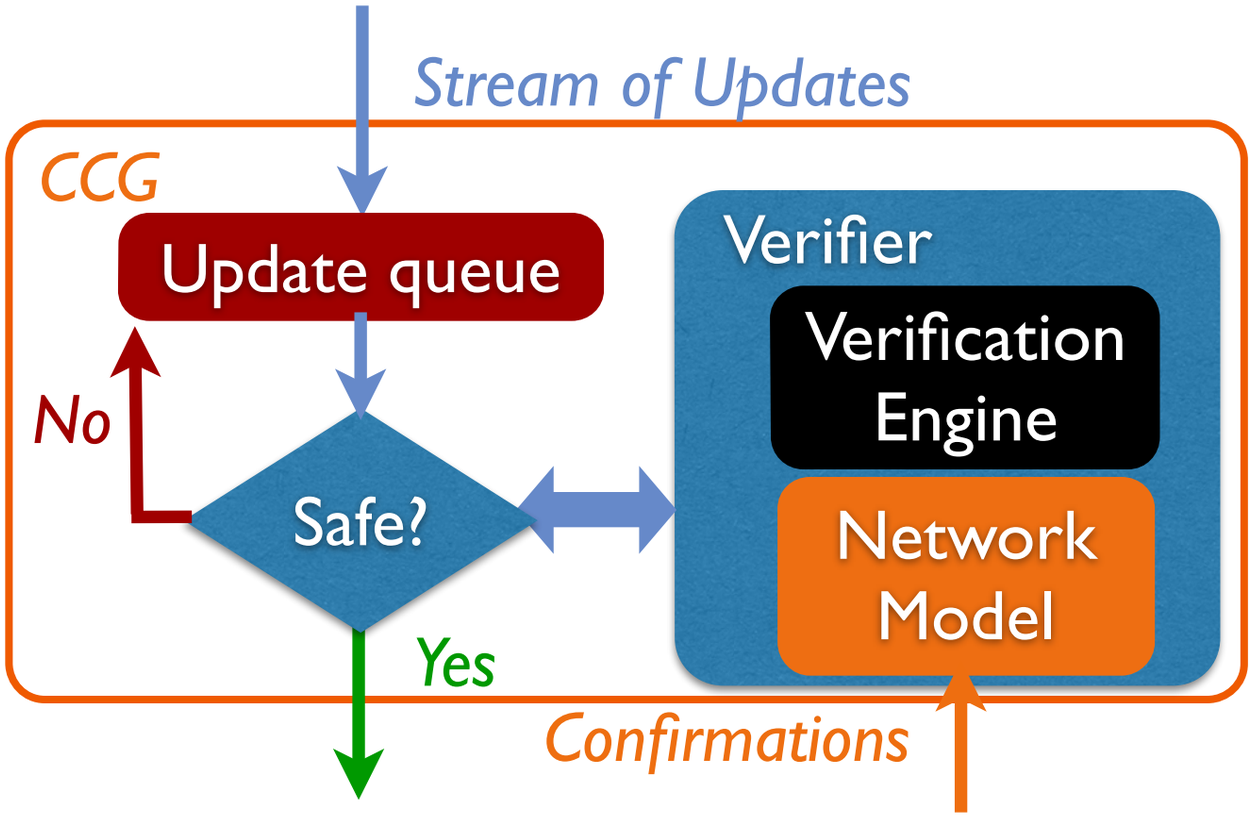


Consistency under Uncertainty — Solution

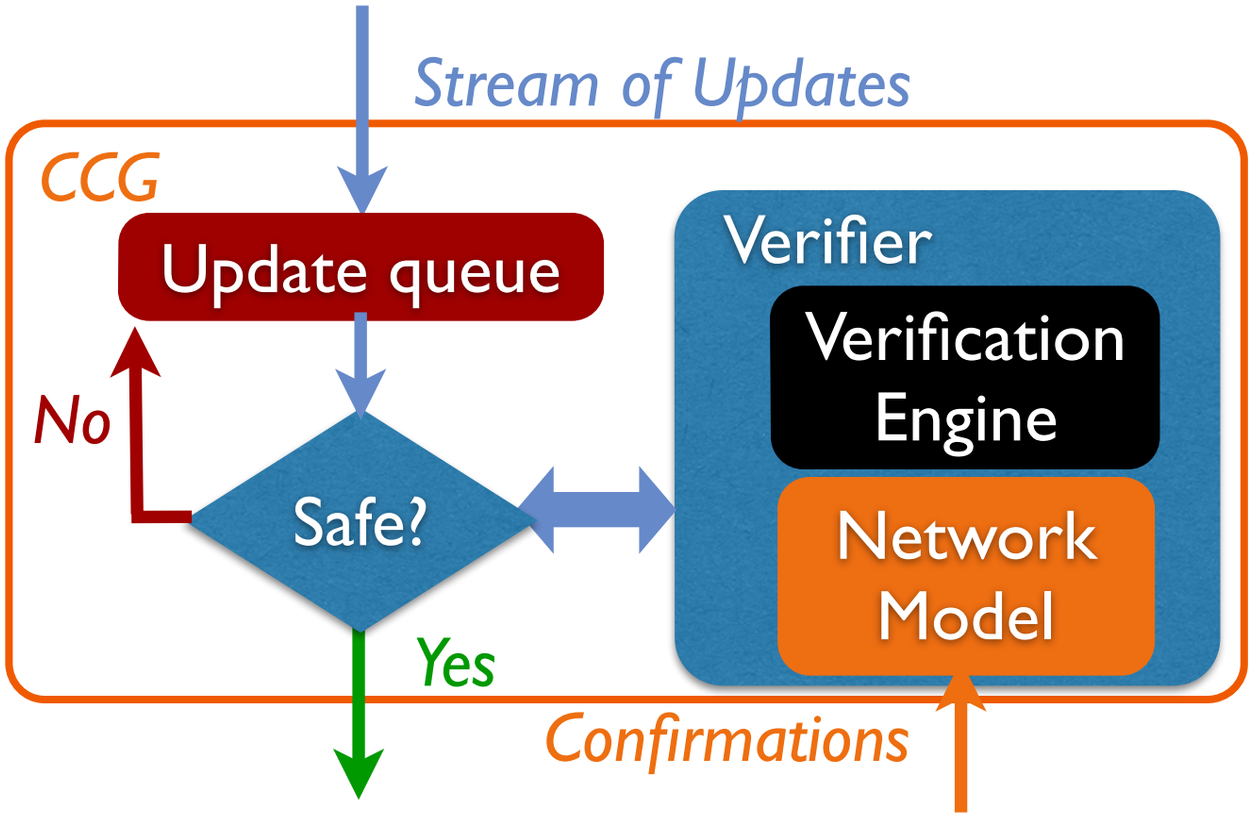


Even with Fallback triggered, CCG achieves better efficiency than using Fallback alone.

