Kulfi

Robust Traffic Engineering Using Semi-Oblivious Routing

Praveen Kumar, Yang Yuan, Chris Yu,
Bobby Kleinberg, Robert Soulé, & Nate Foster

Cornell, Carnegie Mellon, Microsoft Research, & Lugano

Kulfi

Robust Traffic Engineering Using Semi-Oblivious Routing

Praveen Kumar, Yang Yuan, Chris Yu,
Bobby Kleinberg, Robert Soulé, & Nate Foster

Cornell, Carnegie Mellon, Microsoft Research, & Lugano

NetKAT

Probabilistic NetKAT

Nate Foster[1], Dexter Kozen[1], Konstantinos Mamouras[2*], Mark Reitblatt[3*], and Alexandra Silva[4]

[1] Cornell University
[2] University of Pennsylvania
[3] Facebook
[4] University College London

Abstract. This paper presents a new language for network programming based on a probabilistic semantics. We extend the NetKAT language with new primitives for expressing probabilistic behaviors and enrich the semantics from one based on deterministic functions to one based on measurable functions on sets of packet histories. We establish fundamental properties of the semantics, prove that it is a conservative extension of the deterministic semantics, show that it satisfies a number of natural equations, and develop a notion of approximation. We present case studies that show how the language can be used to model a diverse collection of scenarios drawn from real-world networks.

1 Introduction

Formal specification and verification of networks has become a reality in recent years with the emergence of network-specific programming languages and property-checking tools. Programming languages like Frenetic [11], Pyretic [36], Maple [52], FlowLog [38], and others are enabling programmers to specify the intended behavior of a network in terms of high-level constructs such as Boolean predicates and functions on packets. Verification tools like Header Space Analysis [21], VeriFlow [22], and NetKAT [12] are making it possible to check properties such as connectivity, loop freedom, and traffic isolation automatically.

However, despite many notable advances, these frameworks all have a fundamental limitation: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network functionality is simple, or where the properties of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory accounts of more complicated situations that often arise in practice:

– **Congestion:** the network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic.
– **Failure:** the network operator wishes to calculate the probability that packets will be delivered to their destination, given that devices and links fail with a certain probability.

[*] Work performed at Cornell University.

Event-Driven Network Programming

Jedidiah McClurg, CU Boulder, USA, jedidiah.mcclurg@colorado.edu
Hossein Hojjat, Cornell University, USA, hojjat@cornell.edu
Nate Foster, Cornell University, USA, jnfoster@cs.cornell.edu
Pavol Černý, CU Boulder, USA, pavol.cerny@colorado.edu

arXiv:1507.07049v3 [cs.PL] 16 Apr 2016

Abstract

Software-defined networking (SDN) programs must simultaneously describe static forwarding behavior and dynamic updates in response to events. Event-driven updates are critical to get right, but difficult to implement correctly due to the high degree of concurrency in networks. Existing SDN platforms offer weak guarantees that can break application invariants, leading to problems such as dropped packets, degraded performance, security violations, etc. This paper introduces *event-driven consistent updates* that are guaranteed to preserve well-defined behaviors when transitioning between configurations in response to events. We propose *network event structures* (NESs) to model constraints on updates, such as which events can be enabled simultaneously and causal dependencies between events. We define an extension of the NetKAT language with mutable state, give semantics to stateful programs using NESs, and discuss provably-correct strategies for implementing NESs in SDNs. Finally, we evaluate our approach empirically, demonstrating that it gives well-defined consistency guarantees while avoiding expensive synchronization and packet buffering.

Categories and Subject Descriptors C.2.3 [*Computer-communication Networks*]: Network Operations—Network Management; D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords network update, consistent update, event structure, software-defined networking, SDN, NetKAT

1. Introduction

Software-defined networking (SDN) allows network behavior to be specified using logically-centralized programs that execute on general-purpose machines. These programs react to events such as topology changes, traffic statistics, receipt of packets, etc. by modifying sets of forwarding rules installed on switches. SDN programs can implement a wide range of advanced network functionality including fine-grained access control [8], network virtualization [22], traffic engineering [15, 16], and many others.

Although the basic SDN model is simple, building sophisticated applications is challenging in practice. Programmers must keep track of numerous low-level details such as encoding configurations into prioritized forwarding rules, processing concurrent events, managing asynchronous events, dealing with unexpected failures, etc. To address these challenges, a number of domain-specific network programming languages have been proposed [2, 10, 19, 21, 29, 31, 36, 37]. The details of these languages vary, but they all offer higher-level abstractions for specifying behavior (e.g., using mathematical functions, boolean predicates, relational operators, etc.), and rely on a compiler and run-time system to generate and manage the underlying network state.

Unfortunately, the languages that have been proposed so far lack critical features that are needed to implement dynamic, event-driven applications. Static languages such as NetKAT [2] offer rich constructs for describing network configurations, but lack features for responding to events and maintaining internal state. Instead, programmers must write a stateful program in a general-purpose language that generates a stream of NetKAT programs. Dynamic languages such as FlowLog and Kinetic [21, 31] offer stateful programming models, but they do not specify how the network behaves while it is being reconfigured in response to state changes. Abstractions such as consistent updates provide strong guarantees during periods of reconfiguration [26, 33], but current realizations are limited to properties involving a single packet (or set of related packets, such as a unidirectional flow). To implement *correct* dynamic SDN applications today, the most effective option is often to use low-level APIs, forgoing the benefits of higher-level languages entirely.

Example: Stateful Firewall. To illustrate the challenges that arise when implementing dynamic applications, consider a topology where an internal host H_1 is connected to switch s_1, an external host H_4 is connected to a switch s_4, and switches s_1 and s_4 are connected to each other (see Fig-

[ESOP '16]

[PLDI '16]

NetKAT


Probabilistic NetKAT

Nate Foster[1], Dexter Kozen[1], Konstantinos Mamouras[2*], Mark Reitblatt[3*], and Alexandra Silva[4]

[1] Cornell University
[2] University of Pennsylvania
[3] Facebook
[4] University College London

Abstract. This paper presents a new language for network programming based on a probabilistic semantics. We extend the NetKAT language with new primitives for expressing probabilistic behaviors and enrich the semantics from one based on deterministic functions to one based on measurable functions on sets of packet histories. We establish fundamental properties of the semantics, prove that it is a conservative extension of the deterministic semantics, show that it satisfies a number of natural equations, and develop a notion of approximation. We present case studies that show how the language can be used to model a diverse collection of scenarios drawn from real-world networks.

1 Introduction

Formal specification and verification of networks has become a reality in recent years with the emergence of network-specific programming languages and property-checking tools. Programming languages like Frenetic [11], Pyretic [36], Maple [52], FlowLog [38], and others are enabling programmers to specify the intended behavior of a network in terms of high-level constructs such as Boolean predicates and functions on packets. Verification tools like Header Space Analysis [21], VeriFlow [22], and NetKAT [12] are making it possible to check properties such as connectivity, loop freedom, and traffic isolation automatically.

However, despite many notable advances, these frameworks all have a fundamental limitation: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network functionality is simple, or where the properties of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory accounts of more complicated situations that often arise in practice:

– **Congestion:** the network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic.
– **Failure:** the network operator wishes to calculate the probability that packets will be delivered to their destination, given that devices and links fail with a certain probability.

* Work performed at Cornell University.

[ESOP '16]


Event-Driven Network Programming

Jedidiah McClurg, CU Boulder, USA, jedidiah.mcclurg@colorado.edu
Hossein Hojjat, Cornell University, USA, hojjat@cornell.edu
Nate Foster, Cornell University, USA, jnfoster@cs.cornell.edu
Pavol Černý, CU Boulder, USA, pavol.cerny@colorado.edu

arXiv:1507.07049v3 [cs.PL] 16 Apr 2016

Abstract

Software-defined networking (SDN) programs must simultaneously describe static forwarding behavior and dynamic updates in response to events. Event-driven updates are critical to get right, but difficult to implement correctly due to the high degree of concurrency in networks. Existing SDN platforms offer weak guarantees that can break application invariants, leading to problems such as dropped packets, degraded performance, security violations, etc. This paper introduces *event-driven consistent updates* that are guaranteed to preserve well-defined behaviors when transitioning between configurations in response to events. We propose *network event structures* (NESs) to model constraints on updates, such as which events can be enabled simultaneously and causal dependencies between events. We define an extension of the NetKAT language with mutable state, give semantics to stateful programs using NESs, and discuss provably-correct strategies for implementing NESs in SDNs. Finally, we evaluate our approach empirically, demonstrating that it gives well-defined consistency guarantees while avoiding expensive synchronization and packet buffering.

Categories and Subject Descriptors C.2.3 [*Computer-communication Networks*]: Network Operations—Network Management; D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords network update, consistent update, event structure, software-defined networking, SDN, NetKAT

1. Introduction

Software-defined networking (SDN) allows network behavior to be specified using logically-centralized programs that execute on general-purpose machines. These programs react to events such as topology changes, traffic statistics, receipt of packets, etc. by modifying sets of forwarding rules installed on switches. SDN programs can implement a wide range of advanced network functionality including fine-grained access control [8], network virtualization [22], traffic engineering [15, 16], and many others.

Although the basic SDN model is simple, building sophisticated applications is challenging in practice. Programmers must keep track of numerous low-level details such as encoding configurations into prioritized forwarding rules, processing concurrent events, managing asynchronous events, dealing with unexpected failures, etc. To address these challenges, a number of domain-specific network programming languages have been proposed [2, 10, 19, 21, 29, 31, 36, 37]. The details of these languages vary, but they all offer higher-level abstractions for specifying behavior (e.g., using mathematical functions, boolean predicates, relational operators, etc.), and rely on a compiler and run-time system to generate and manage the underlying network state.

Unfortunately, the languages that have been proposed so far lack critical features that are needed to implement dynamic, event-driven applications. Static languages such as NetKAT [2] offer rich constructs for describing network configurations, but lack features for responding to events and maintaining internal state. Instead, programmers must write a stateful program in a general-purpose language that generates a stream of NetKAT programs. Dynamic languages such as FlowLog and Kinetic [21, 31] offer stateful programming models, but they do not specify how the network behaves while it is being reconfigured in response to state changes. Abstractions such as consistent updates provide strong guarantees during periods of reconfiguration [26, 33], but current realizations are limited to properties involving a single packet (or set of related packets, such as a unidirectional flow). To implement *correct* dynamic SDN applications today, the most effective option is often to use low-level APIs, forgoing the benefits of higher-level languages entirely.

Example: Stateful Firewall. To illustrate the challenges that arise when implementing dynamic applications, consider a topology where an internal host H_1 is connected to switch s_1, an external host H_4 is connected to a switch s_4, and switches s_1 and s_4 are connected to each other (see Fig-

Event-Driven Network Programming 1 2016/4/19

[PLDI '16]

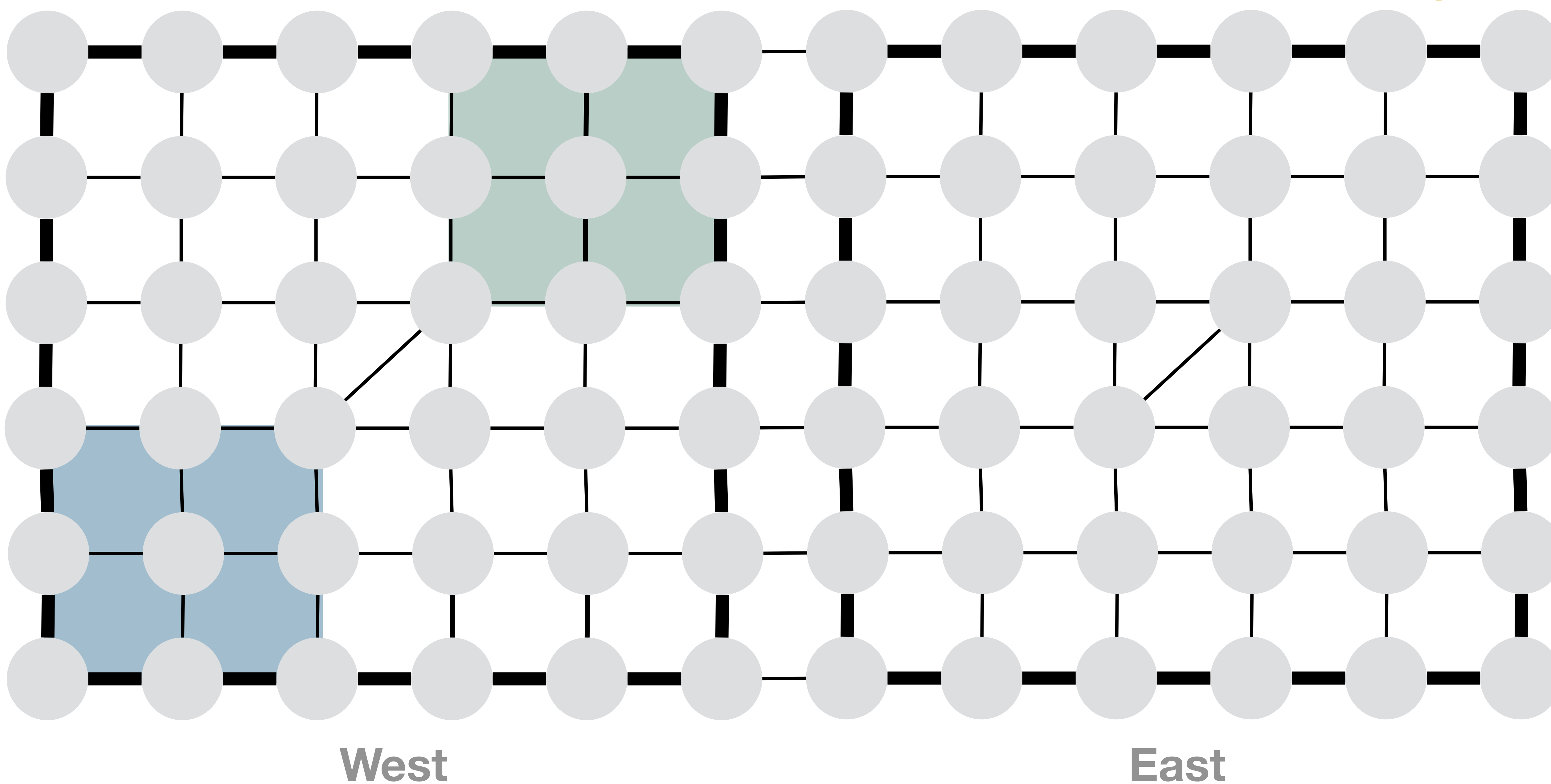
A Bus Ride...

"Why aren't more algorithms researchers working on SDN?"

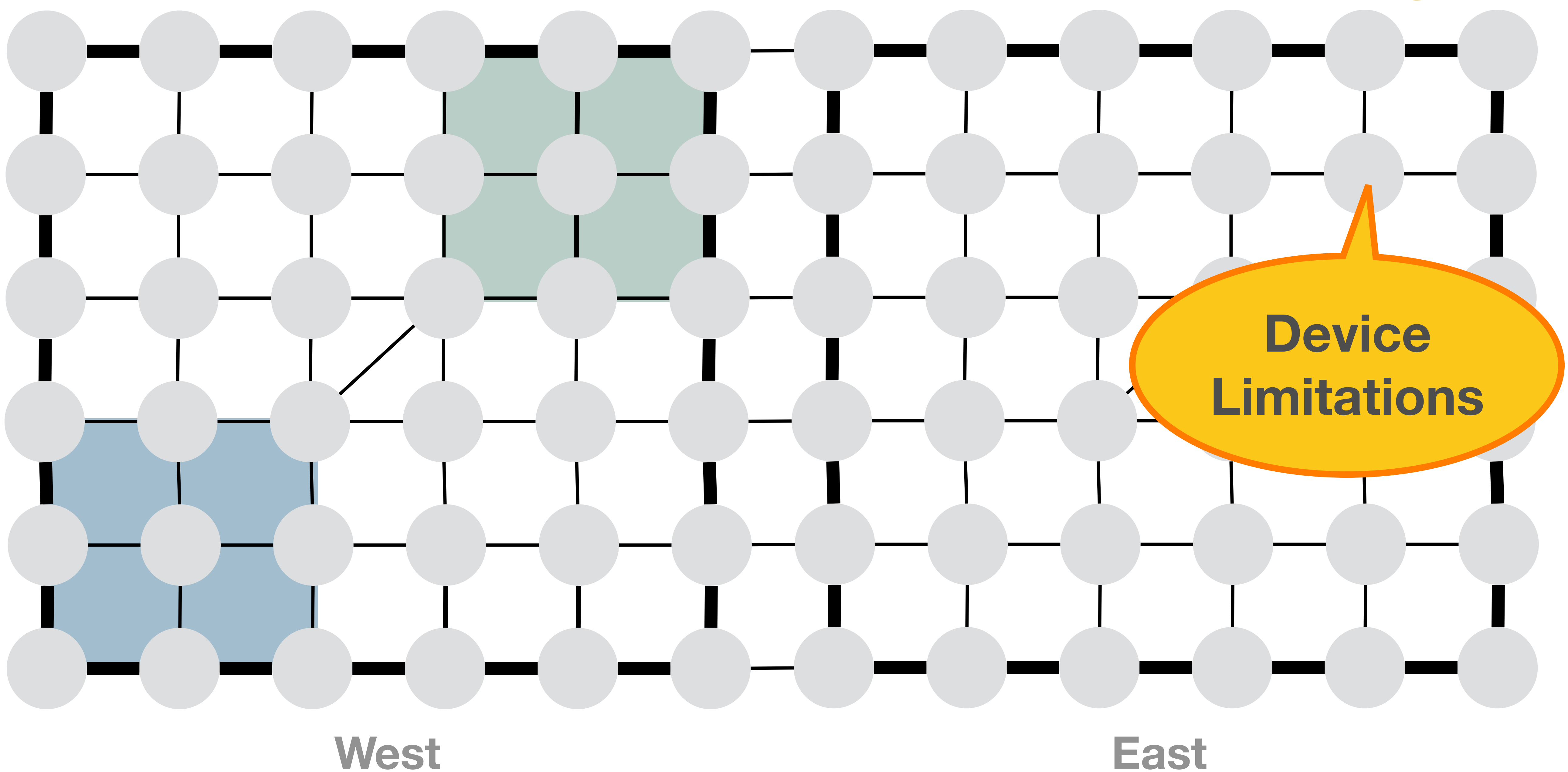
WAN Traffic Engineering

- Network infrastructure is expensive!
- Operators must balance latency-sensitive customer traffic with high-volume, operational traffic
- Many competing objectives:
  - Balances load
  - Achieves low latency
  - Tolerates failures
  - Simple to implement