Consistency under Uncertainty



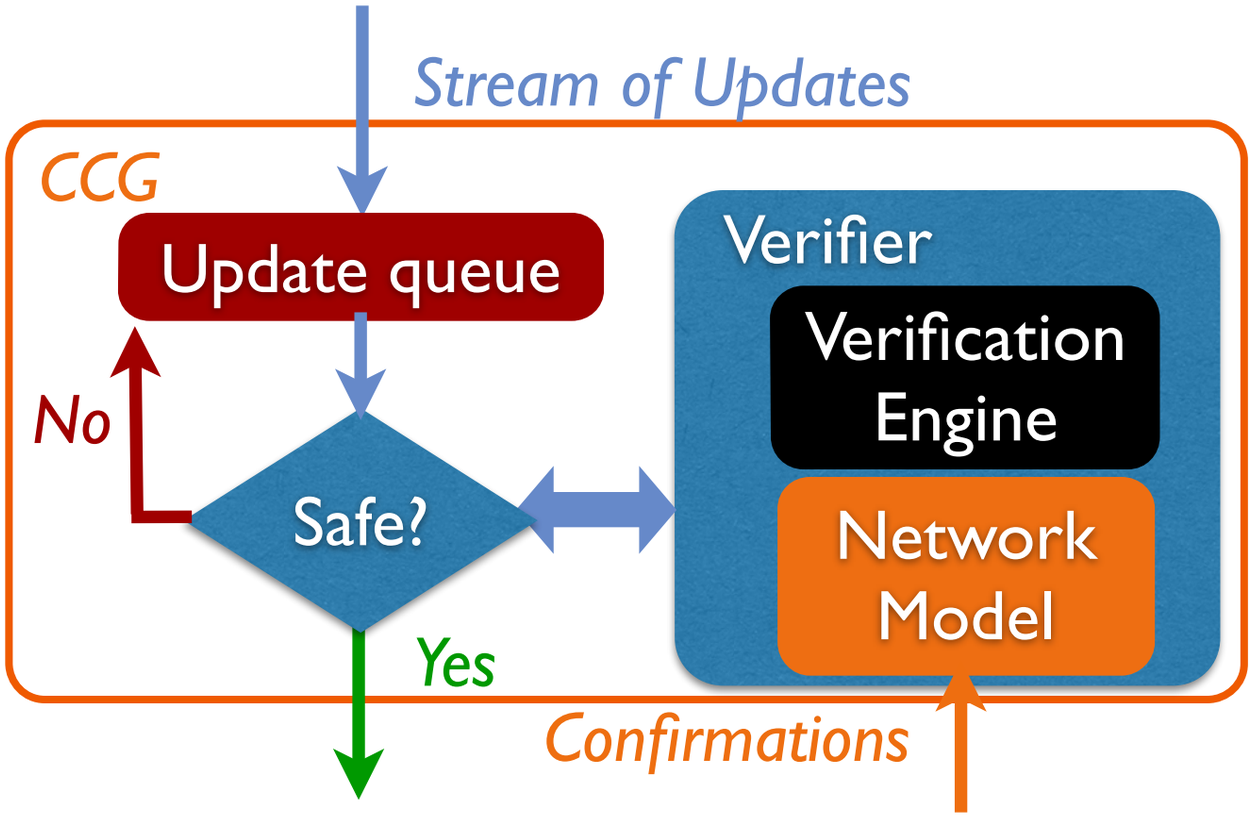
Consistency under Uncertainty



Waypoint Properties: flows are required to traverse a set of waypoints

- connectivity,
- waypointing,
- access control,
- service chaining, …

Consistency under Uncertainty

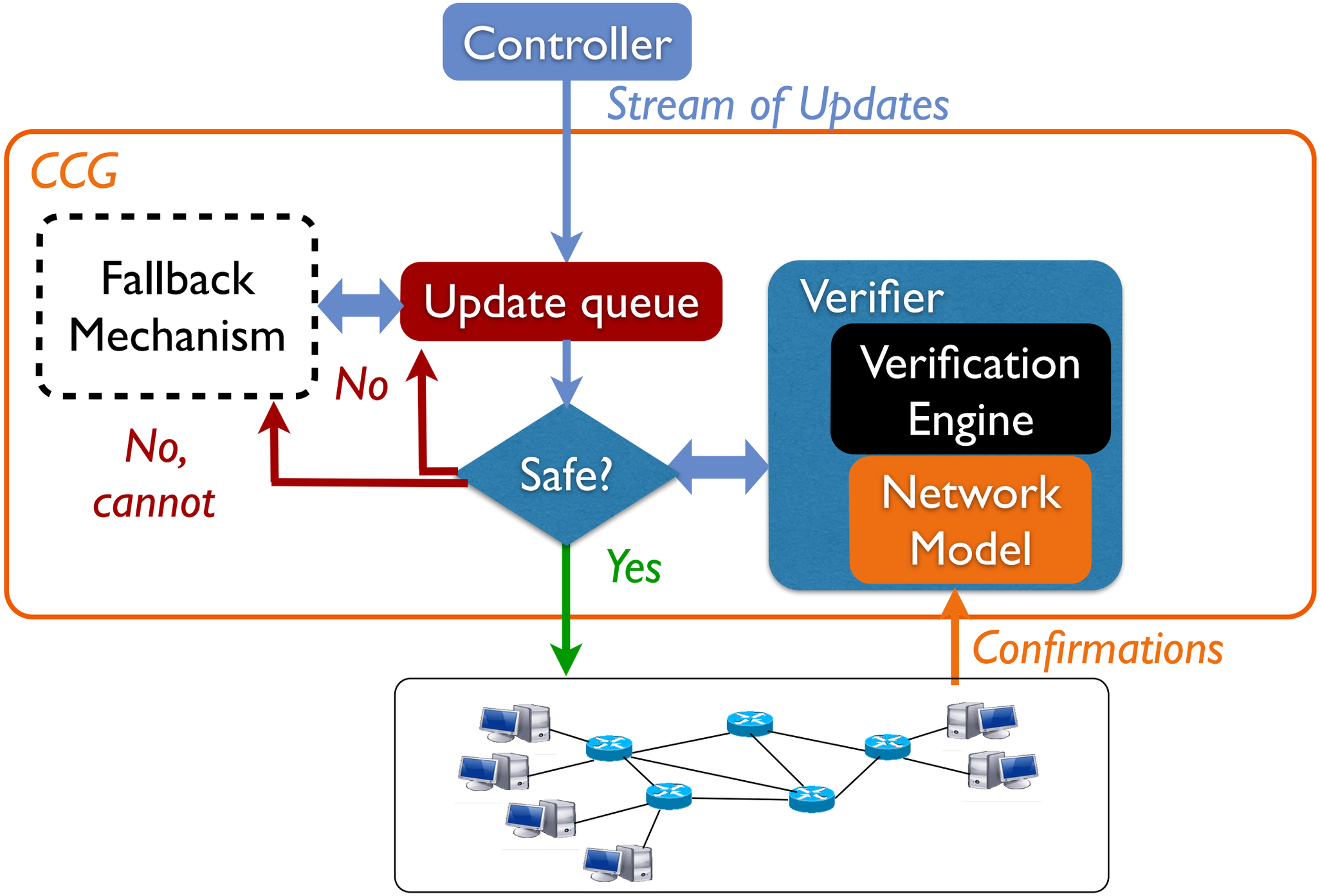


Waypoint Properties: flows are required to traverse a set of waypoints

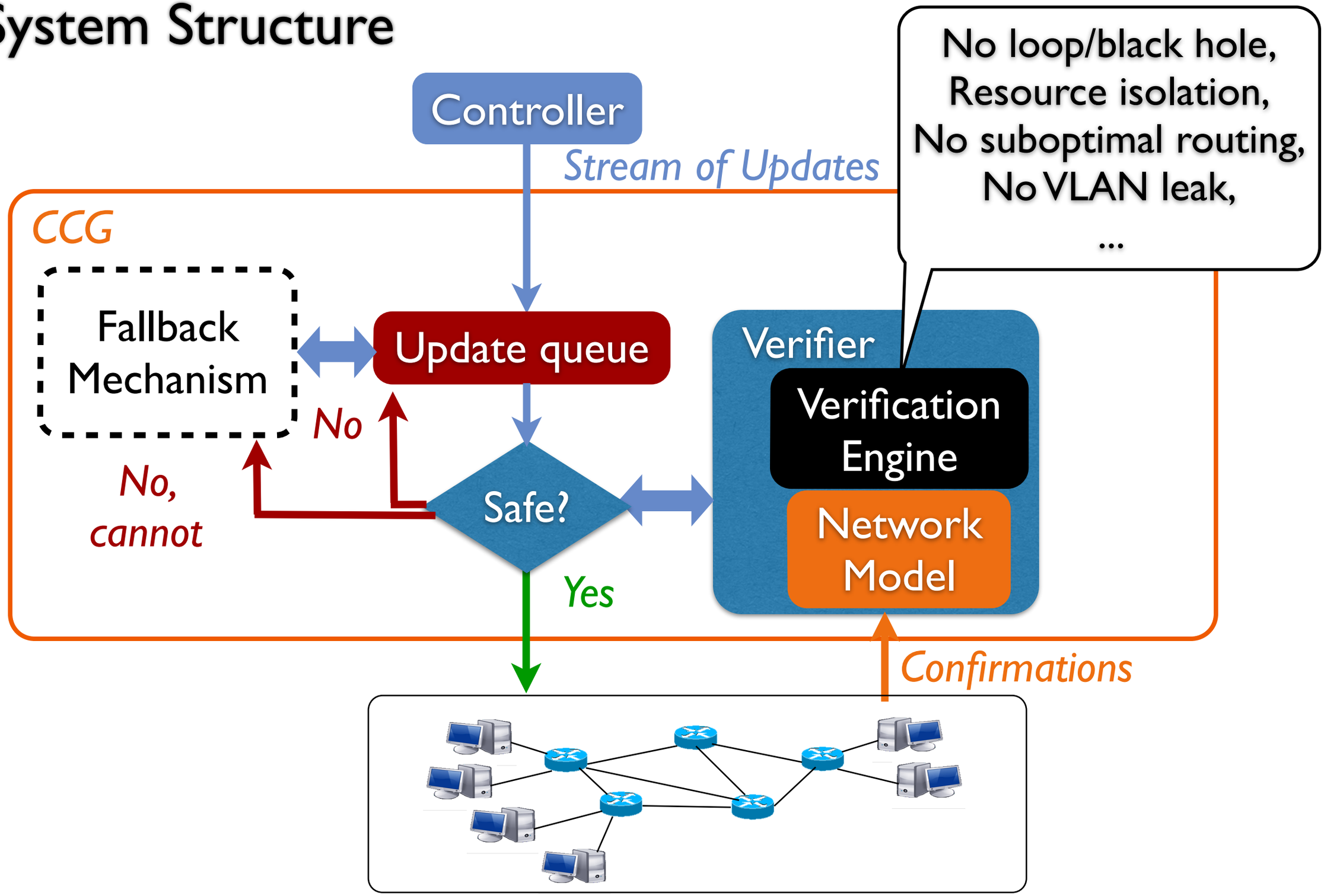
- connectivity,
- waypointing,
- access control,
- service chaining, …

Theorem: *Segment independent* properties are guaranteed to complete using greedy updates.

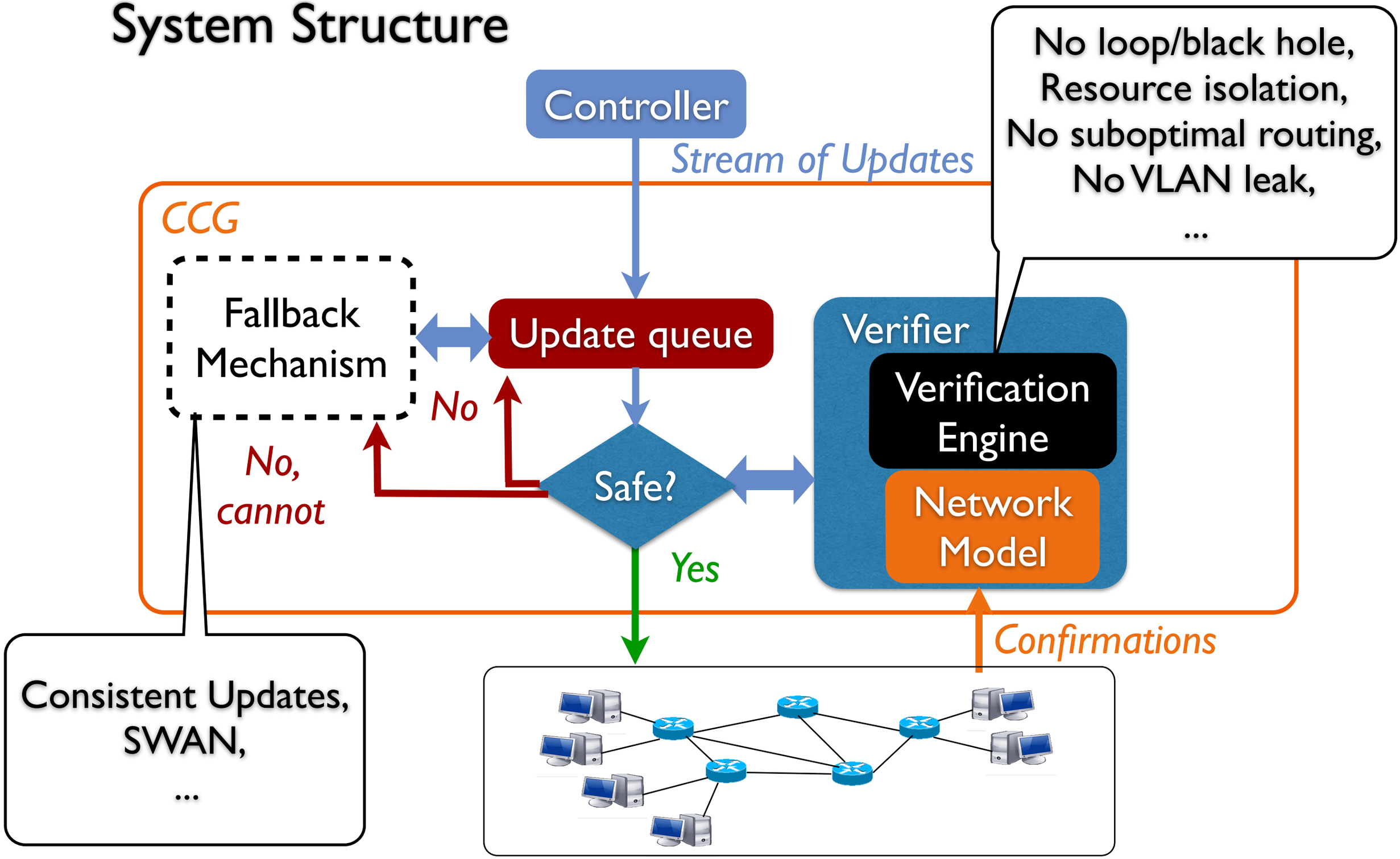
System Structure



System Structure



System Structure



Evaluation

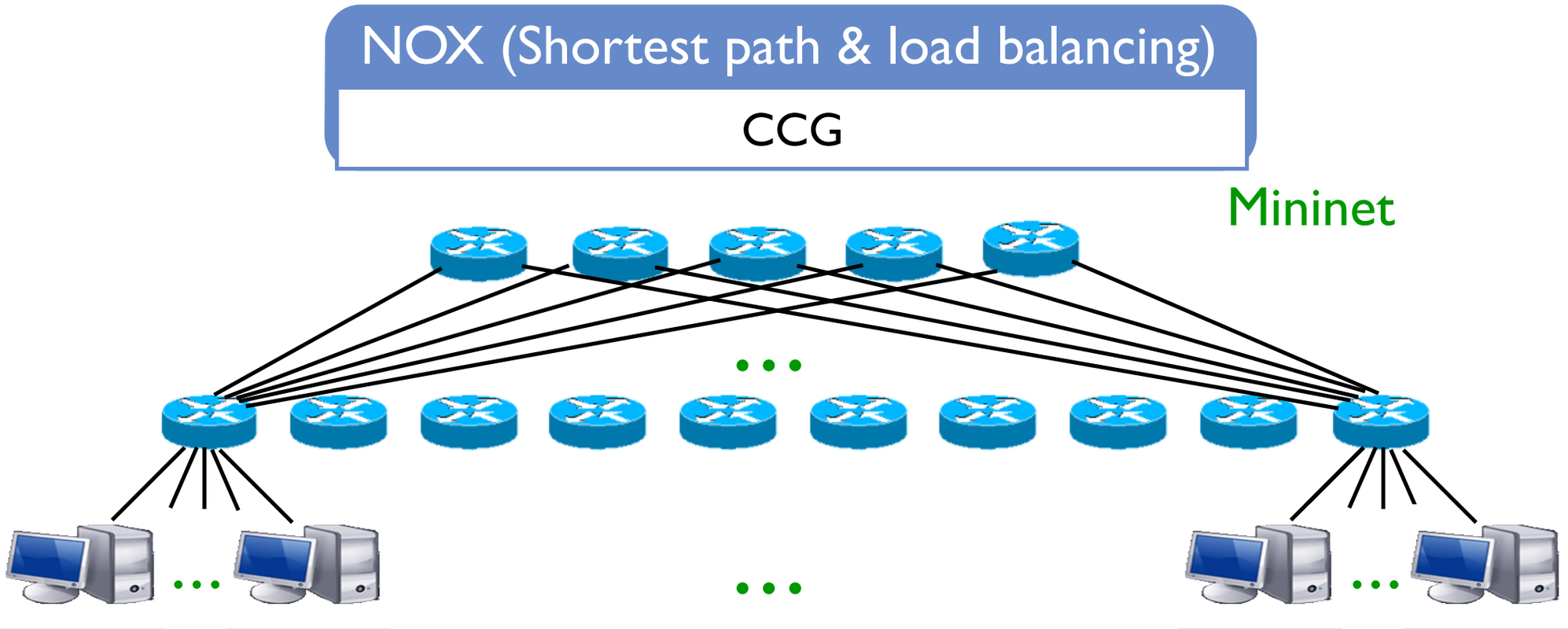
Can CCG improve ***performance*** over prior work?

- Segment-independent Properties
- Non-segment-independent Properties

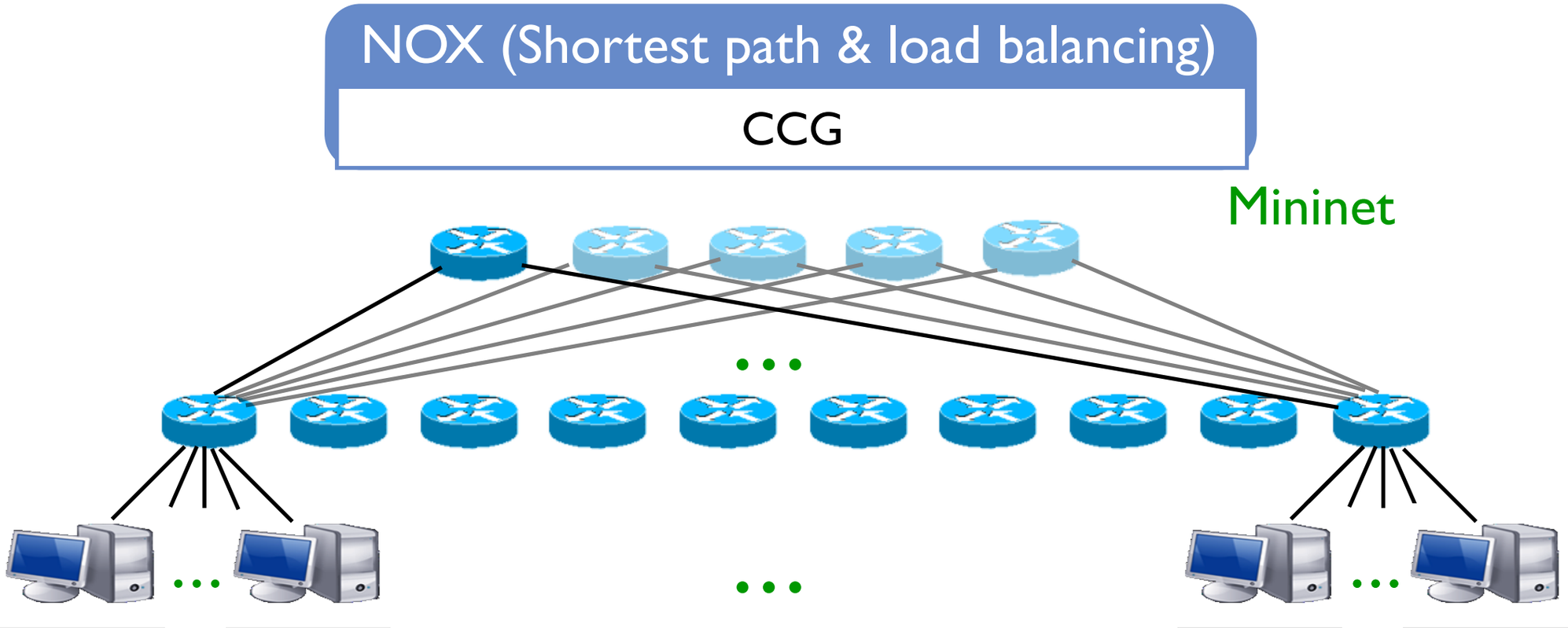
Evaluation approaches:

- Emulations
- Testbed experiments

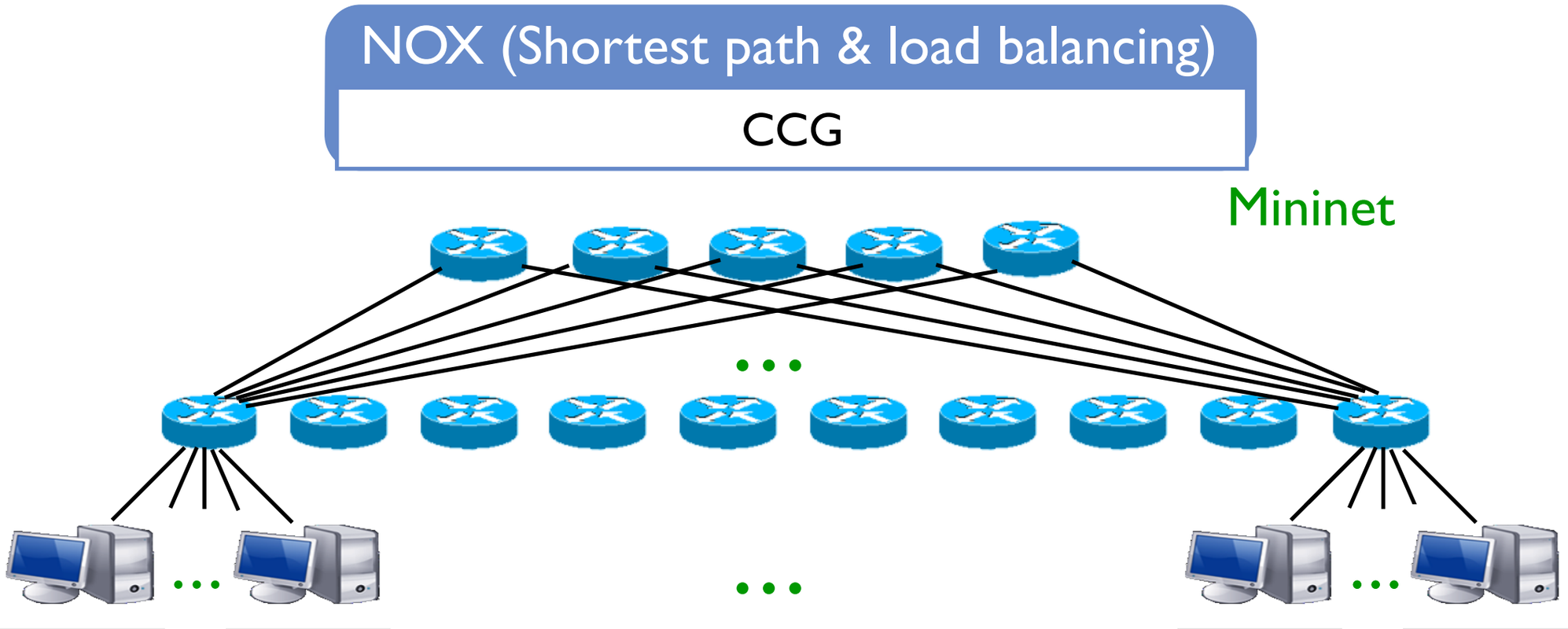
Emulation: Segment-independent Properties



Emulation: Segment-independent Properties

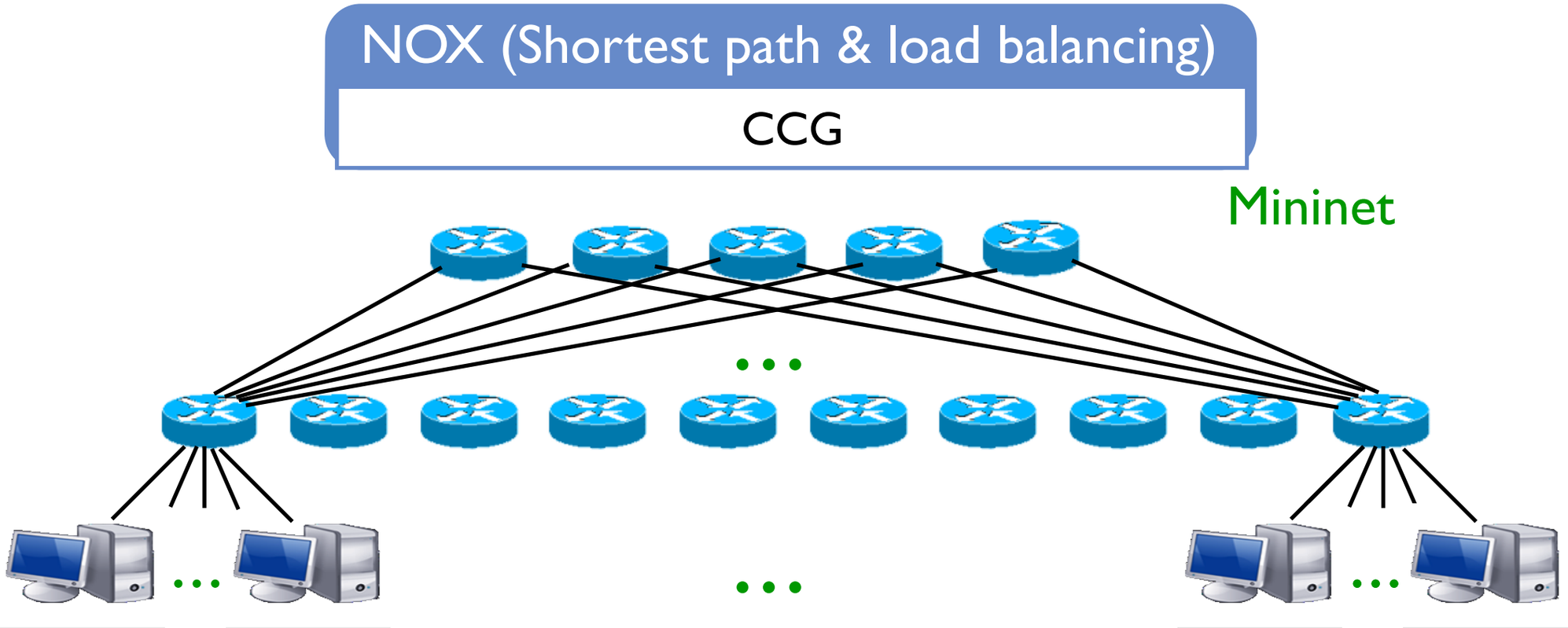


Emulation: Segment-independent Properties



Emulation: Segment-independent Properties

Controller-switch delay = network delay + processing delay

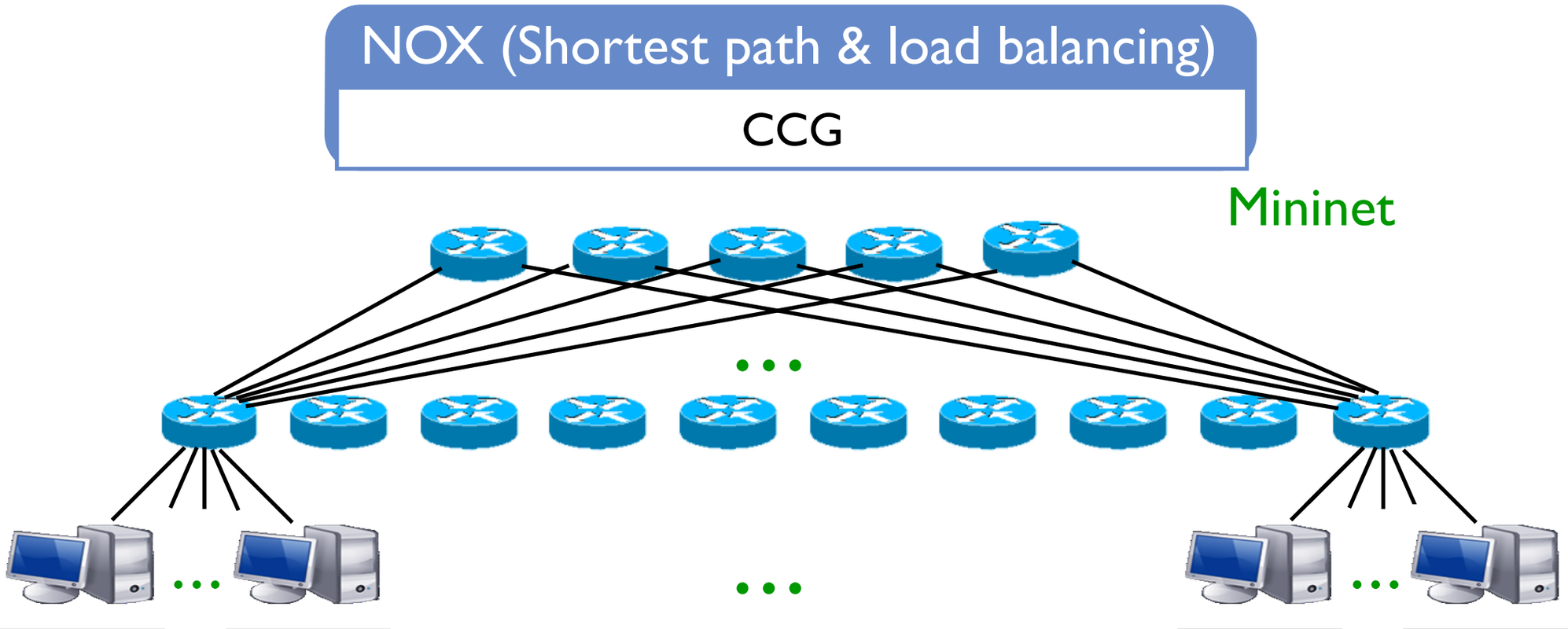


Emulation: Segment-independent Properties

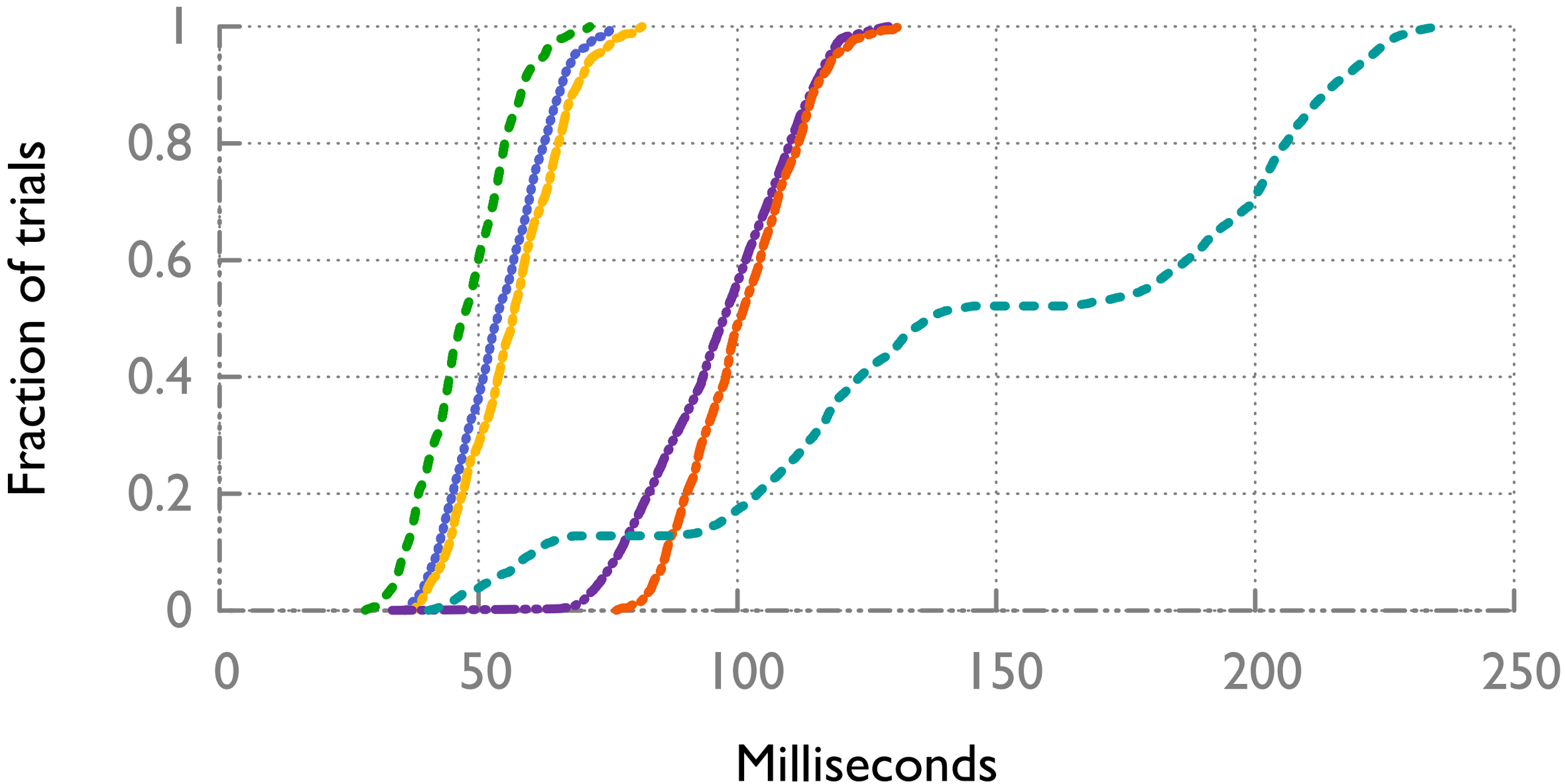
Controller-switch delay = network delay + processing delay