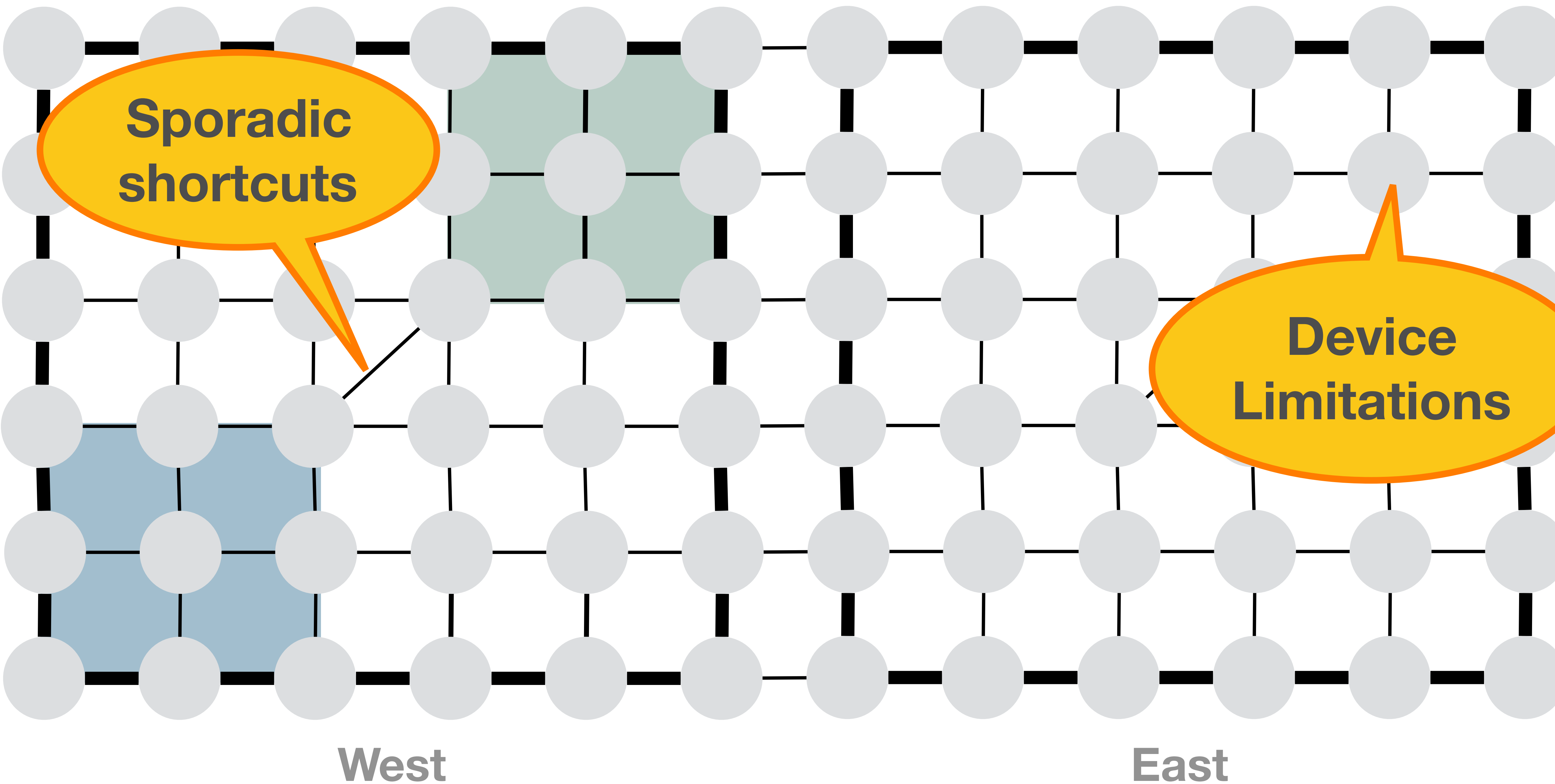
Challenges



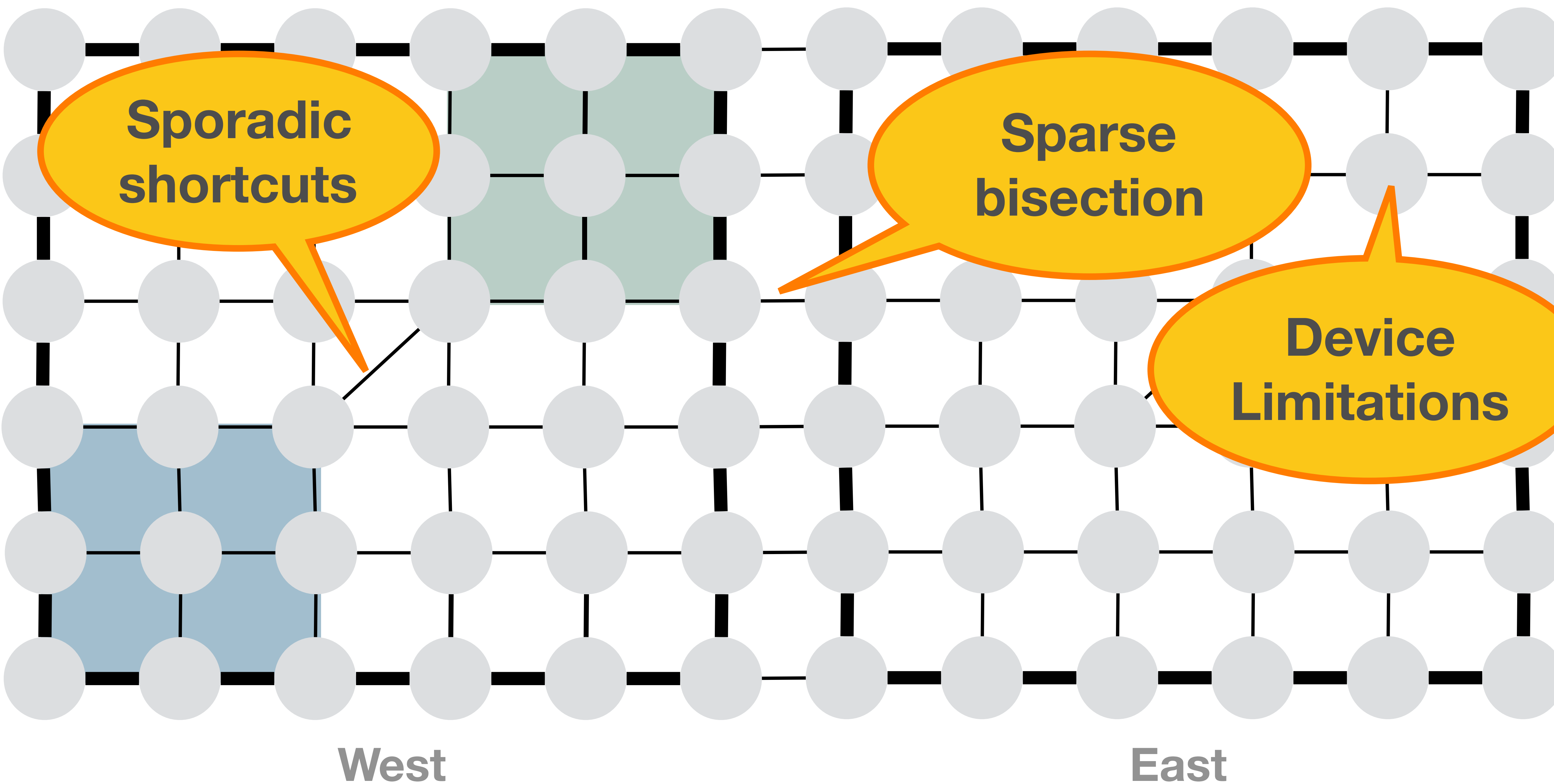
Challenges



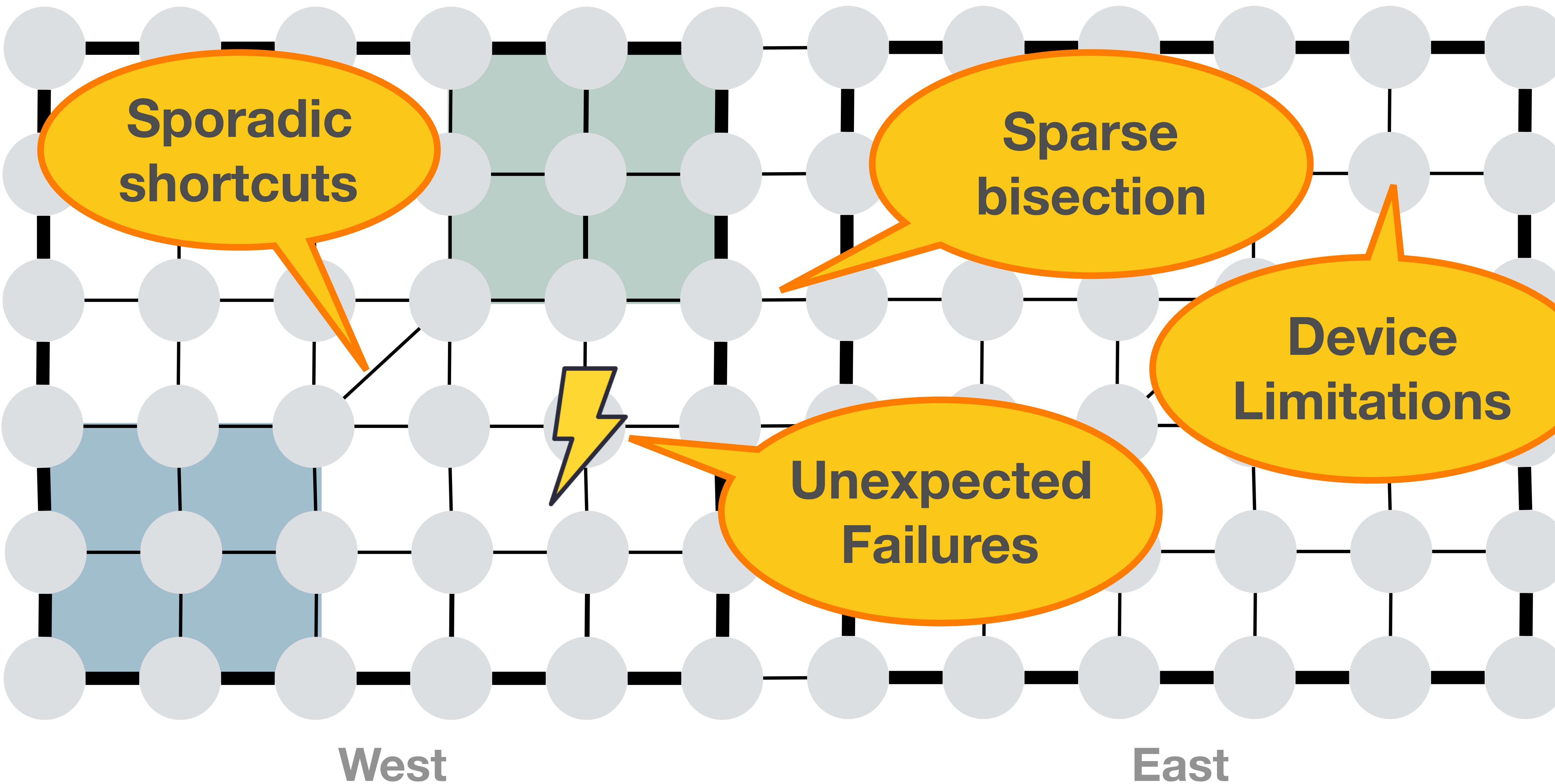
Challenges



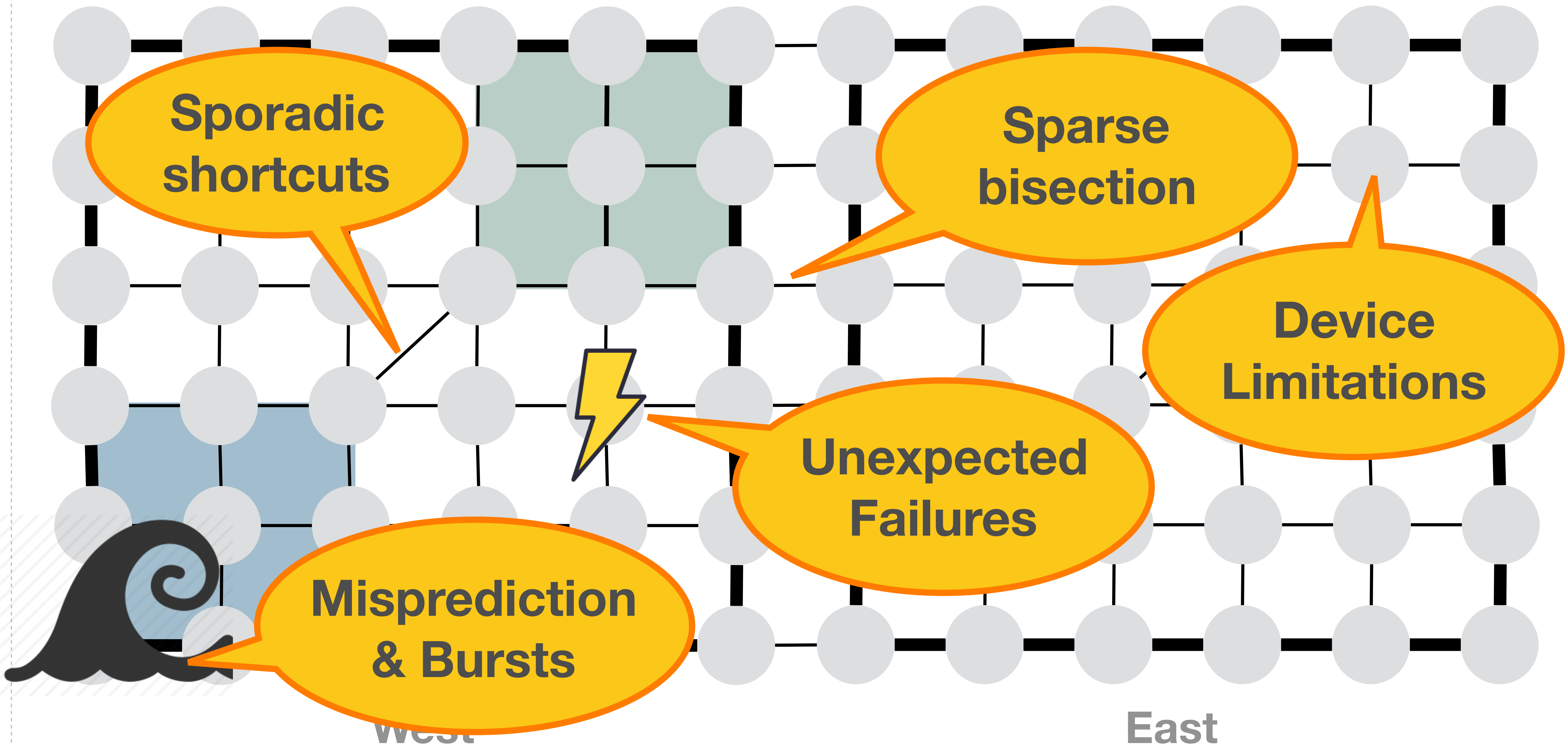
Challenges


Sporadic shortcuts
Sparse bisection
Device Limitations
West
East

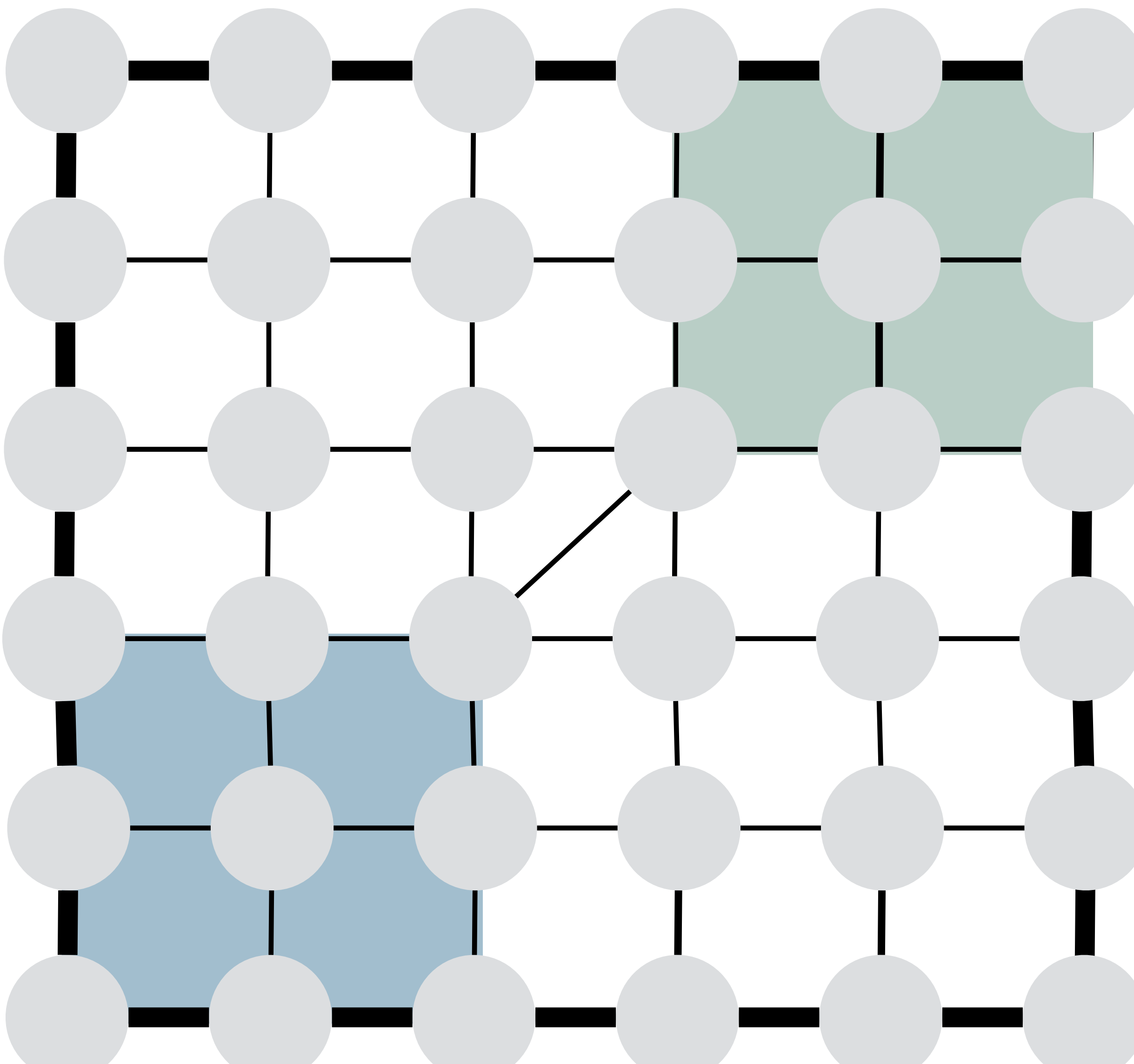
Challenges


Sporadic shortcuts
Sparse bisection
Device Limitations
Unexpected Failures
West
East

Challenges


Sporadic shortcuts
Sparse bisection
Device Limitations
Unexpected Failures
Misprediction & Bursts
West
East