- Local (4ms)
- Wide area (100ms)

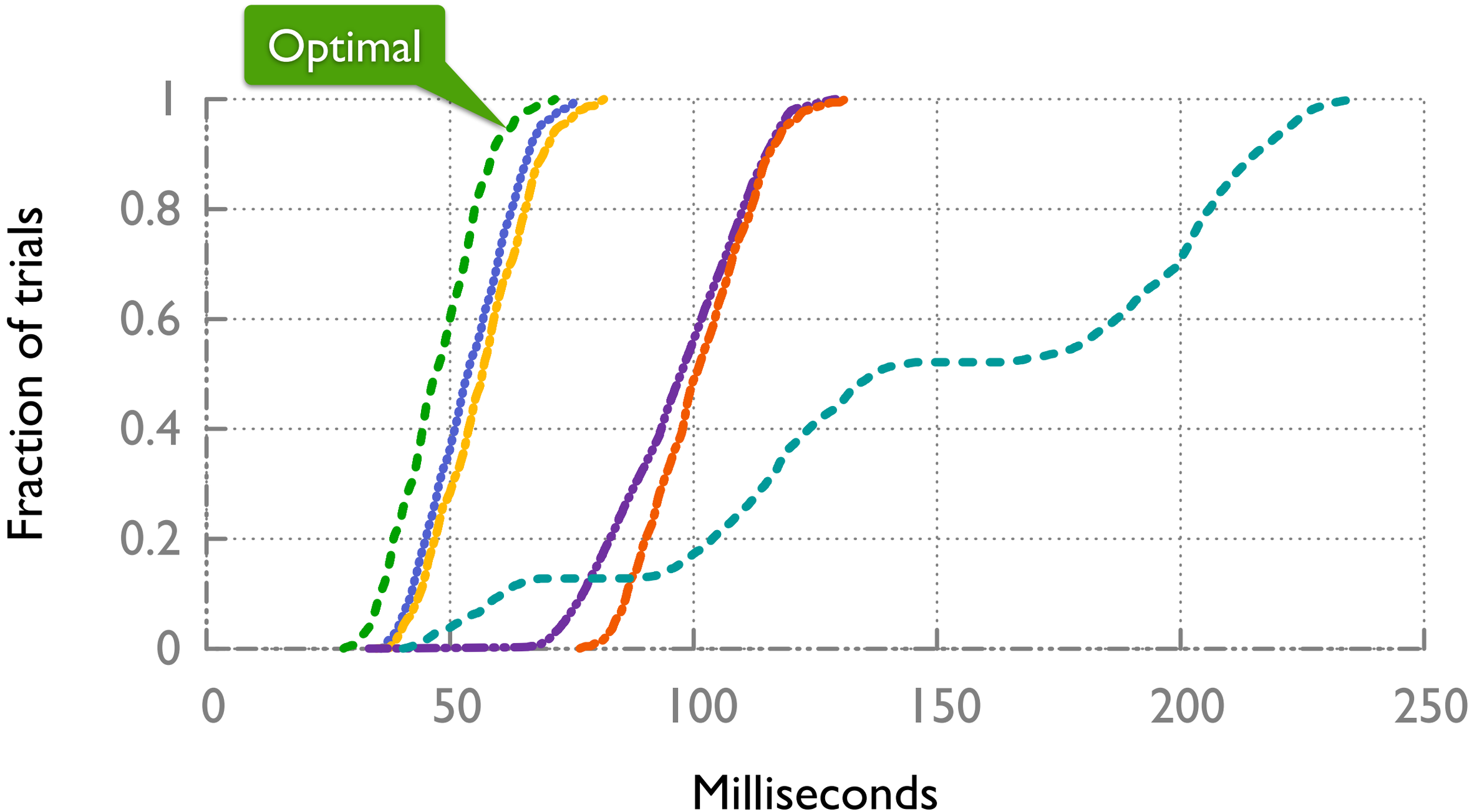
Measure: path completion time



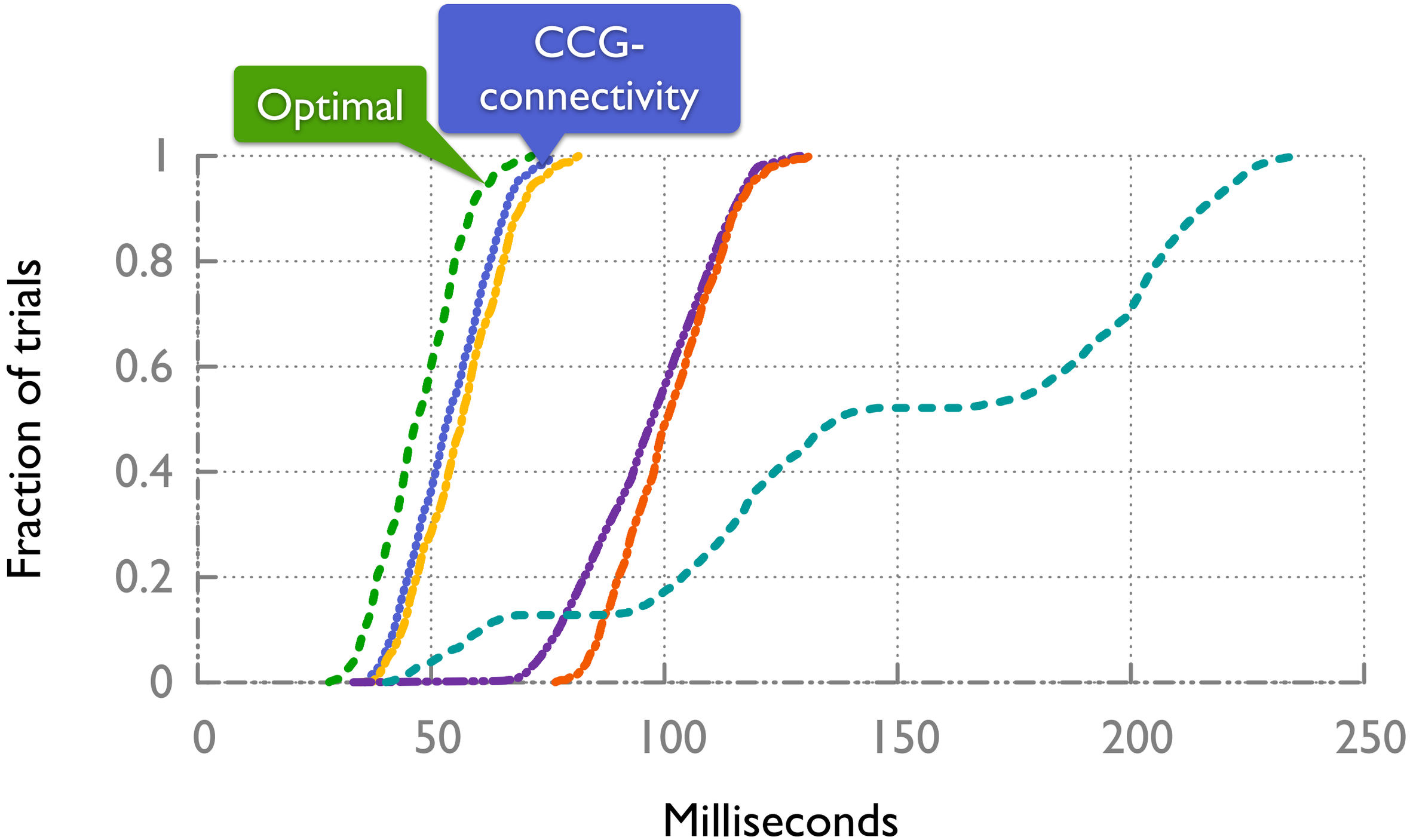
Emulation: Segment-independent Properties



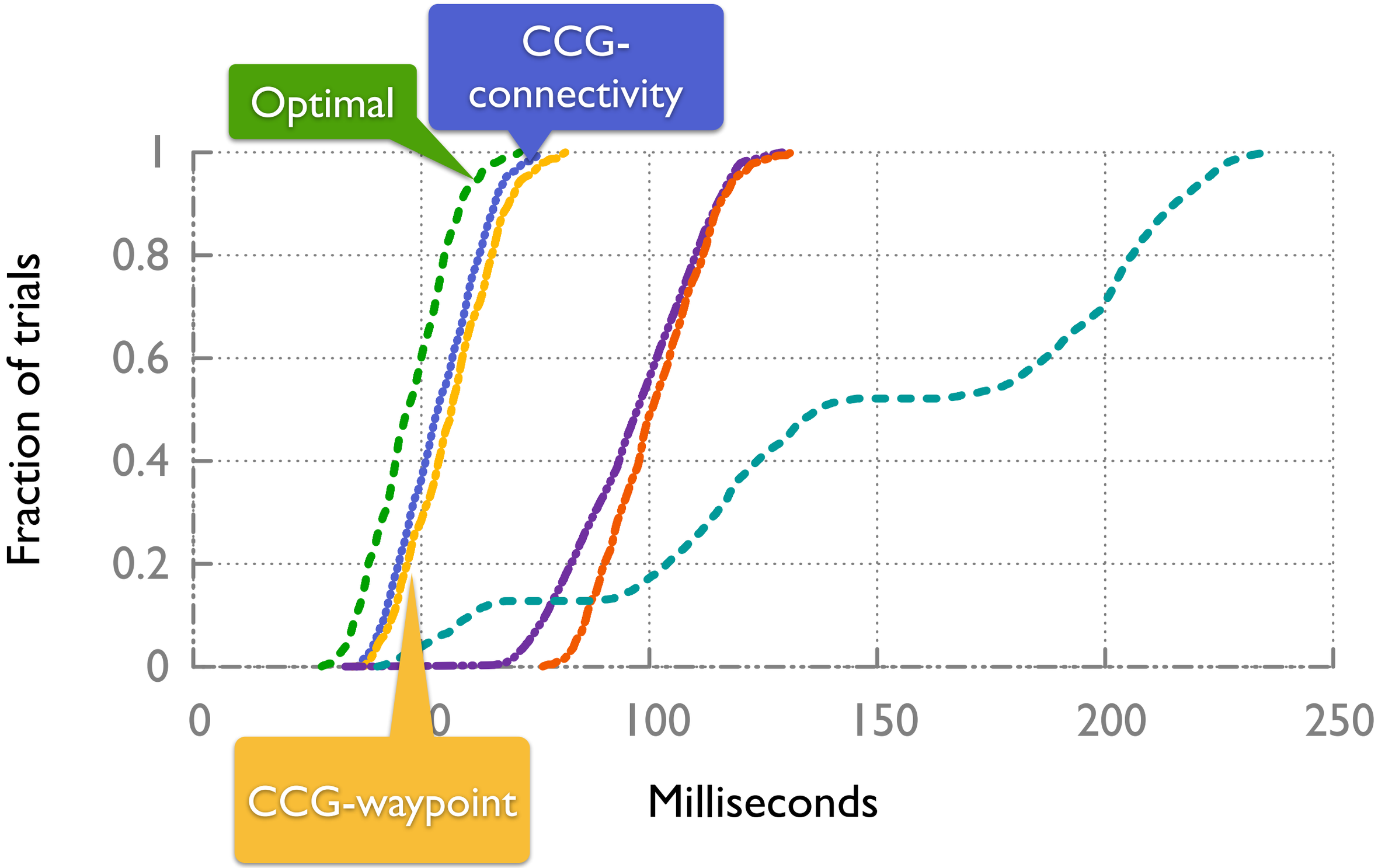
Emulation: Segment-independent Properties



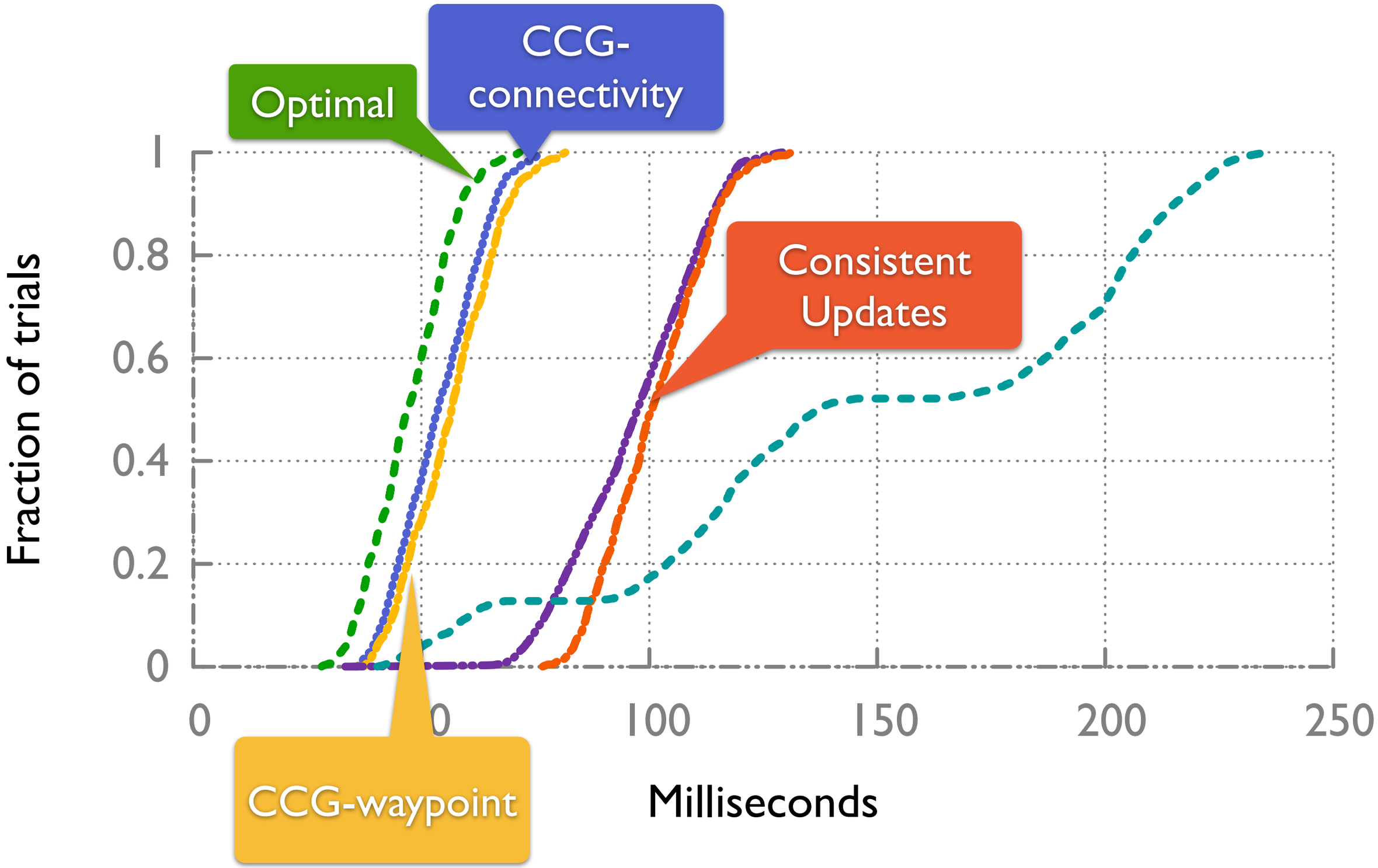
Emulation: Segment-independent Properties



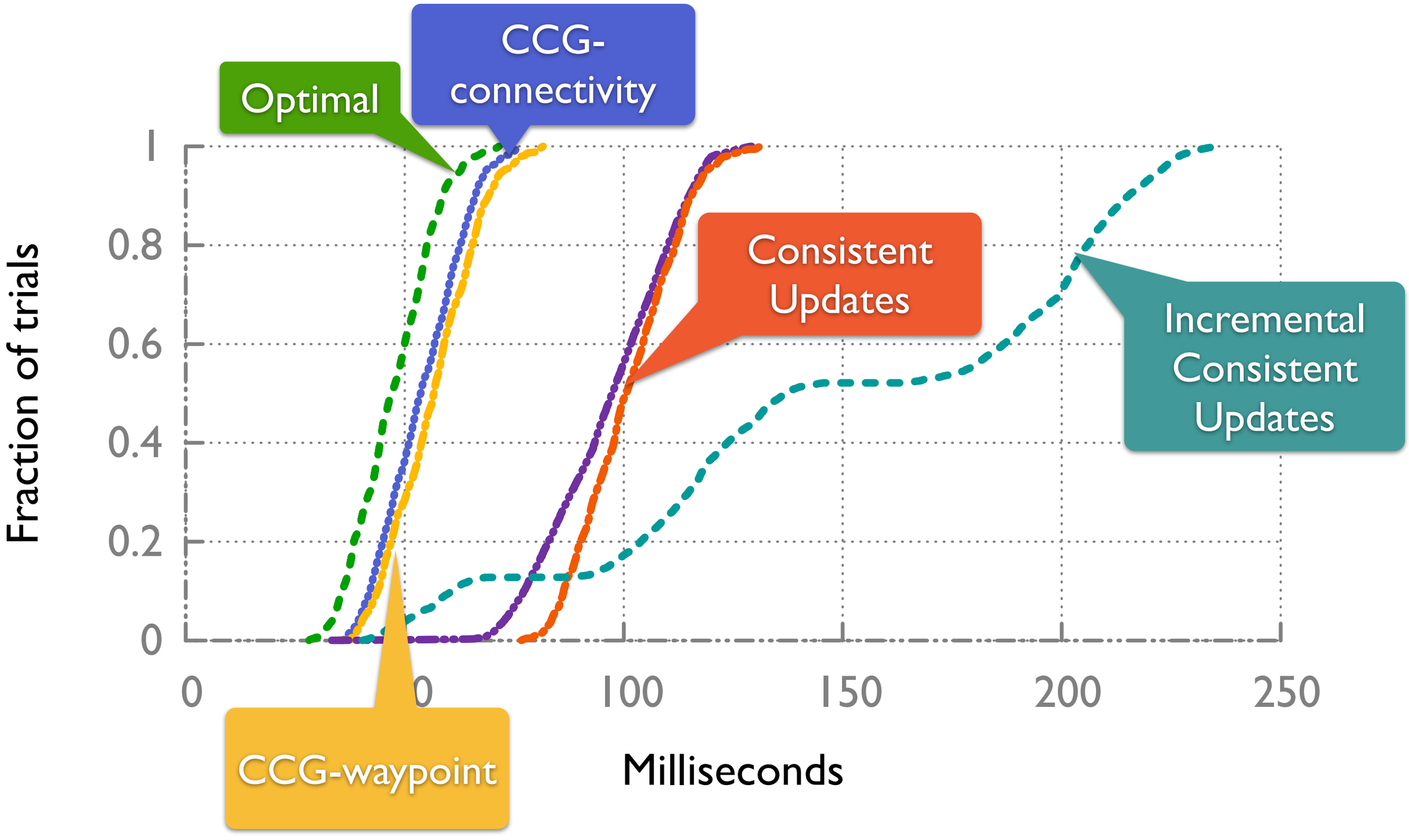
Emulation: Segment-independent Properties



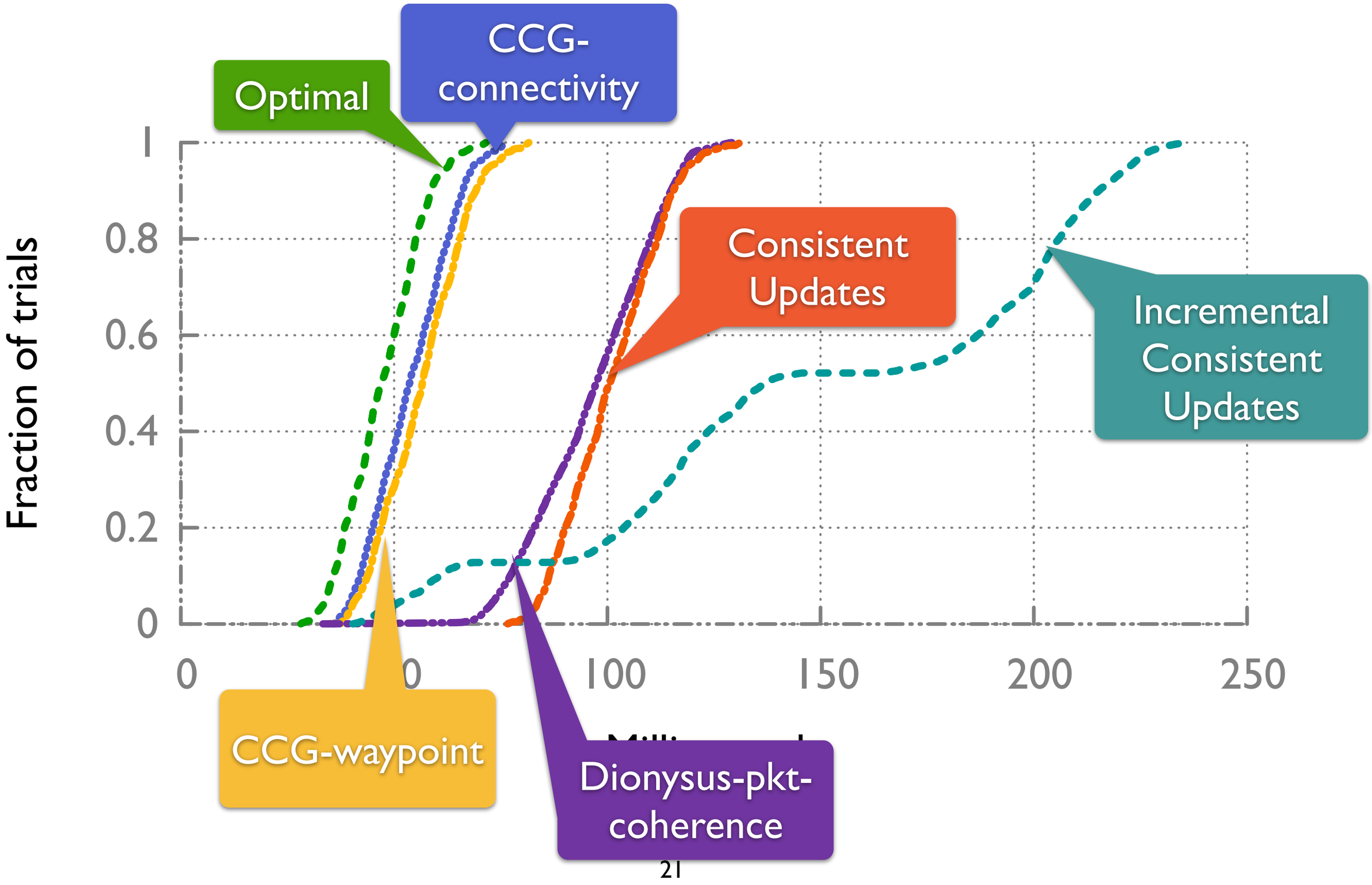
Emulation: Segment-independent Properties



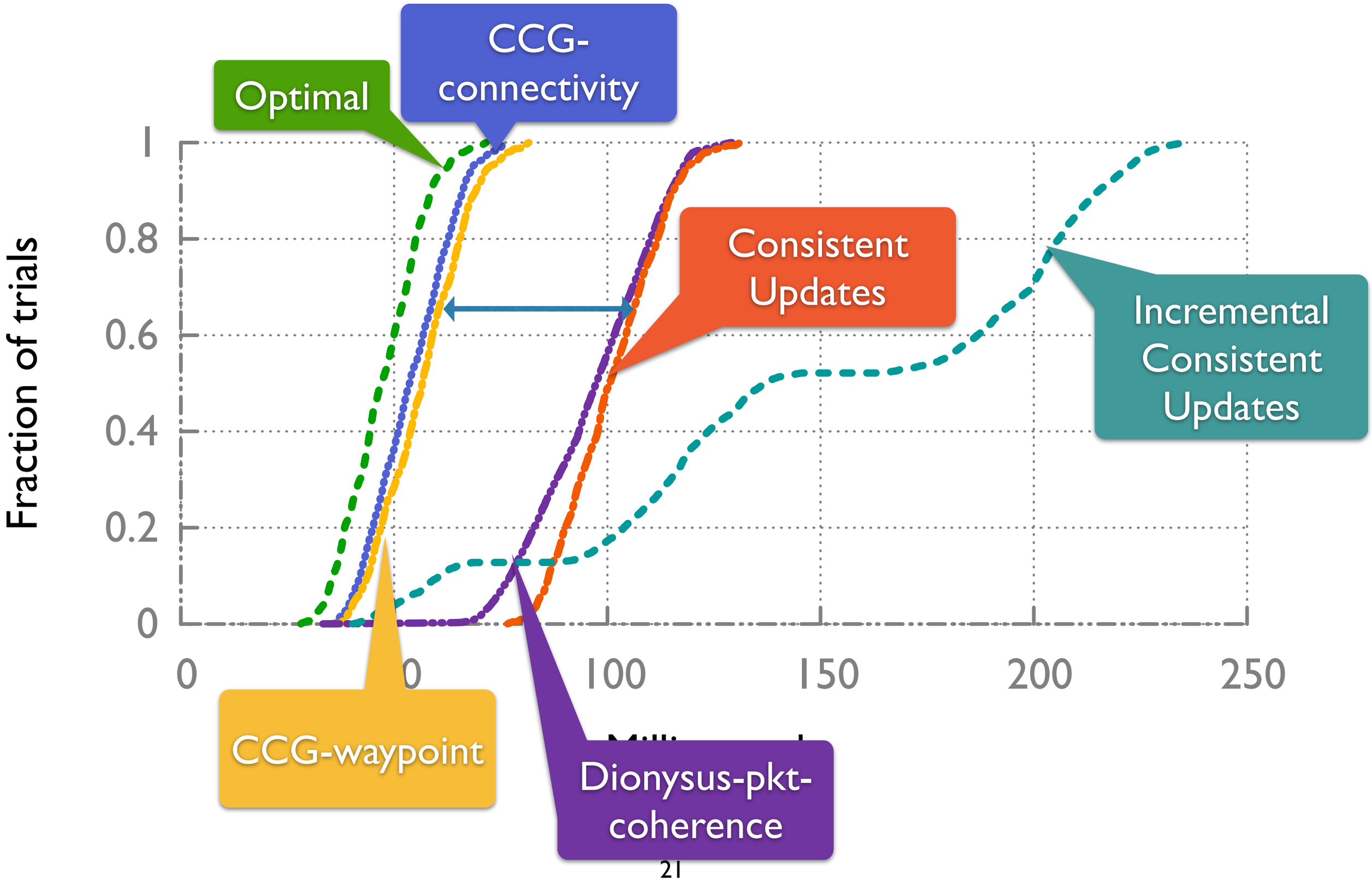
Emulation: Segment-independent Properties