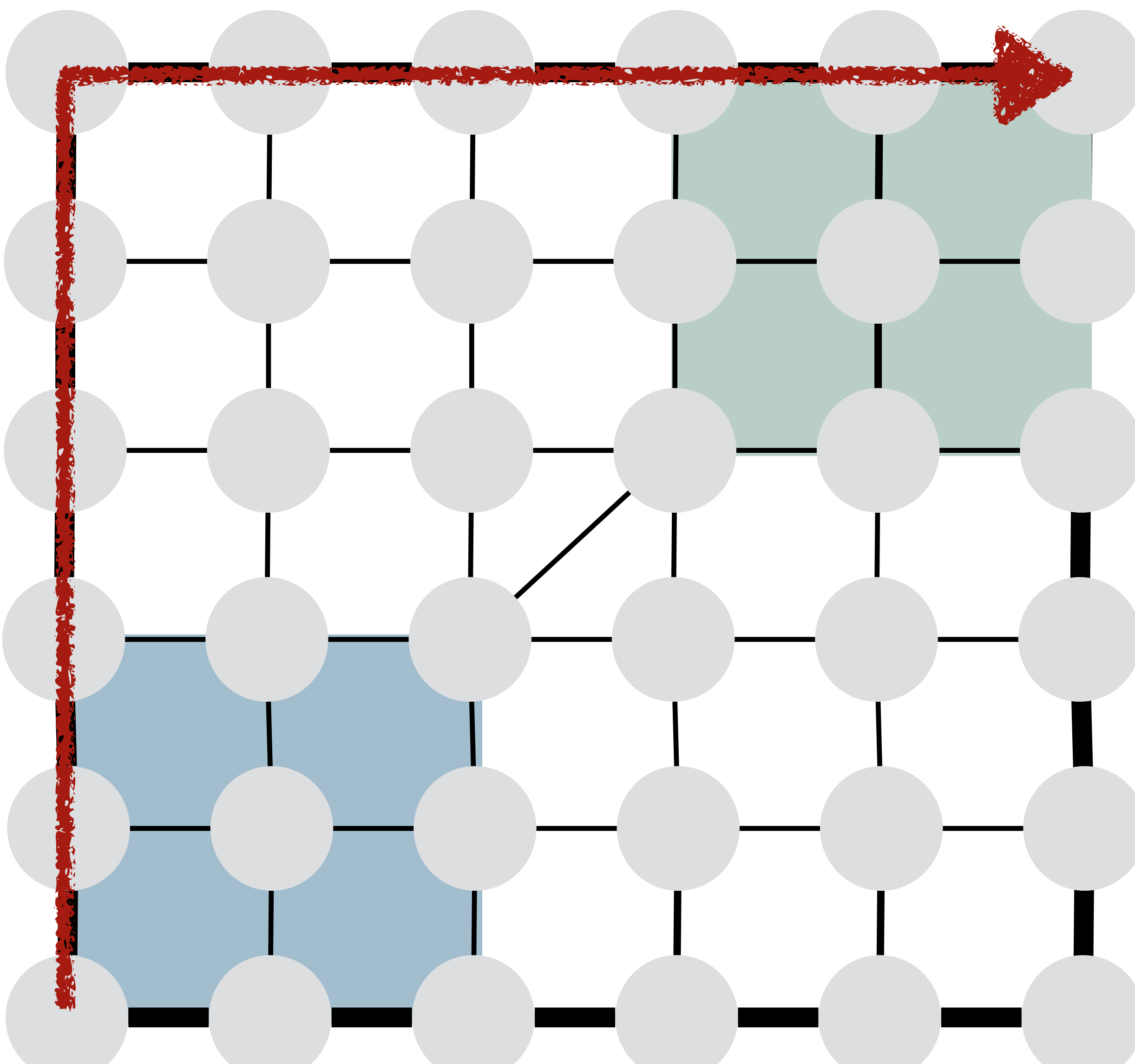
Routing Scheme



1. Which forwarding paths to use send traffic from sources to destinations?

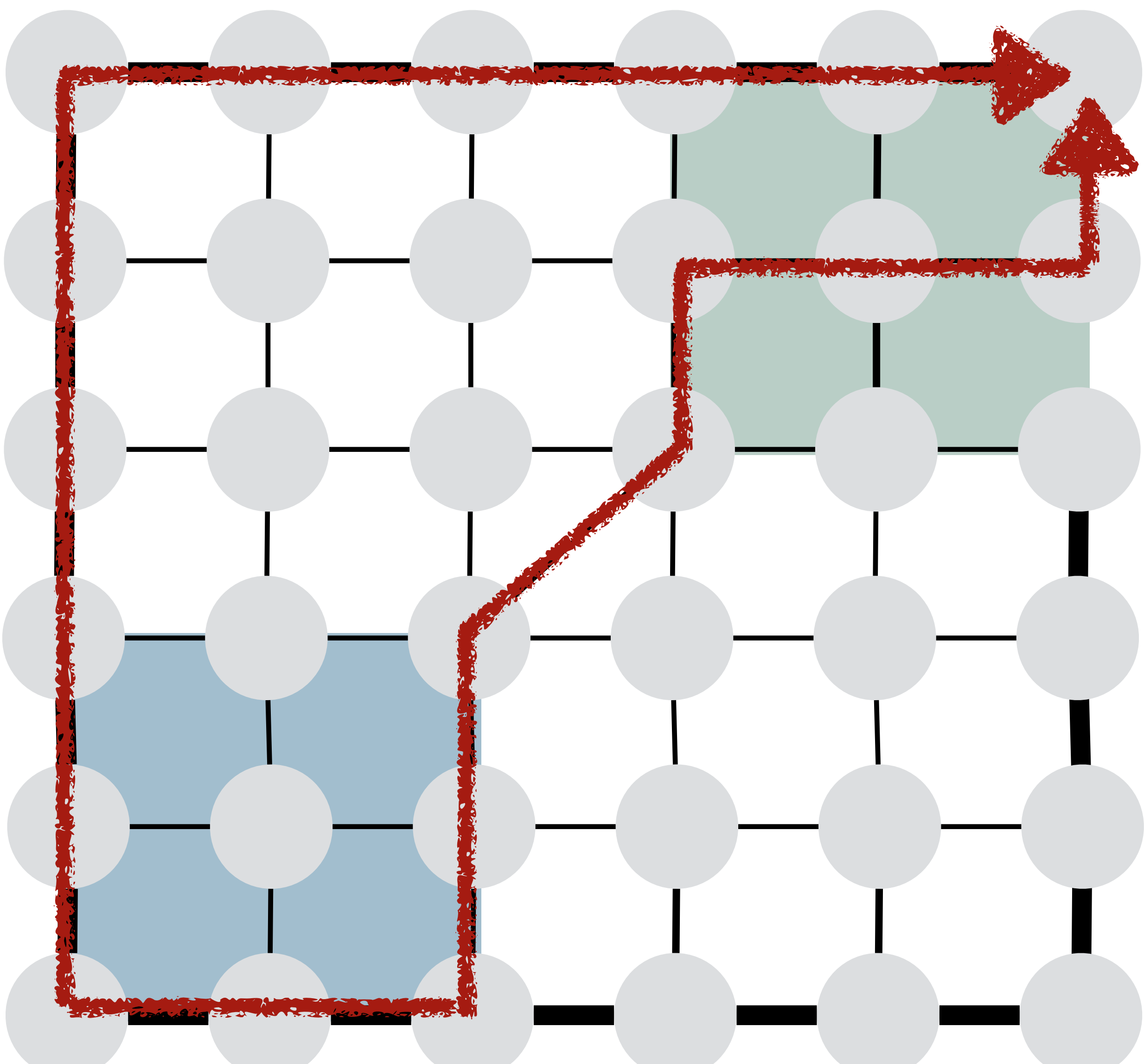
2. How to map incoming traffic flows onto multiple forwarding paths?

Routing Scheme



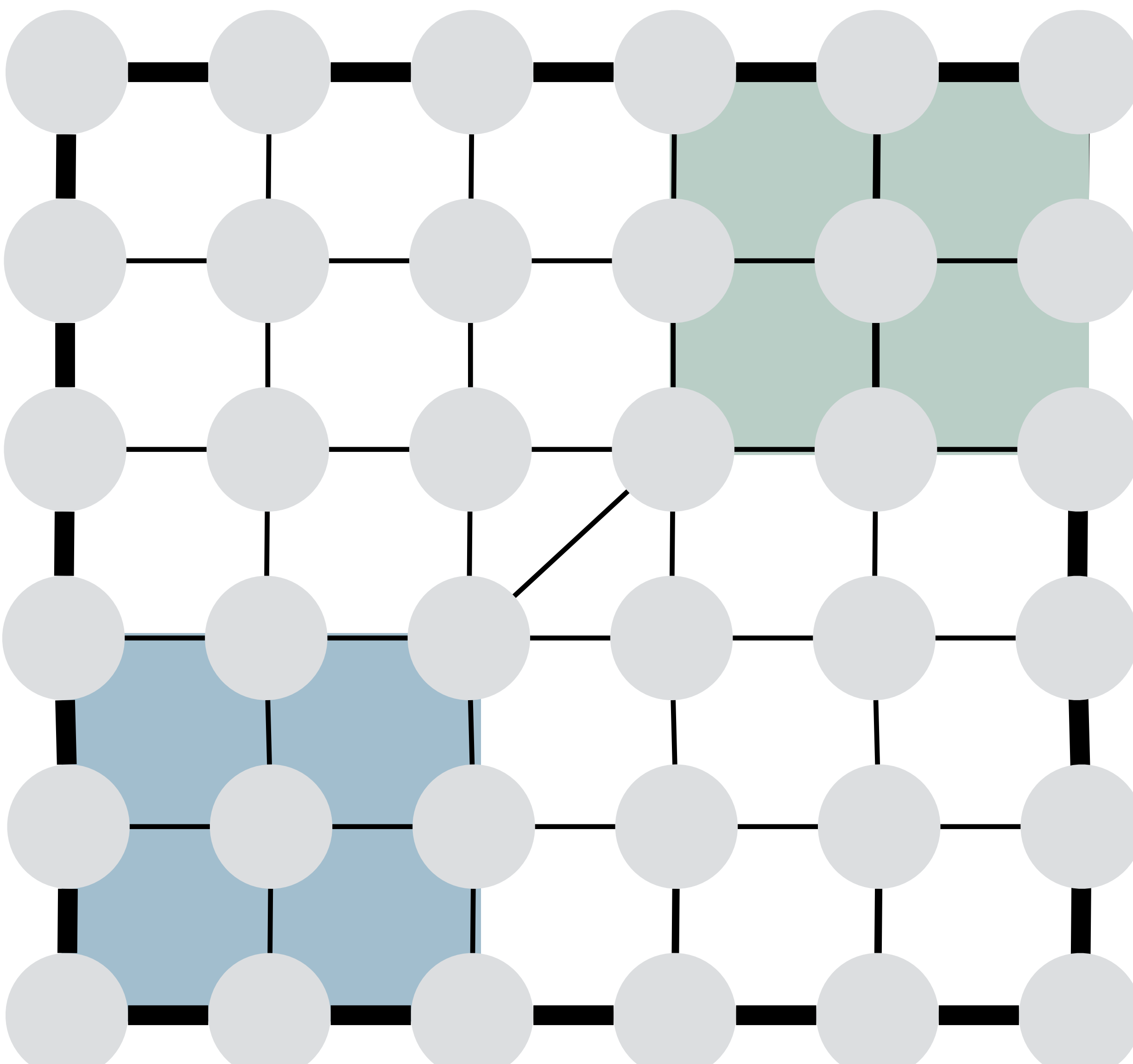
1. Which forwarding paths to use send traffic from sources to destinations?
2. How to map incoming traffic flows onto multiple forwarding paths?

Routing Scheme



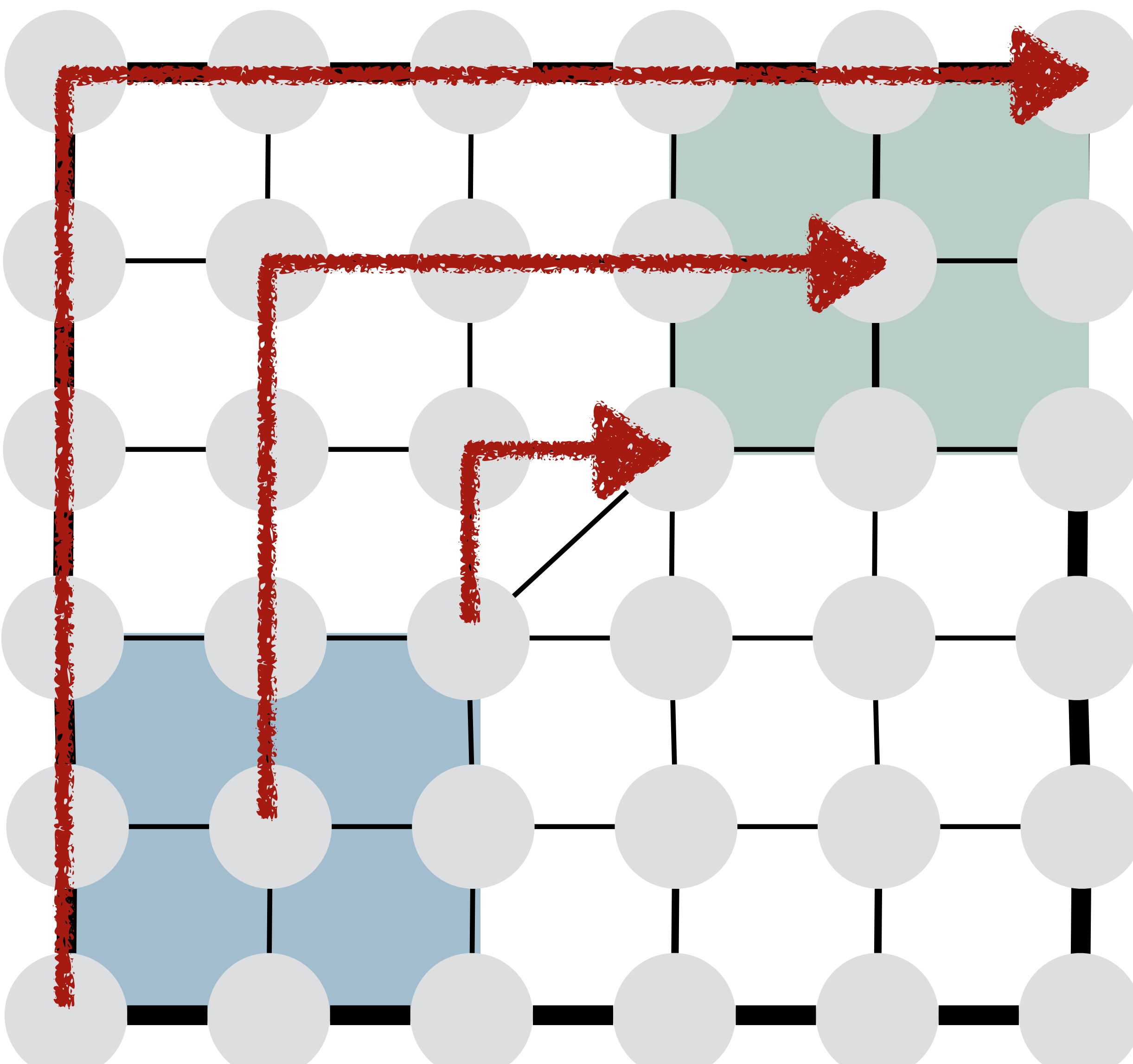
1. Which forwarding paths to use send traffic from sources to destinations?
2. How to map incoming traffic flows onto multiple forwarding paths?

Optimal Approach (Strawman MCF)



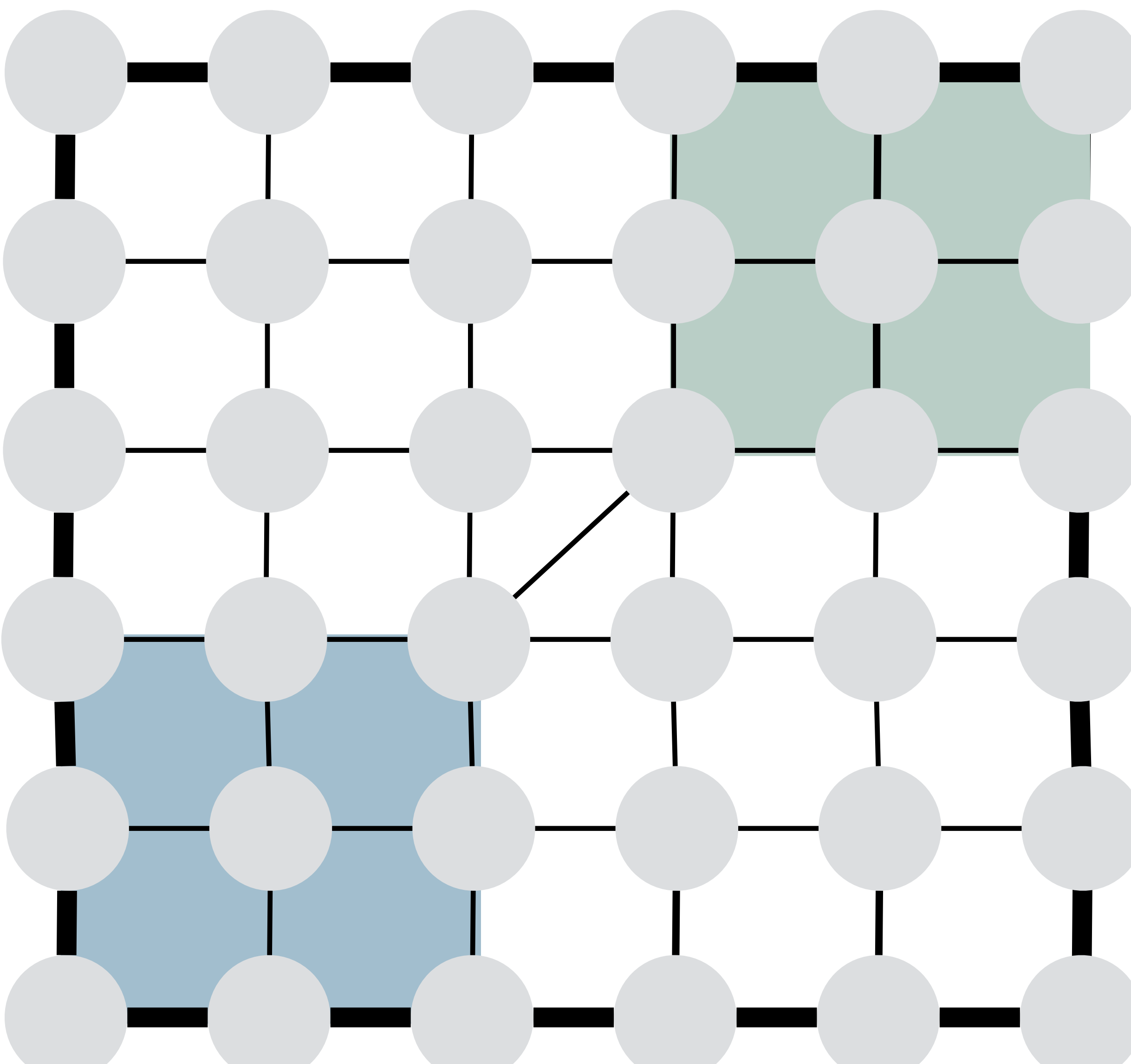
1. Estimate traffic demands from historical data
2. Encode routing problem as an optimization problem
3. Extract forwarding paths and sending rates from solution
4. Modify forwarding state
5. Repeat…

Optimal Approach (Strawman MCF)



1. Estimate traffic demands from historical data
2. Encode routing problem as an optimization problem
3. Extract forwarding paths and sending rates from solution
4. Modify forwarding state
5. Repeat...