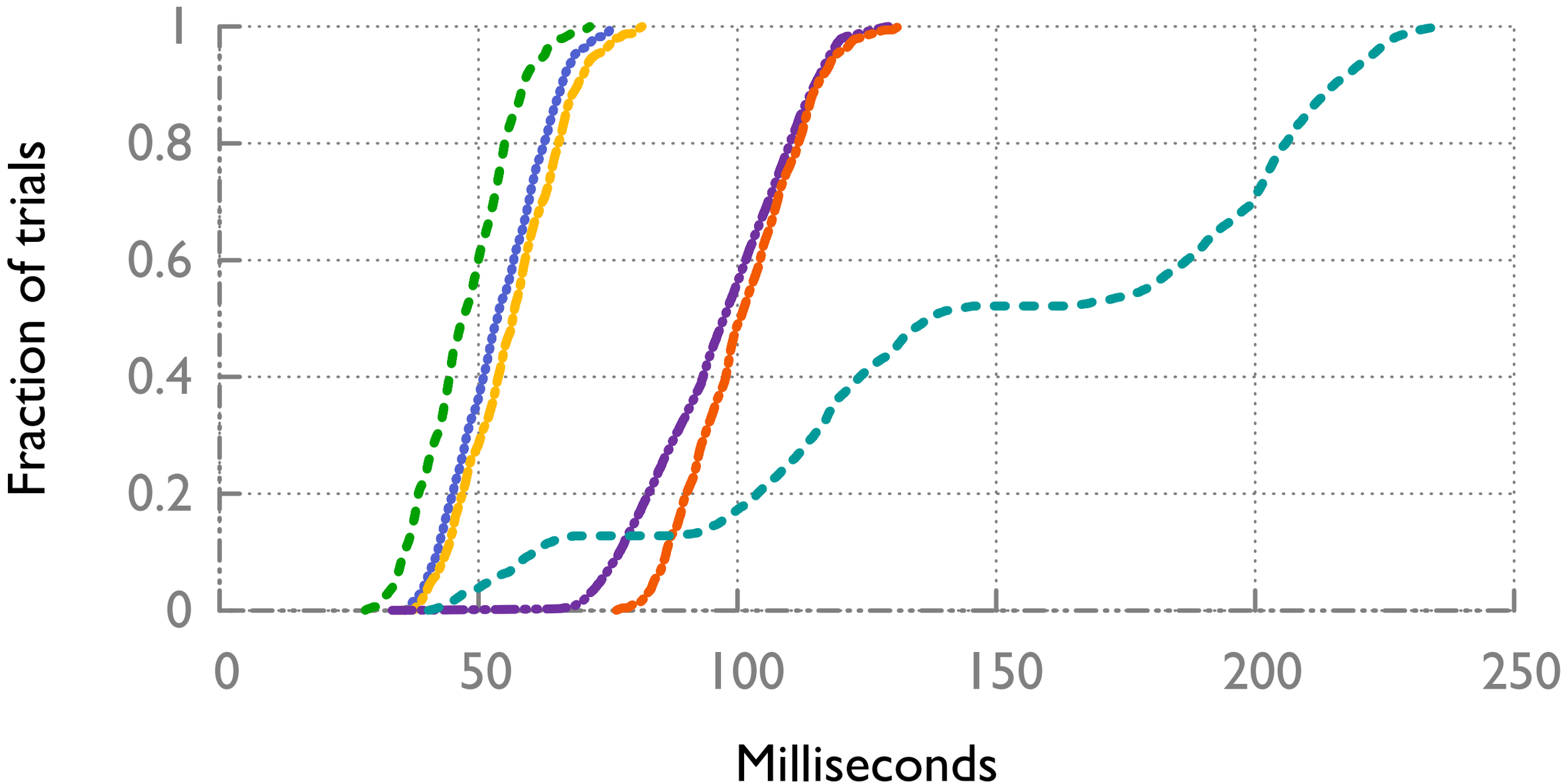
Emulation: Segment-independent Properties



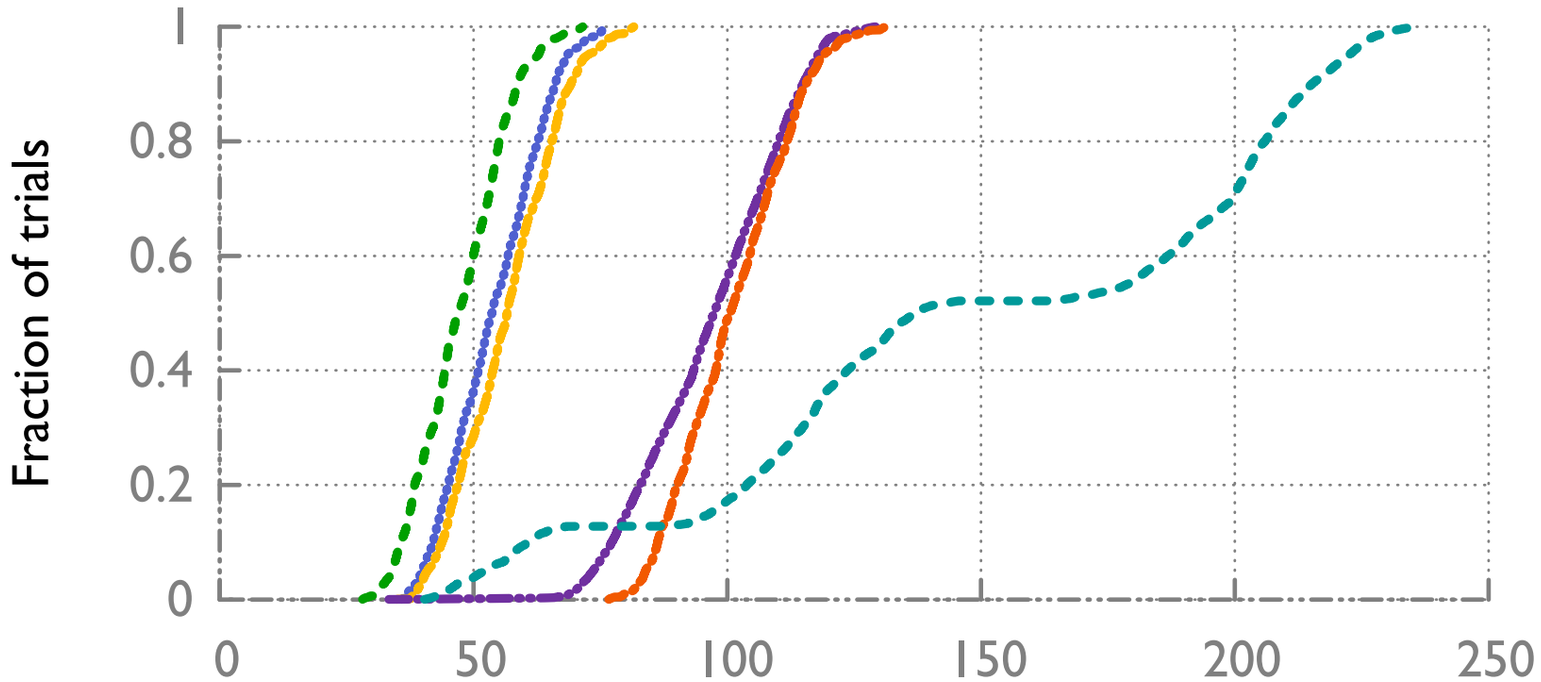
Emulation: Segment-independent Properties



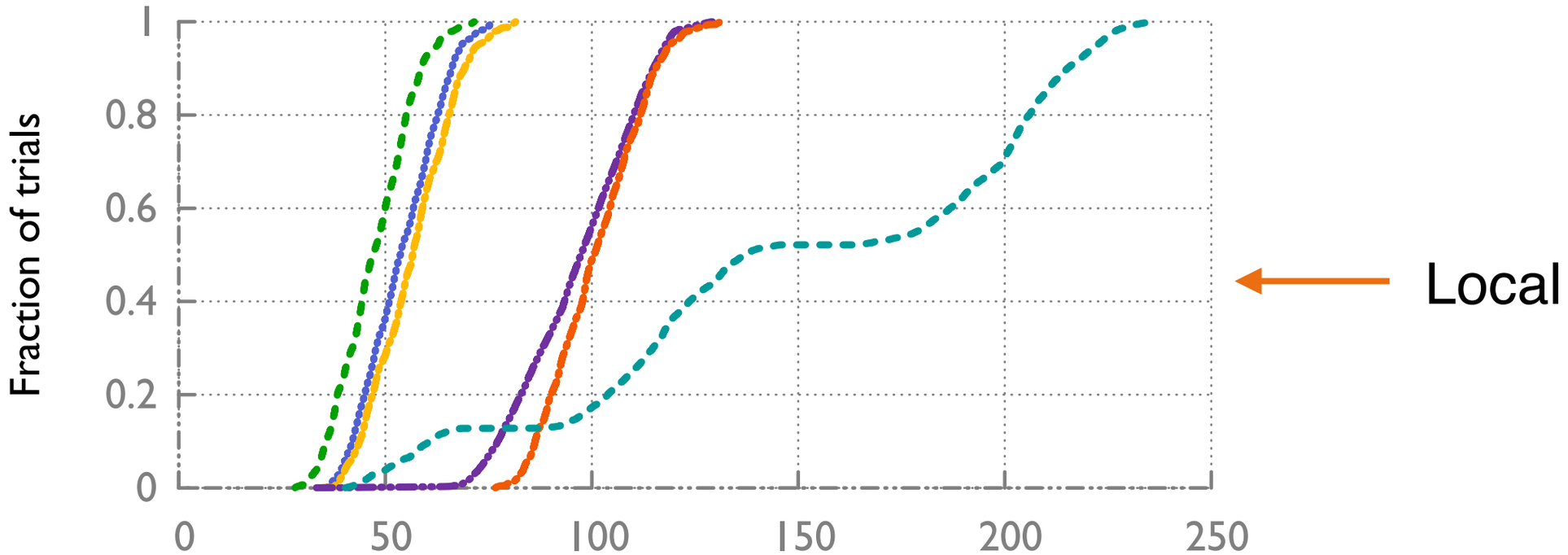
Emulation: Segment-independent Properties



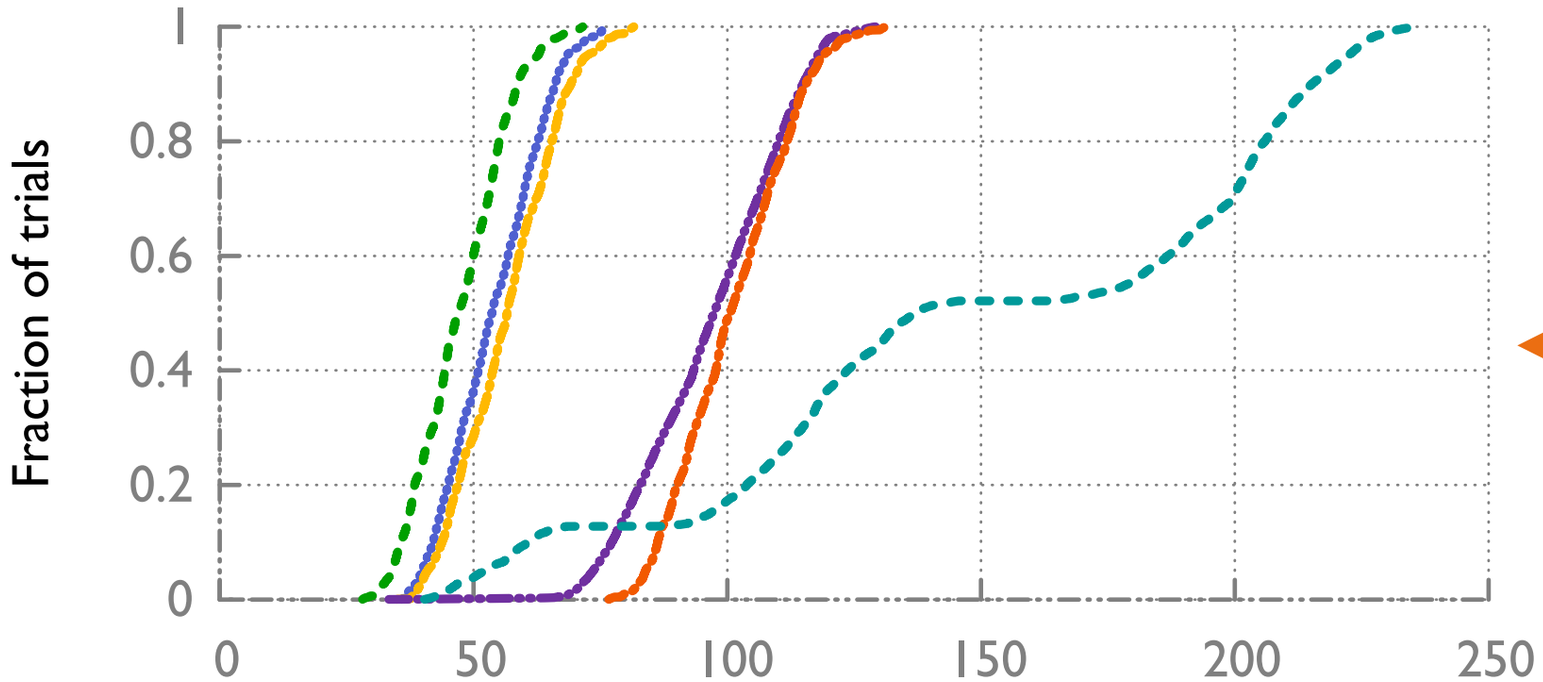
Emulation: Segment-independent Properties



Emulation: Segment-independent Properties



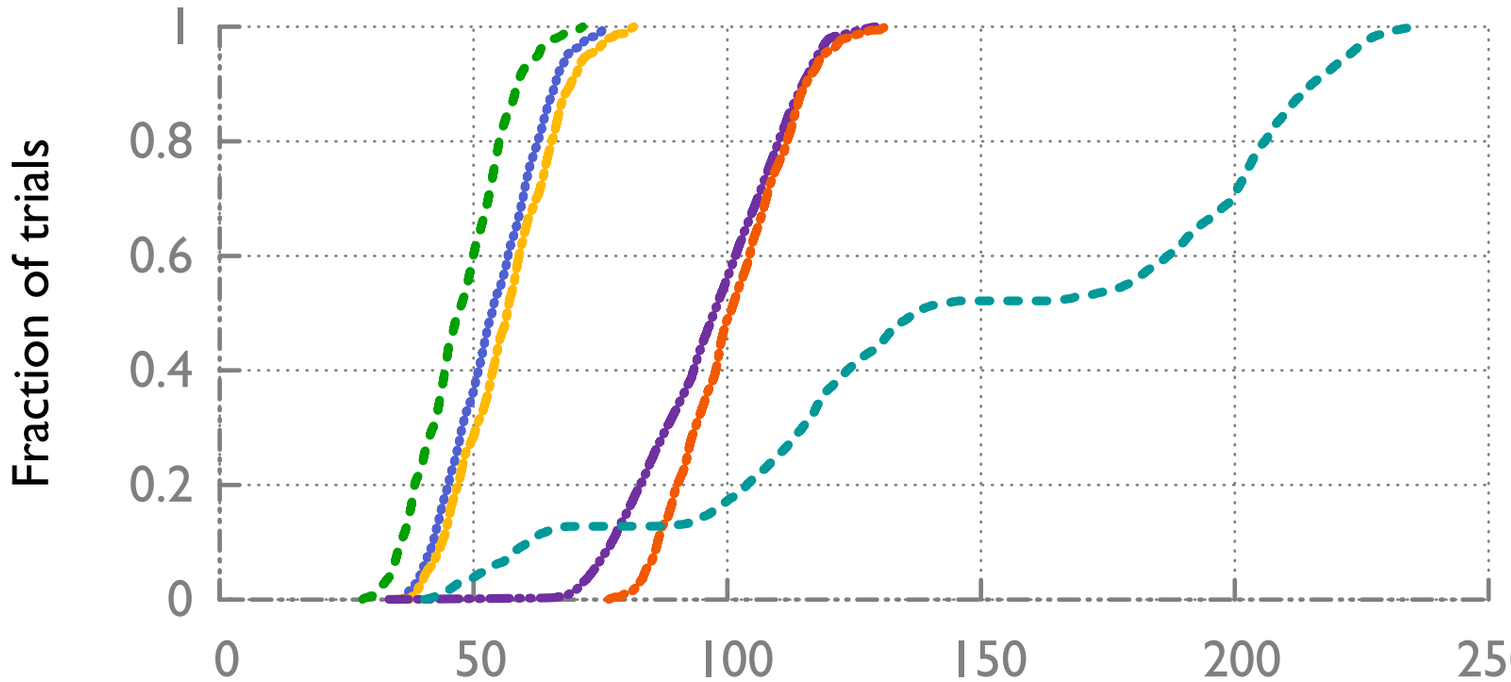
Emulation: Segment-independent Properties



No fallback triggered
No additional memory

Local

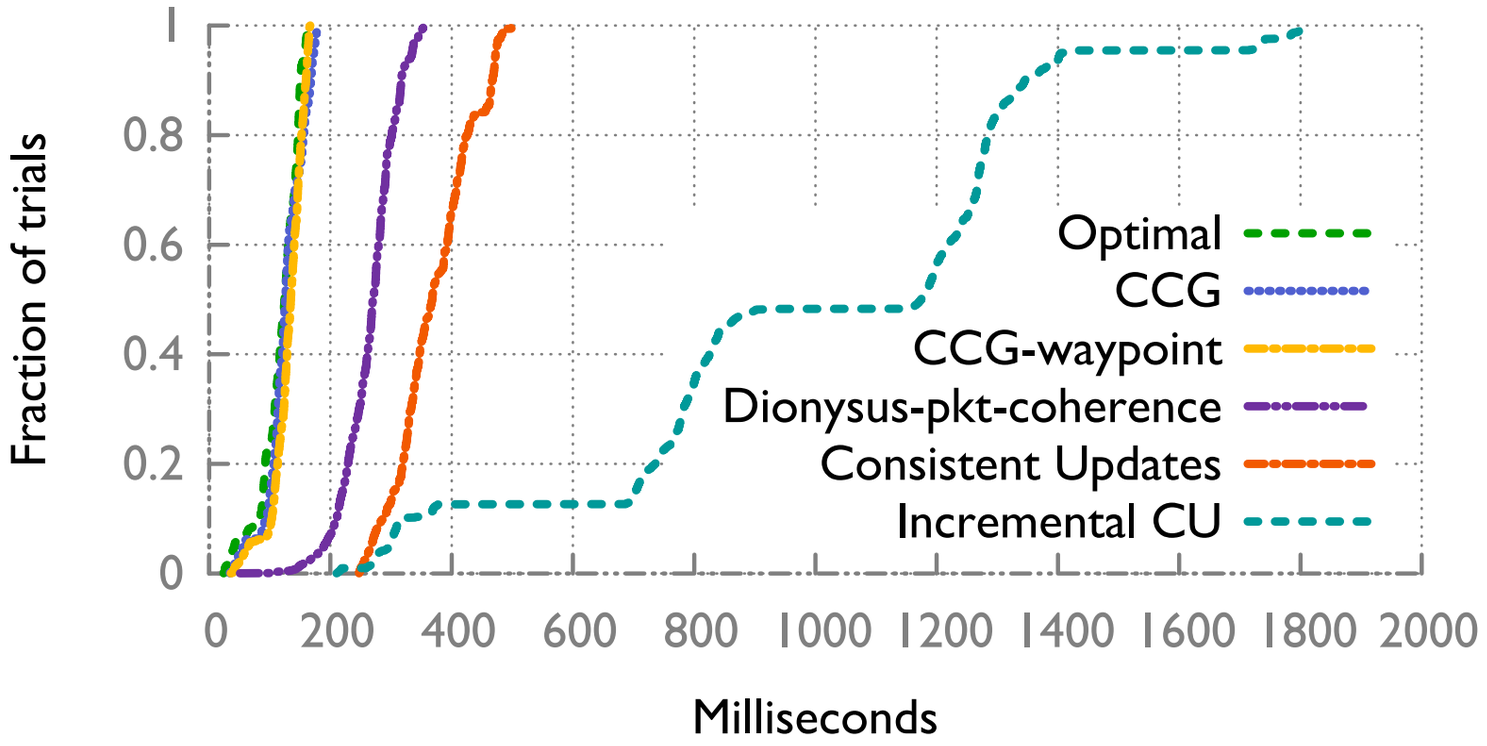
Emulation: Segment-independent Properties



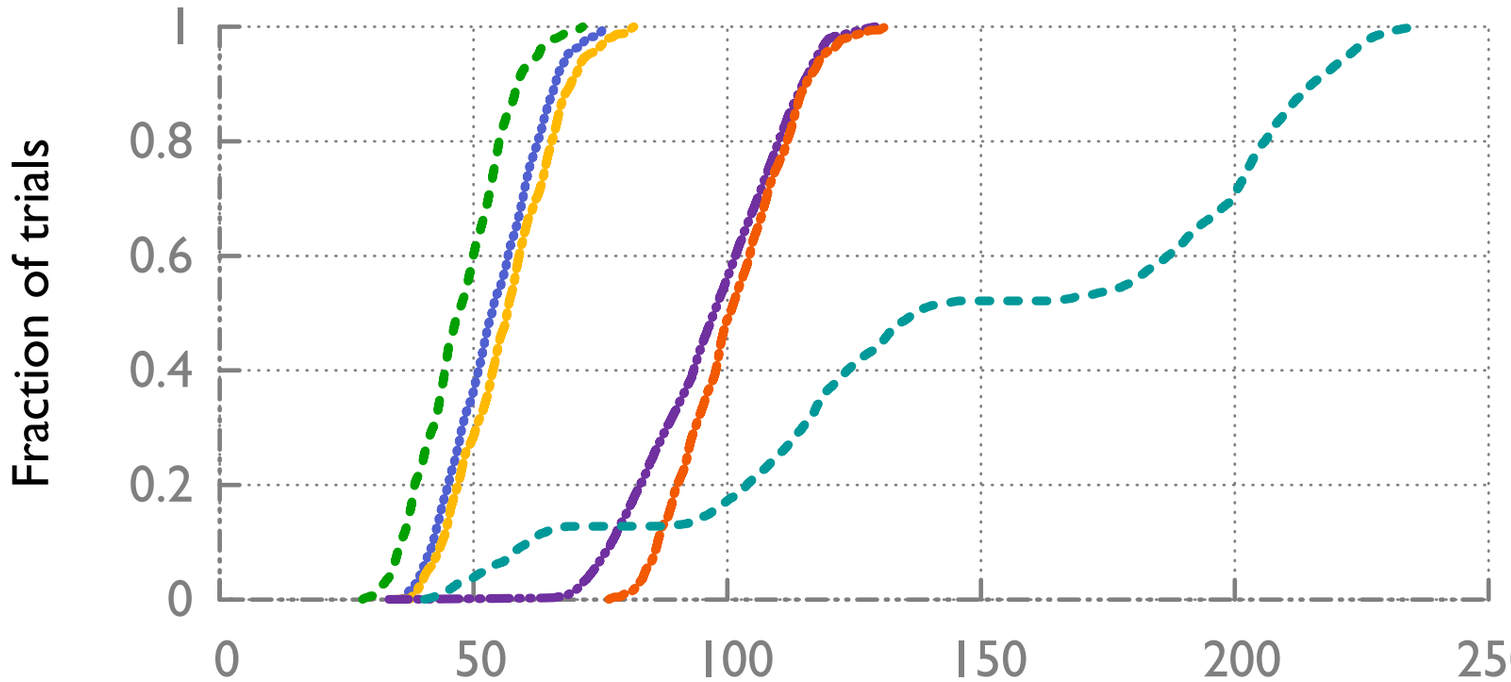
No fallback triggered
No additional memory