Centralized Traffic Engineering

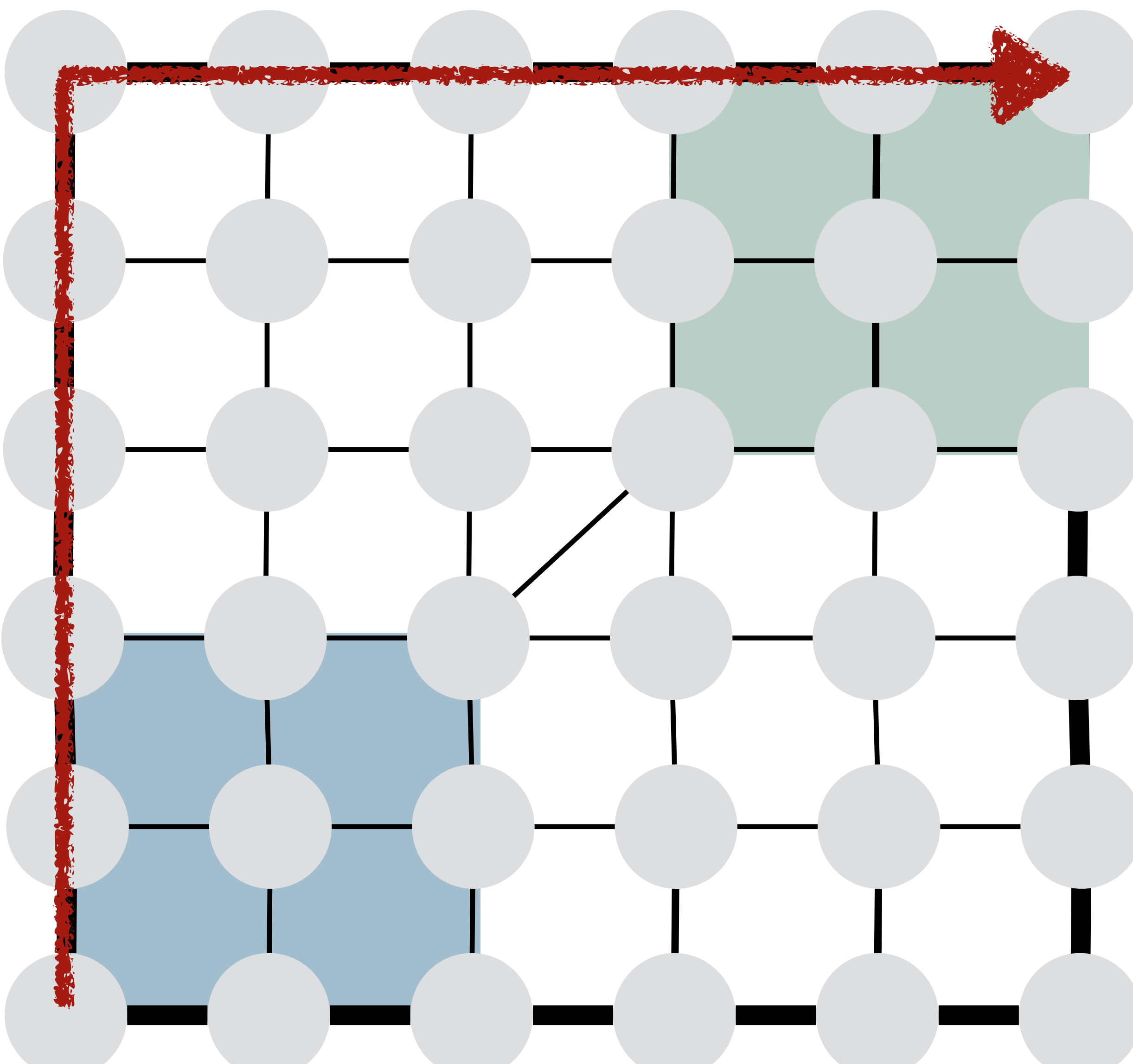


SWAN & B4 [SIGCOMM ’13]

1. Pre-compute several forwarding paths between each source and destination (e.g., K-shortest paths)

2. Compute optimal sending rates in response to (estimated or scheduled) demands

Centralized Traffic Engineering

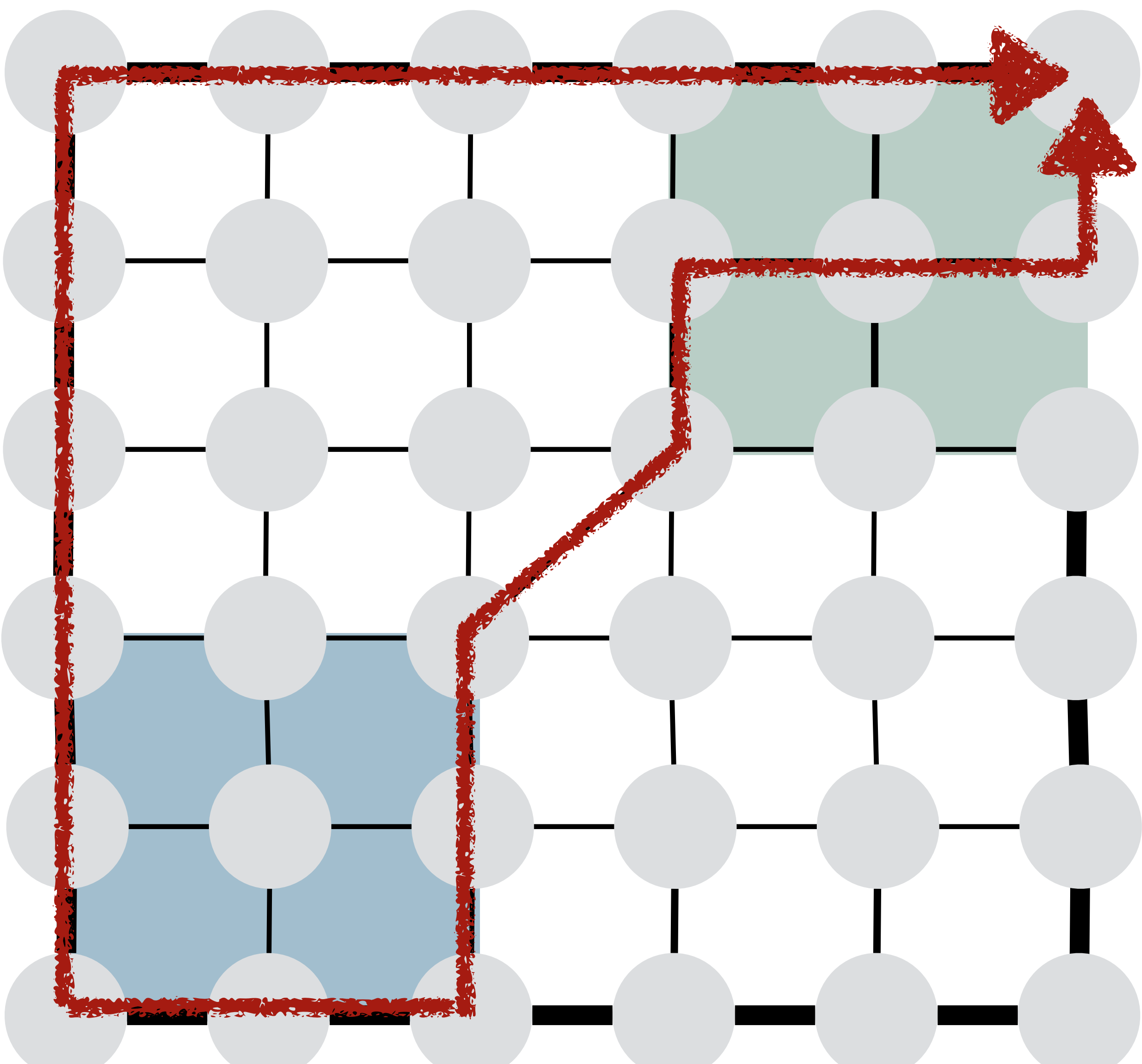


SWAN & B4 [SIGCOMM '13]

1. Pre-compute several forwarding paths between each source and destination (e.g., K-shortest paths)

2. Compute optimal sending rates in response to (estimated or scheduled) demands

Centralized Traffic Engineering



SWAN & B4 [SIGCOMM '13]

1. Pre-compute several forwarding paths between each source and destination (e.g., K-shortest paths)

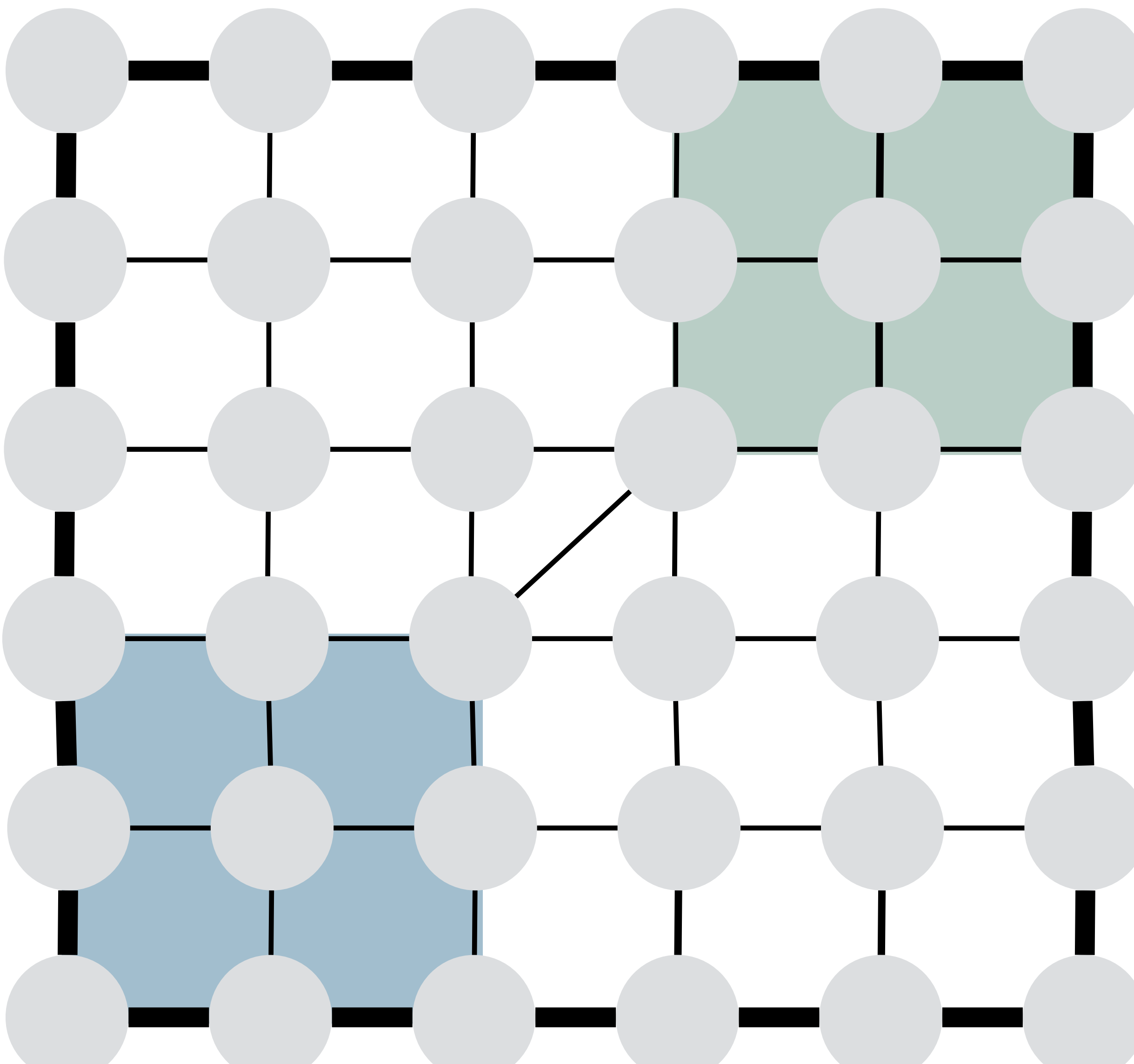
2. Compute optimal sending rates in response to (estimated or scheduled) demands

Talk Outline

- Motivation
- Randomized Routing
- Evaluation
- Conclusions

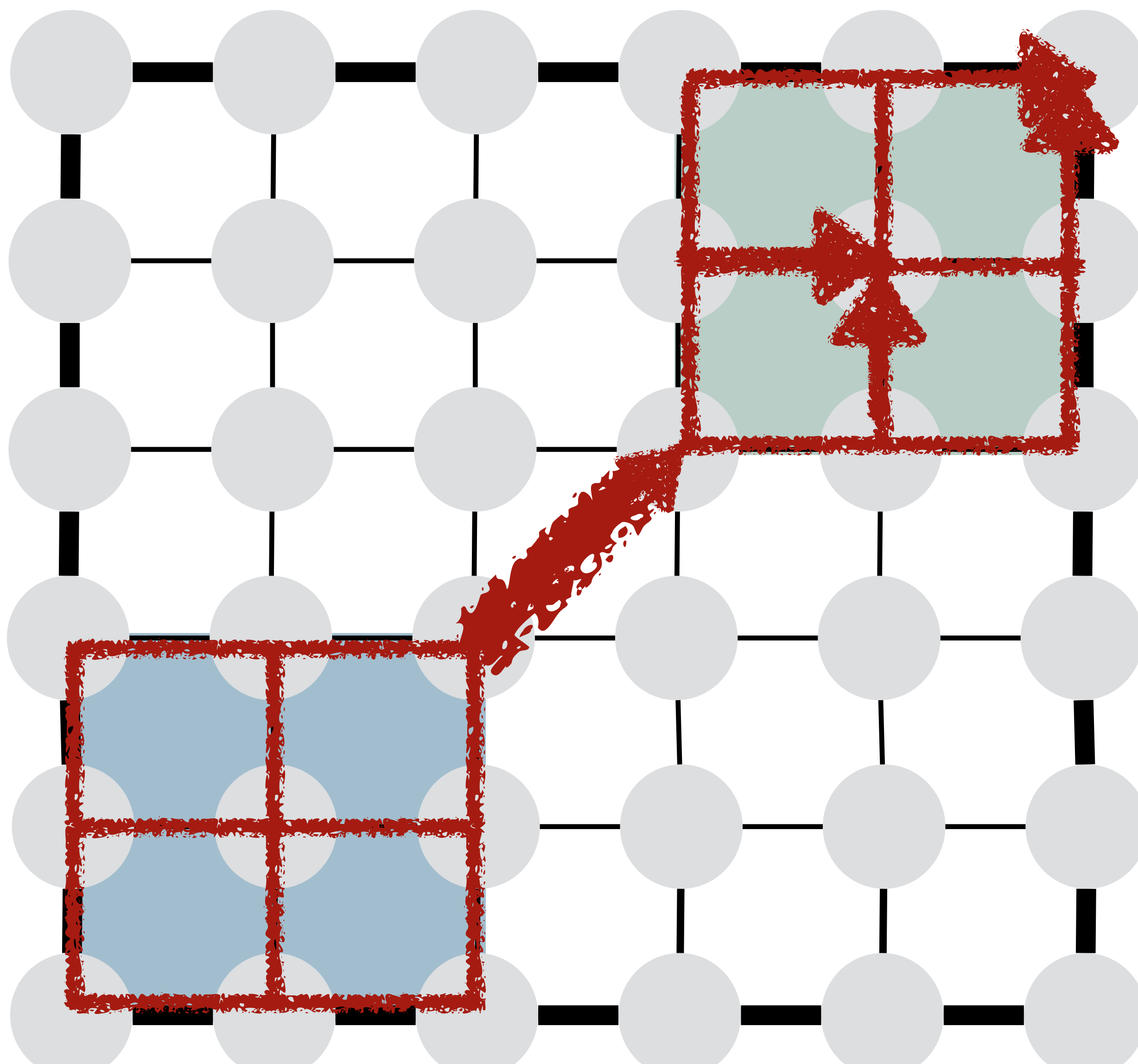
Randomized Routing

ECMP



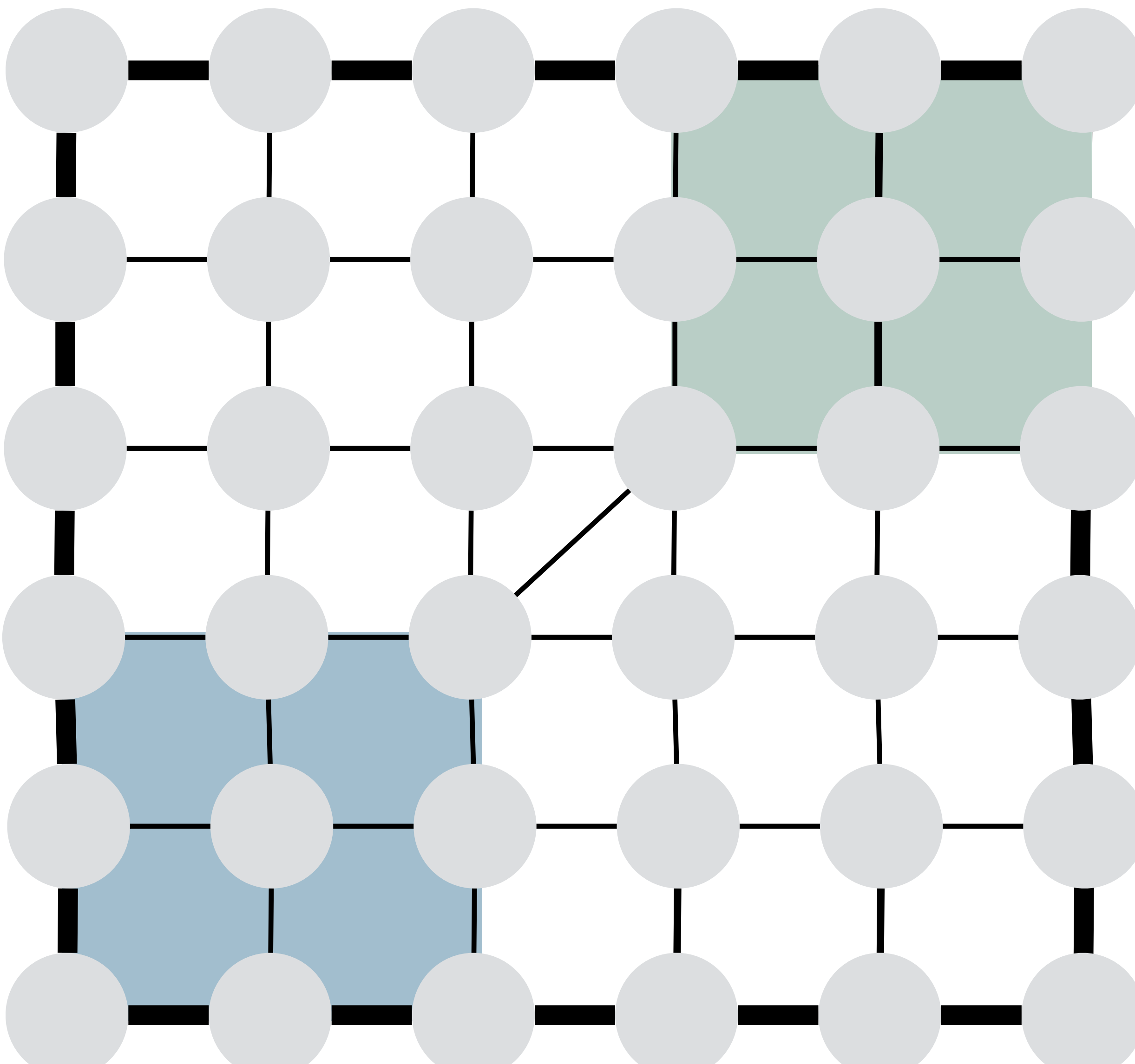
1. Pre-compute a set of least-cost paths
2. Identify flows by hashing packet header fields
3. Randomly forward along least cost paths

ECMP



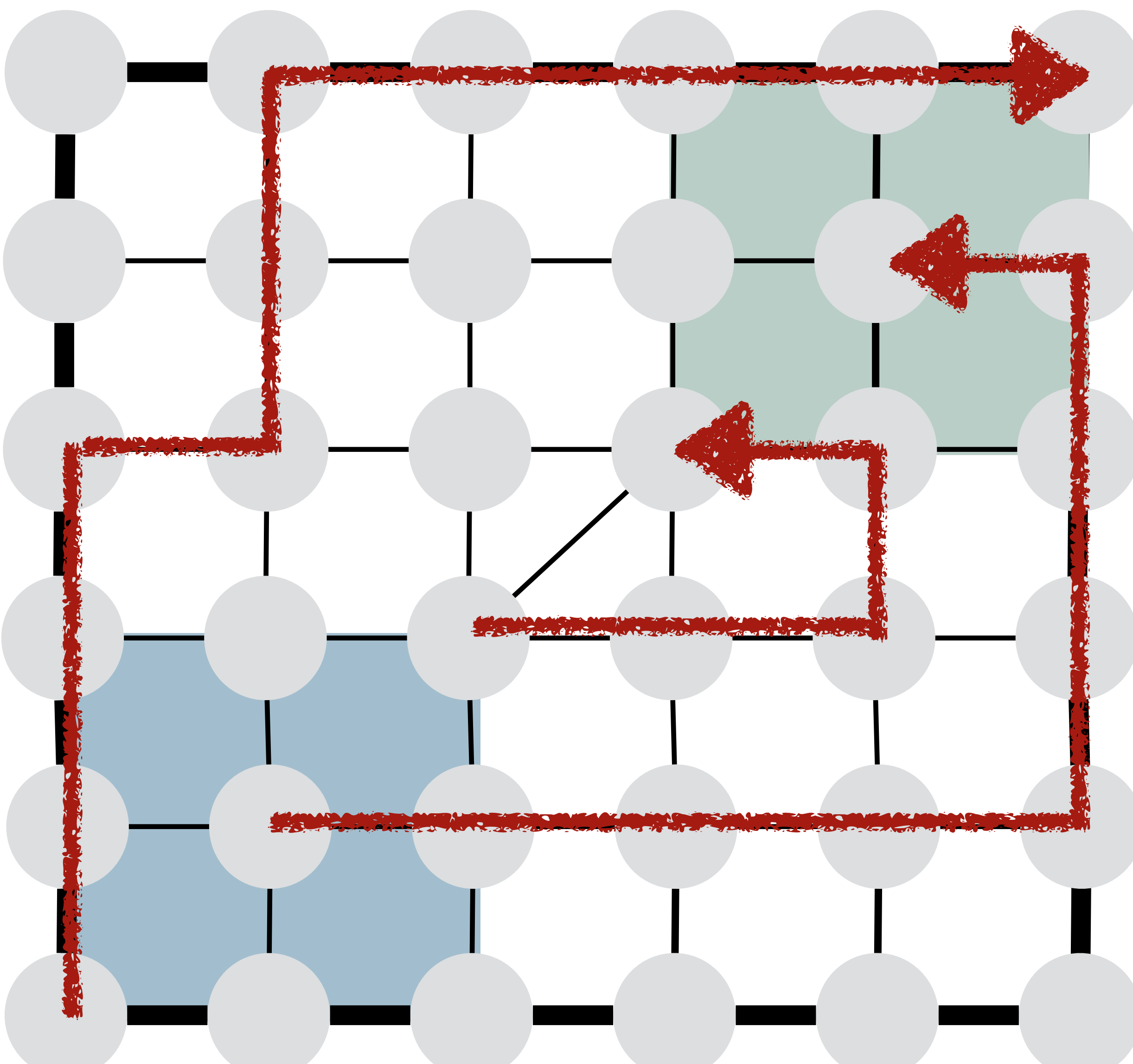
1. Pre-compute a set of least-cost paths
2. Identify flows by hashing packet header fields
3. Randomly forward along least cost paths

Valiant Load Balancing



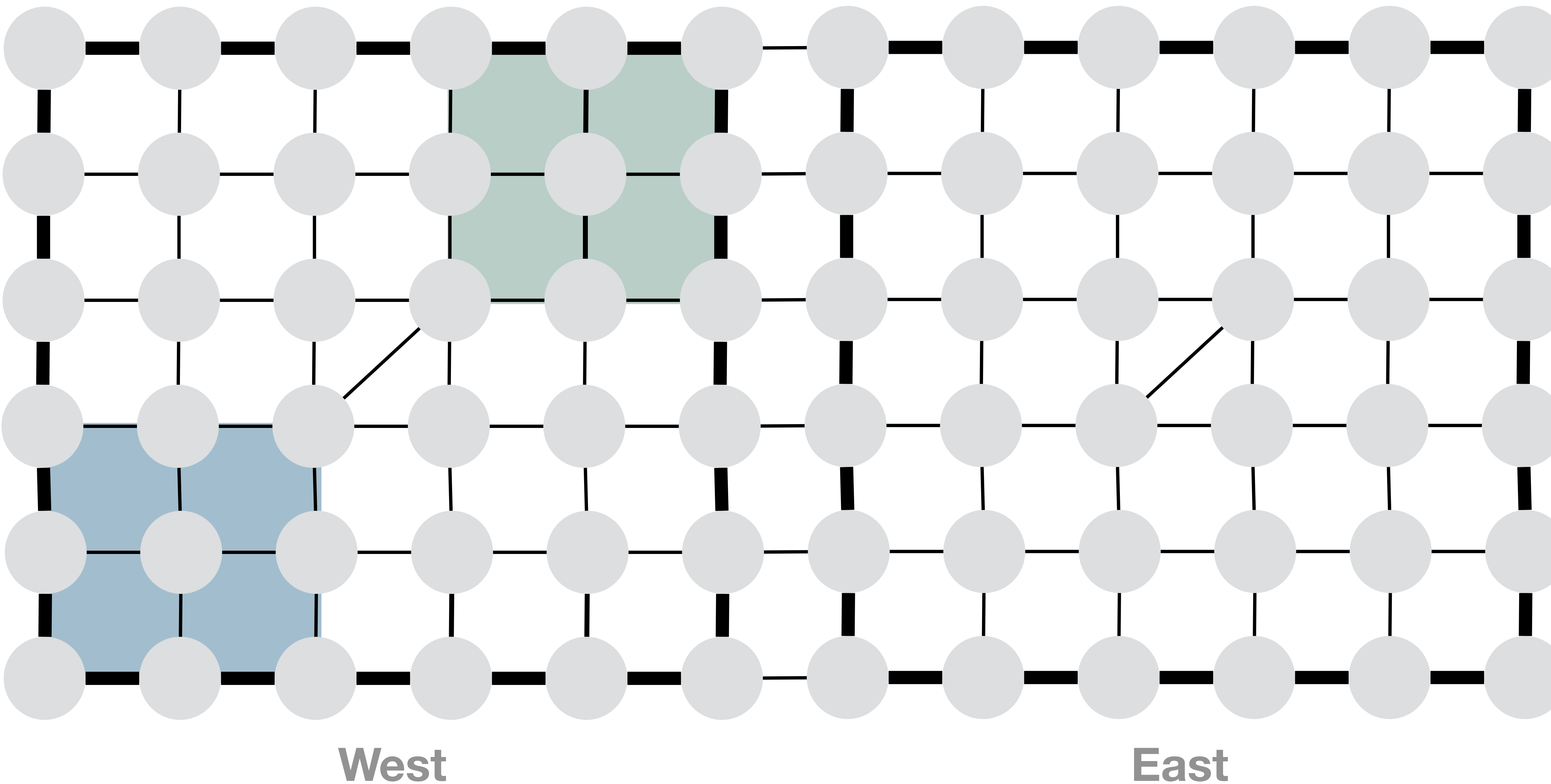
1. Choose a random intermediate node
2. Route from source to intermediate node
3. Route from intermediate node to destination