Local

Wide area



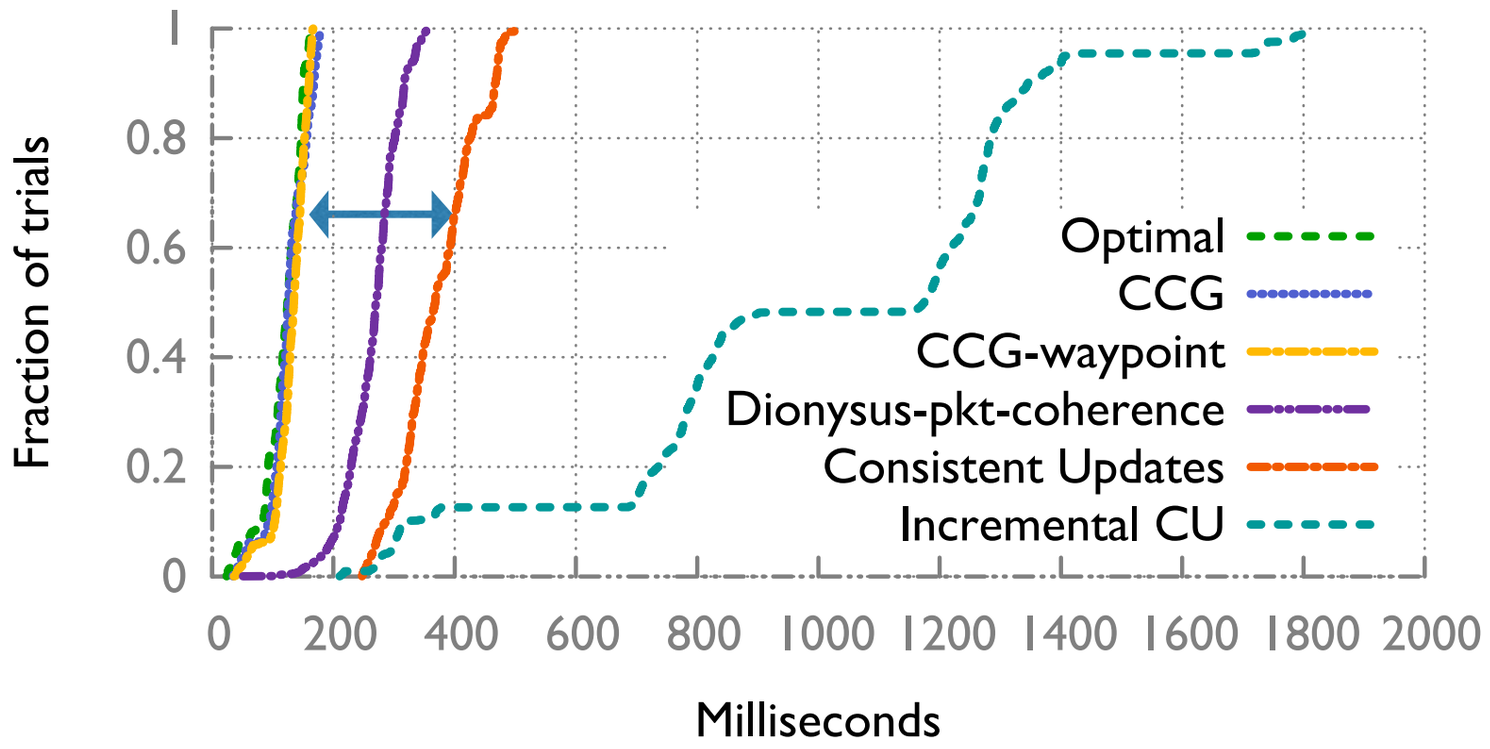
Emulation: Segment-independent Properties



No fallback triggered
No additional memory

Local

Wide area



Emulation: Non-segment-independent Properties

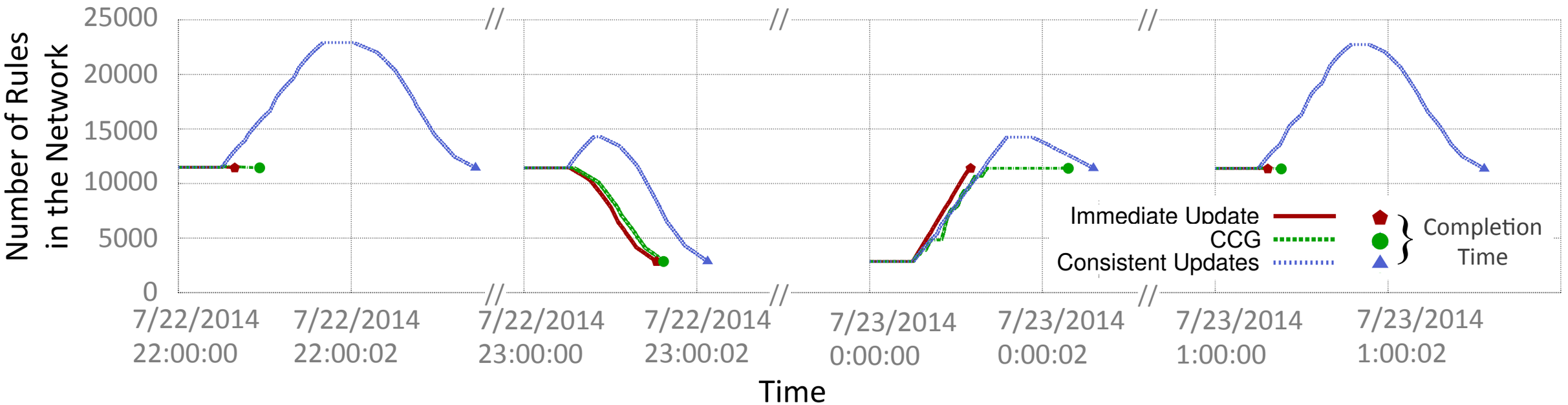
Traces from a operational network with 200+ layer-3 devices.

One day, one snapshot per hour, 24 transitions, 4ms delay.

- New rules were added first, then old rules deleted.

Properties: Black hole freedom + Loop freedom

- Rules overlapped with longest prefix match, not segment-independent.



Emulation: Non-segment-independent Properties

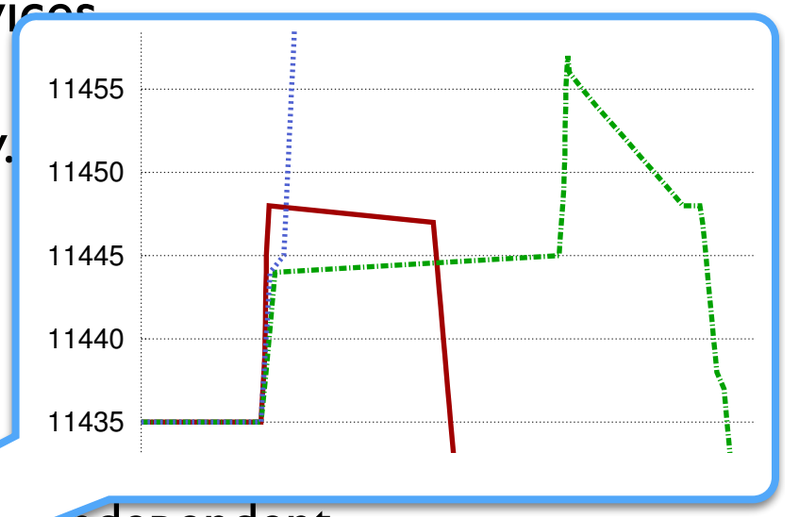
Traces from a operational network with 200+ layer-3 devices

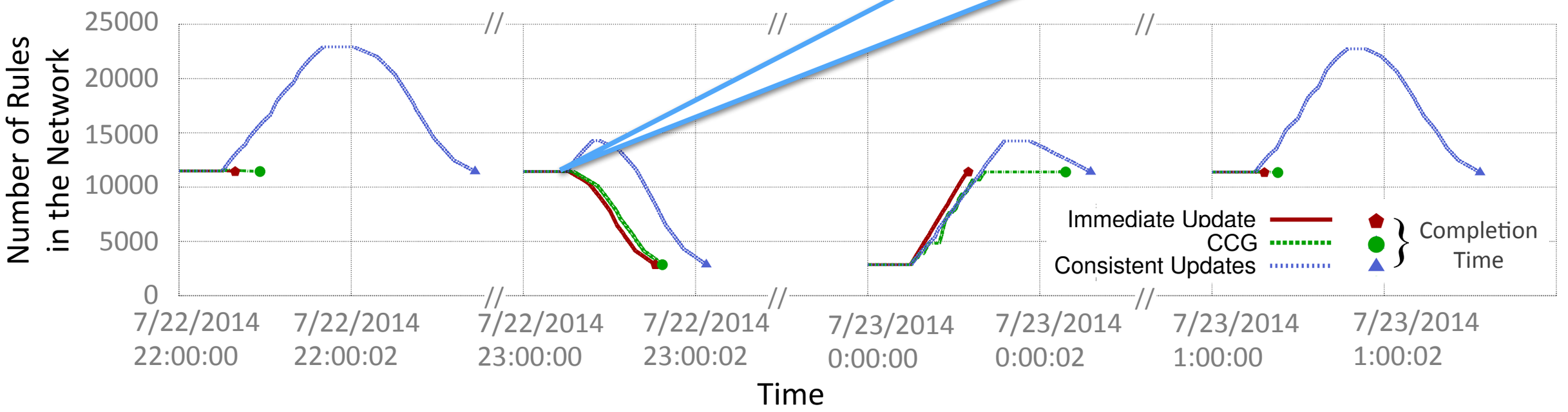
One day, one snapshot per hour, 24 transitions, 4ms delay.

- New rules were added first, then old rules deleted.

Properties: Black hole freedom + Loop freedom

- Rules overlapped with longest prefix match, not segment-independent.





Emulation: Non-segment-independent Properties

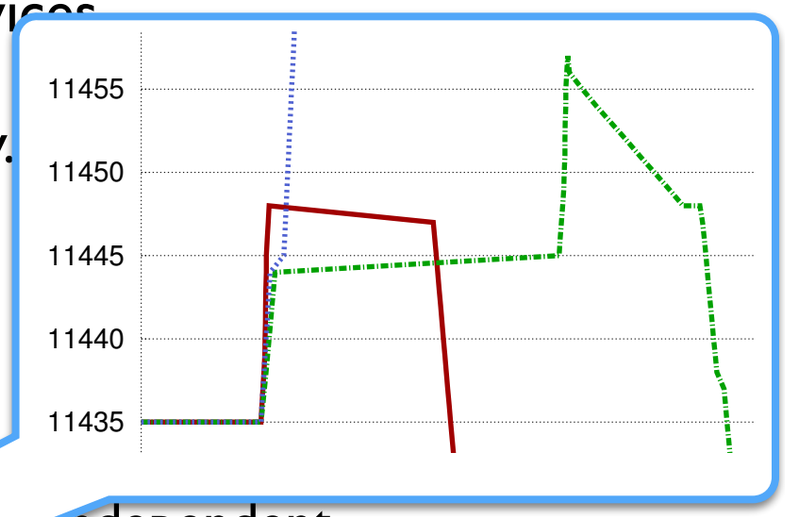
Traces from a operational network with 200+ layer-3 devices.

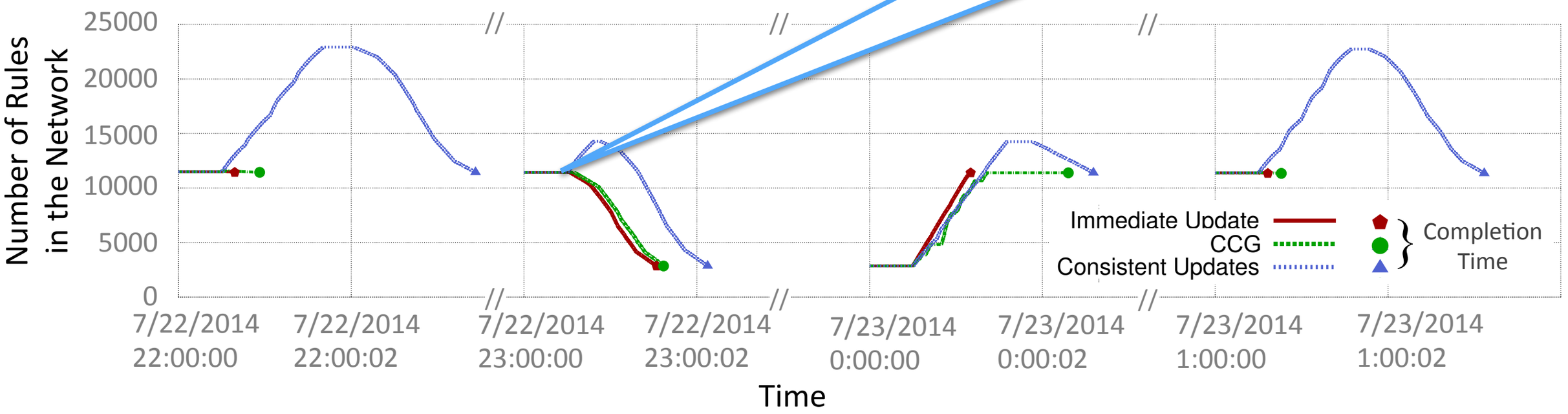
One day, one snapshot per hour, 24 transitions, 4ms delay.

- New rules were added first, then old rules deleted.

Properties: Black hole freedom + Loop freedom

- Rules overlapped with longest prefix match, not segment-independent.





- *Fallbacks happened rarely.*

- *Overhead close to Immediate Update, with no transient connectivity violations.*

Conclusion

Conclusion

CCG, a system that

Conclusion

CCG, a system that

- enforces customizable network consistency properties with

Conclusion

CCG, a system that

- enforces customizable network consistency properties with
- heuristically optimized efficiency.

Conclusion

CCG, a system that

- enforces customizable network consistency properties with
- heuristically optimized efficiency.

Conclusion

CCG, a system that

- enforces customizable network consistency properties with
- heuristically optimized efficiency.

Ongoing work:

Conclusion

CCG, a system that

- enforces customizable network consistency properties with
- heuristically optimized efficiency.

Ongoing work:

- Study generality of *segment independency*

Conclusion

CCG, a system that

- enforces customizable network consistency properties with
- heuristically optimized efficiency.

Ongoing work:

- Study generality of *segment independency*
- Handle network-initiated changes

Conclusion

CCG, a system that

- enforces customizable network consistency properties with
- heuristically optimized efficiency.

Ongoing work:

- Study generality of *segment independency*
- Handle network-initiated changes
- …

Conclusion

CCG, a system that

- enforces customizable network consistency properties with
- heuristically optimized efficiency.

Ongoing work:

- Study generality of *segment independency*
- Handle network-initiated changes
- …