Valiant Load Balancing

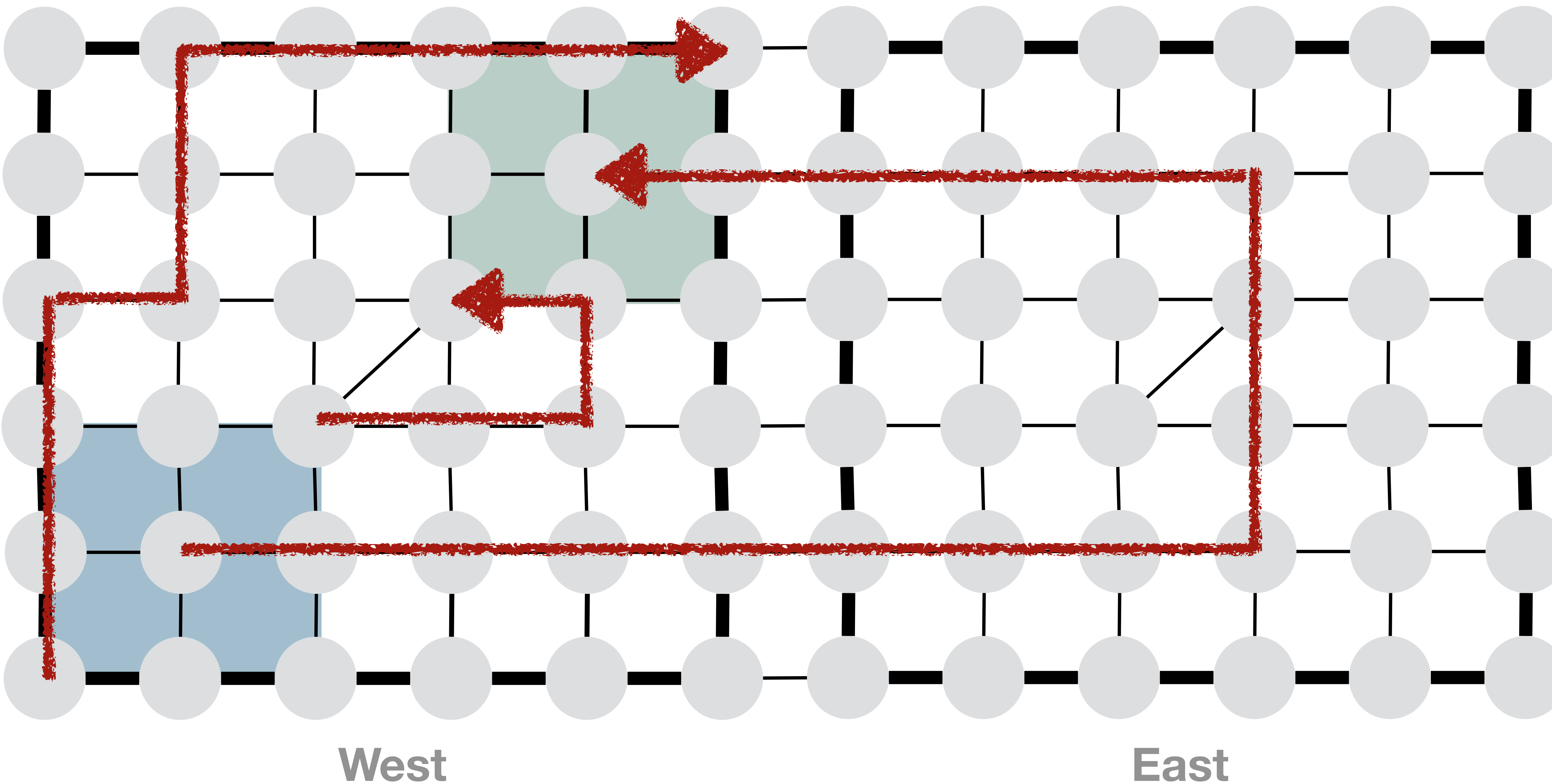


1. Choose a random intermediate node
2. Route from source to intermediate node
3. Route from intermediate node to destination

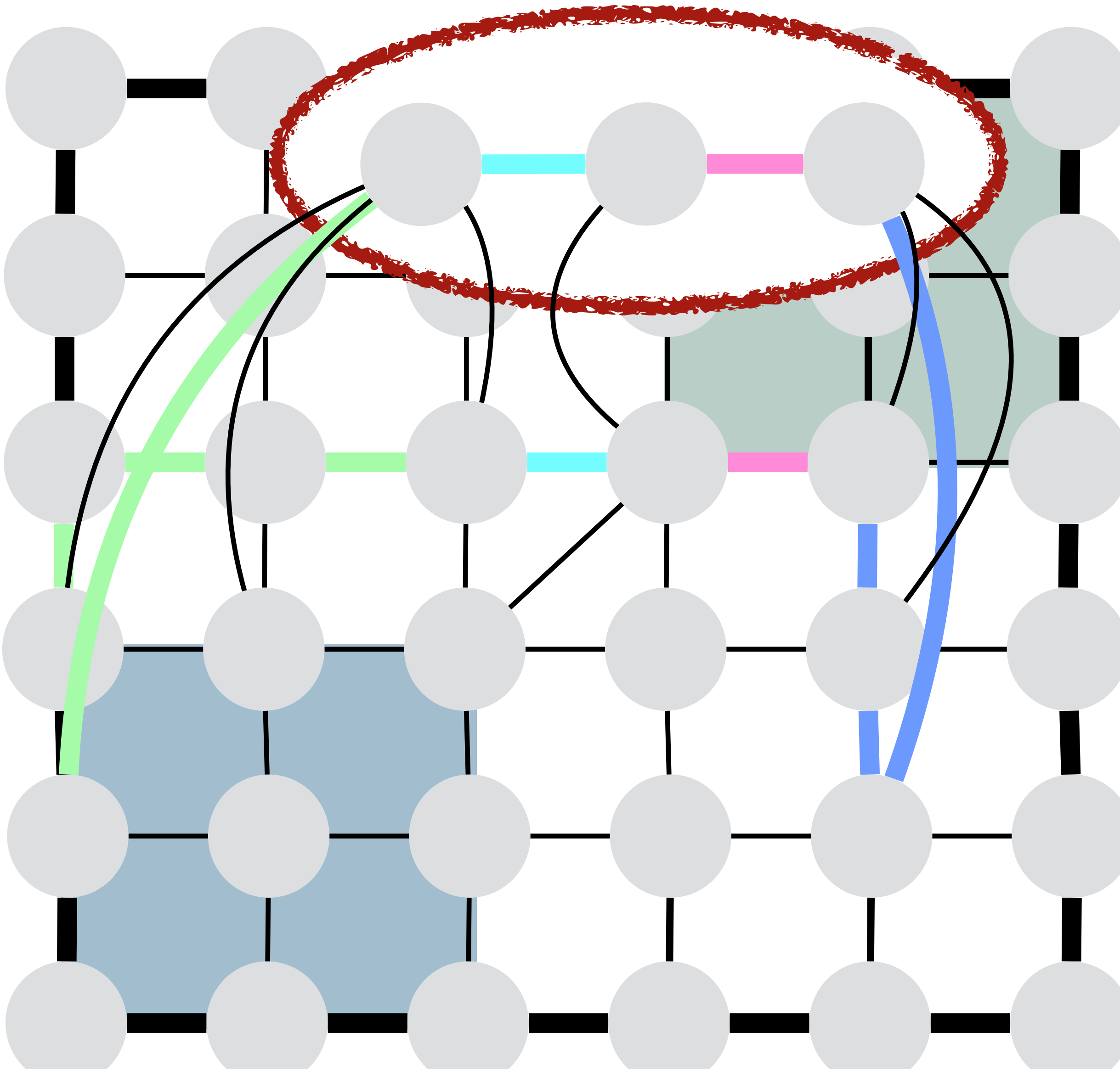
Valiant Load Balancing



Valiant Load Balancing



Oblivious Routing



- A *routing tree* is an overlay in which nodes correspond to physical nodes and edges to physical paths
- A *randomized routing tree* is probability distribution over routing trees
- Intuition: there is a duality between low-stretch routing trees and low-congestion routing schemes

Räcke's Algorithm

- Räcke's algorithm iteratively constructs a randomized routing tree
- At each iteration, it penalizes edges that have been heavily utilized in previous trees
- Achieves a polylogarithmic competitive ratio with respect to the optimal scheme regardless of the demand matrix—i.e. it is *oblivious!*

Semi-Oblivious Routing

- *Semi-oblivious routing* combines Räcke's oblivious routing with dynamic rate adaptation / local failure recovery
  - Forwarding paths: computed statically
  - Sending rates: adapt to changing demands
- 👎 Hajiaghayi et al. proved Ω(log(n)/log (log(n))) competitive ratio
- 👍 Realistic workloads are different from worst-case

SDN Implementation & Evaluation

Kulfi Framework

Routing Algorithm	Description	Type	Path Diversity	Max Congestion	Overheads	
					Churn	Recovery
MCF	Multi-Commodity Flow solved with LP [20]	conscious	medium	least	high	slow
MW	Multi-Commodity Flow solved with Multiplicative Weights [17]	conscious	medium	least	high	slow
SPF	Shortest Path First	oblivious	least	high	none	none
ECMP	Equal-Cost, Multi-Path	oblivious	low	high	none	fast
KSP	K-Shortest Paths	oblivious	medium	medium	none	fast
Räcke	Räcke [38]	oblivious	high	low	none	fast
VLB	Valiant Load Balancing [44]	oblivious	high	medium	none	fast
SemiMCF-MCF	MCF for paths MCF for weights	semi-oblivious	medium	least	none	fast
SemiMCF-ECMP	ECMP for paths MCF for weights	semi-oblivious	low	medium	none	fast
SemiMCF-KSP	KSP for paths MCF for weights	semi-oblivious	medium	medium	none	fast
SemiMCF-Räcke	Räcke for paths MCF for weights	semi-oblivious	high	low	none	fast
SemiMCF-VLB	VLB for paths MCF for weights	semi-oblivious	high	medium	none	fast
SemiMCF-MCF-Env	MCF over demand envelope for paths MCF for weights [43]	semi-oblivious	medium	low	none	fast
SemiMCF-MCF-FT-Env	Multiple MCF-Env considering failures MCF for weights [43]	semi-oblivious	high	medium	none	fast

- **Implemented over a dozen different traffic engineering schemes**
- **Measure performance in simulator and hardware testbed with a variety of demands and failures**
- **Used "local" failure recovery**

Kulfi Framework

Routing Algorithm	Description	Type	Path Diversity	Max Congestion	Overheads Churn	Overheads Recovery
MCF	Multi-Commodity Flow solved with LP [20]	conscious	medium	least	high	slow
MW	Multi-Commodity Flow solved with Multiplicative Weights [17]	conscious	medium	least	high	slow
SPF	Shortest Path First	oblivious	least	high	none	none
ECMP	Equal-Cost, Multi-Path	oblivious	low	high	none	fast
KSP	K-Shortest Paths	oblivious	medium	medium	none	fast
Räcke	Räcke [38]	oblivious	high	low	none	fast
VLB	Valiant Load Balancing [44]	oblivious	high	medium	none	fast
SemiMCF-MCF	MCF for paths MCF for weights	semi-oblivious	medium	least	none	fast
SemiMCF-ECMP	ECMP for paths MCF for weights	semi-oblivious	low	medium	none	fast
SemiMCF-KSP	KSP for paths MCF for weights	semi-oblivious	medium	medium	none	fast
SemiMCF-Räcke	Räcke for paths MCF for weights	semi-oblivious	high	low	none	fast
SemiMCF-VLB	VLB for paths MCF for weights	semi-oblivious	high	medium	none	fast
SemiMCF-MCF-Env	MCF over demand envelope for paths MCF for weights [43]	semi-oblivious	medium	low	none	fast
SemiMCF-MCF-FT-Env	Multiple MCF-Env considering failures MCF for weights [43]	semi-oblivious	high	medium	none	fast

- Implemented over a dozen different traffic engineering schemes
- Measure performance in simulator and hardware testbed with a variety of demands and failures
- Used “local” failure recovery

Kulfi Framework

Routing Algorithm	Description	Type	Path	Max	Overheads	
			Diversity	Congestion	Churn	Recovery
MCF	Multi-Commodity Flow solved with LP [20]	conscious	medium	least	high	slow
MW	Multi-Commodity Flow solved with Multiplicative Weights [17]	conscious	medium	least	high	slow
SPF	Shortest Path First	oblivious	least	high	none	none
ECMP	Equal-Cost, Multi-Path	oblivious	low	high	none	fast
KSP	K-Shortest Paths	oblivious	medium	medium	none	fast
Räcke	Räcke [38]	oblivious	high	low	none	fast
VLB	Valiant Load Balancing [44]	oblivious	high	medium	none	fast
SemiMCF-MCF	MCF for paths MCF for weights	semi-oblivious	medium	least	none	fast
SemiMCF-ECMP	ECMP for paths MCF for weights	semi-oblivious	low	medium	none	fast
SemiMCF-KSP	KSP for paths MCF for weights	semi-oblivious	medium	medium	none	fast
SemiMCF-Räcke	Räcke for paths MCF for weights	semi-oblivious	high	low	none	fast
SemiMCF-VLB	VLB for paths MCF for weights	semi-oblivious	high	medium	none	fast
SemiMCF-MCF-Env	MCF over demand envelope for paths MCF for weights [43]	semi-oblivious	medium	low	none	fast
SemiMCF-MCF-FT-Env	Multiple MCF-Env considering failures MCF for weights [43]	semi-oblivious	high	medium	none	fast

- Implemented over a dozen different traffic engineering schemes
- Measure performance in simulator and hardware testbed with a variety of demands and failures
- Used “local” failure recovery

Kulfi Framework

Routing Algorithm	Description	Type	Path Diversity	Max Congestion	Overheads Churn	Overheads Recovery
MCF	Multi-Commodity Flow solved with LP [20]	conscious	medium	least	high	slow
MW	Multi-Commodity Flow solved with Multiplicative Weights [17]	conscious	medium	least	high	slow
SPF	Shortest Path First	oblivious	least	high	none	none
ECMP	Equal-Cost, Multi-Path	oblivious	low	high	none	fast
KSP	K-Shortest Paths	oblivious	medium	medium	none	fast
Räcke	Räcke [38]	oblivious	high	low	none	fast
VLB	Valiant Load Balancing [44]	oblivious	high	medium	none	fast
SemiMCF-MCF	MCF for paths MCF for weights	semi-oblivious	medium	least	none	fast
SemiMCF-ECMP	ECMP for paths MCF for weights	semi-oblivious	low	medium	none	fast
SemiMCF-KSP	KSP for paths MCF for weights	semi-oblivious	medium	medium	none	fast
SemiMCF-Räcke	Räcke for paths MCF for weights	semi-oblivious	high	low	none	fast
SemiMCF-VLB	VLB for paths MCF for weights	semi-oblivious	high	medium	none	fast
SemiMCF-MCF-Env	MCF over demand envelope for paths MCF for weights [43]	semi-oblivious	medium	low	none	fast
SemiMCF-MCF-FT-Env	Multiple MCF-Env considering failures MCF for weights [43]	semi-oblivious	high	medium	none	fast

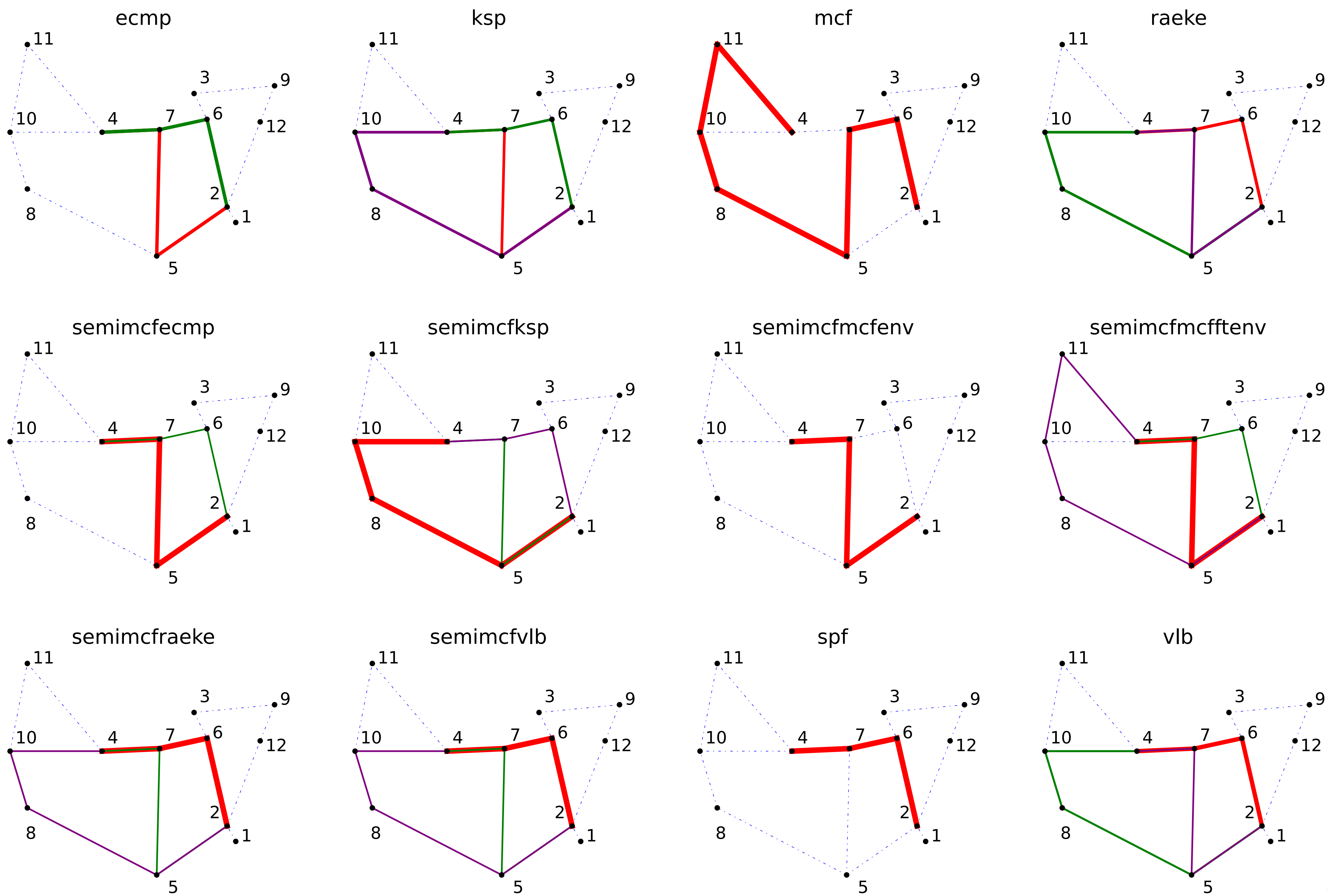
- Implemented over a dozen different traffic engineering schemes
- Measure performance in simulator and hardware testbed with a variety of demands and failures
- Used “local” failure recovery

Kulfi Framework

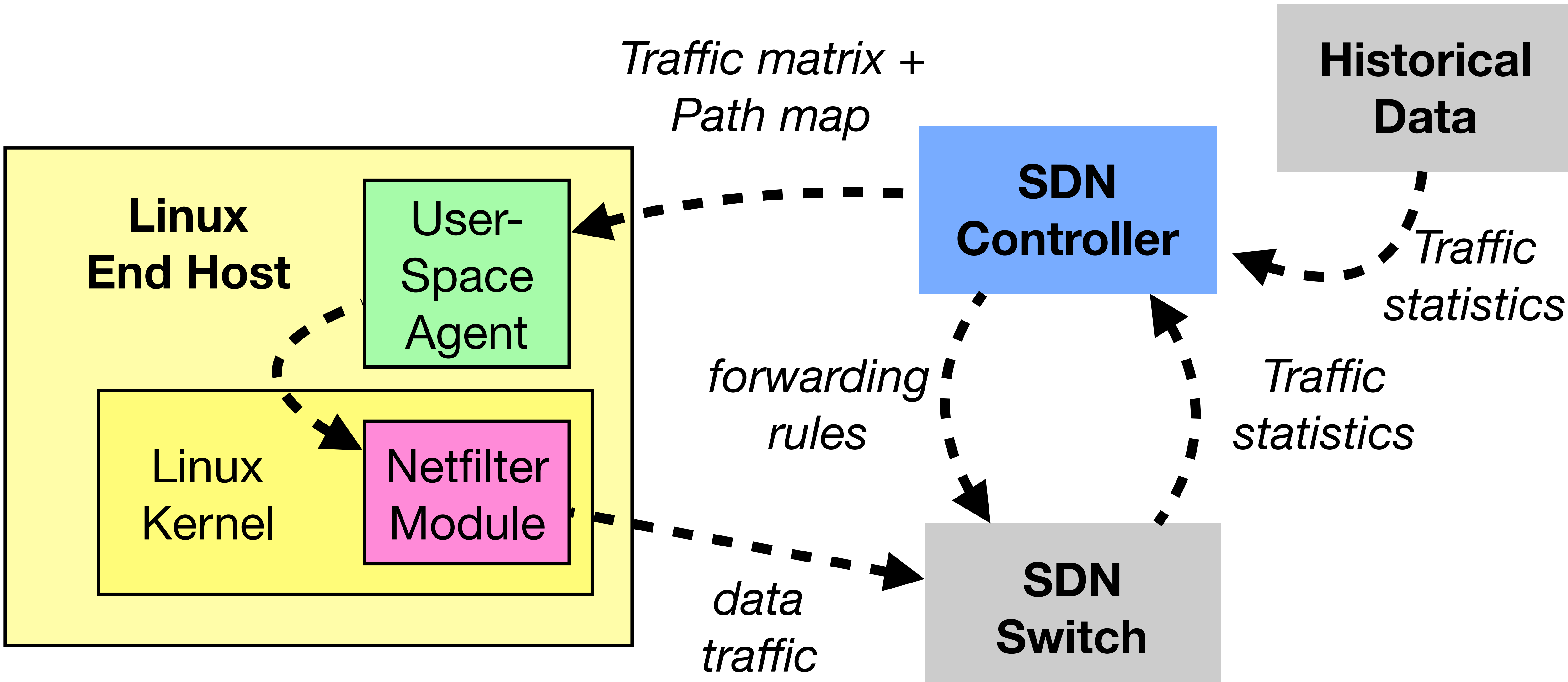
Routing Algorithm	Description	Type	Path Diversity	Max Congestion	Overheads	
					Churn	Recovery
MCF	Multi-Commodity Flow solved with LP [20]	conscious	medium	least	high	slow
MW	Multi-Commodity Flow solved with Multiplicative Weights [17]	conscious	medium	least	high	slow
SPF	Shortest Path First	oblivious	least	high	none	none
ECMP	Equal-Cost, Multi-Path	oblivious	low	high	none	fast
KSP	K-Shortest Paths	oblivious	medium	medium	none	fast
Räcke	Räcke [38]	oblivious	high	low	none	fast
VLB	Valiant Load Balancing [44]	oblivious	high	medium	none	fast
SemiMCF-MCF	MCF for paths MCF for weights	semi-oblivious	medium	least	none	fast
SemiMCF-ECMP	ECMP for paths MCF for weights	semi-oblivious	low	medium	none	fast
SemiMCF-KSP	KSP for paths MCF for weights	semi-oblivious	medium	medium	none	fast
SemiMCF-Räcke	Räcke for paths MCF for weights	semi-oblivious	high	low	none	fast
SemiMCF-VLB	VLB for paths MCF for weights	semi-oblivious	high	medium	none	fast
SemiMCF-MCF-Env	MCF over demand envelope for paths MCF for weights [43]	semi-oblivious	medium	low	none	fast
SemiMCF-MCF-FT-Env	Multiple MCF-Env considering failures MCF for weights [43]	semi-oblivious	high	medium	none	fast

- Implemented over a dozen different traffic engineering schemes
- Measure performance in simulator and hardware testbed with a variety of demands and failures
- Used “local” failure recovery

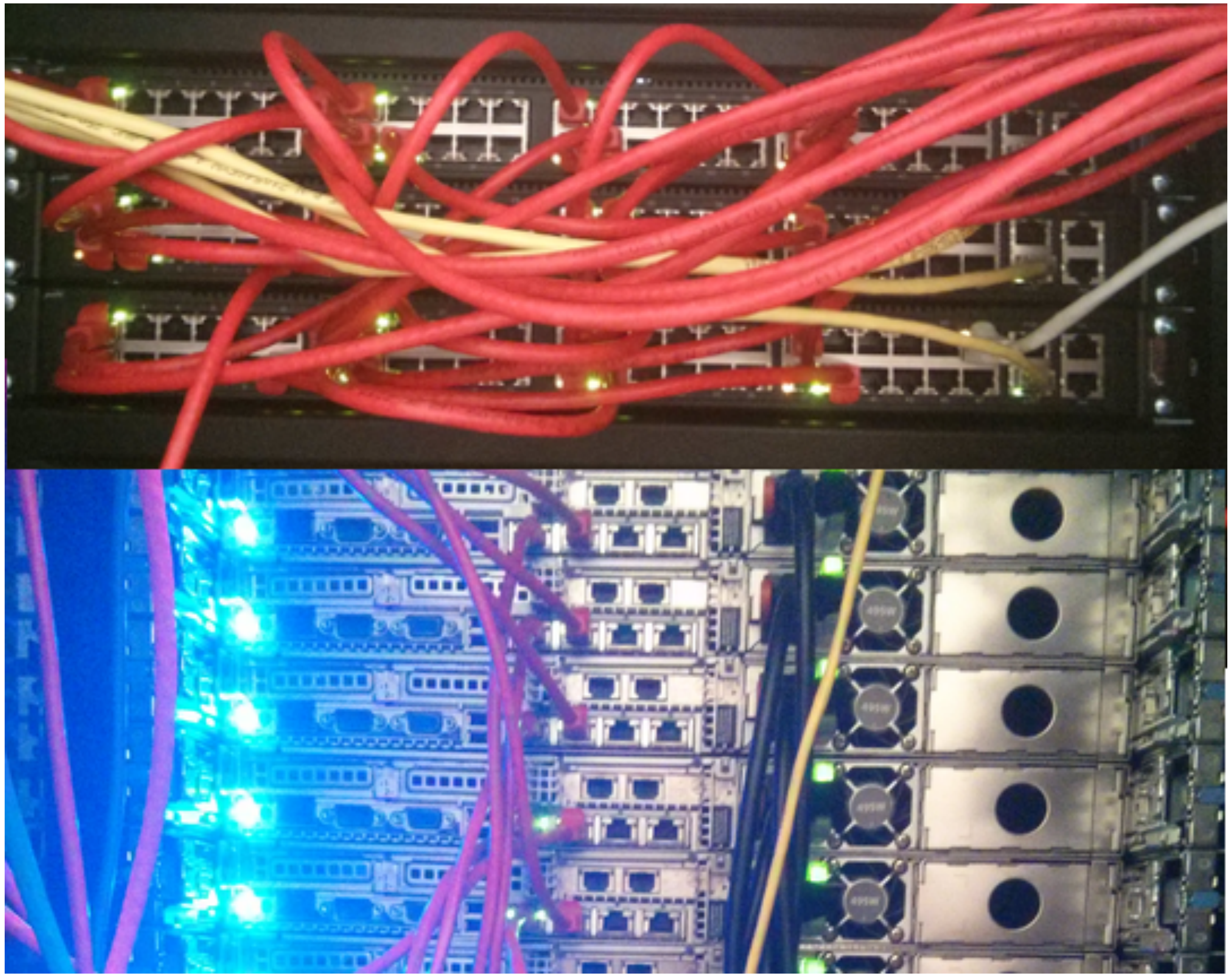
Visualizing Routing Schemes



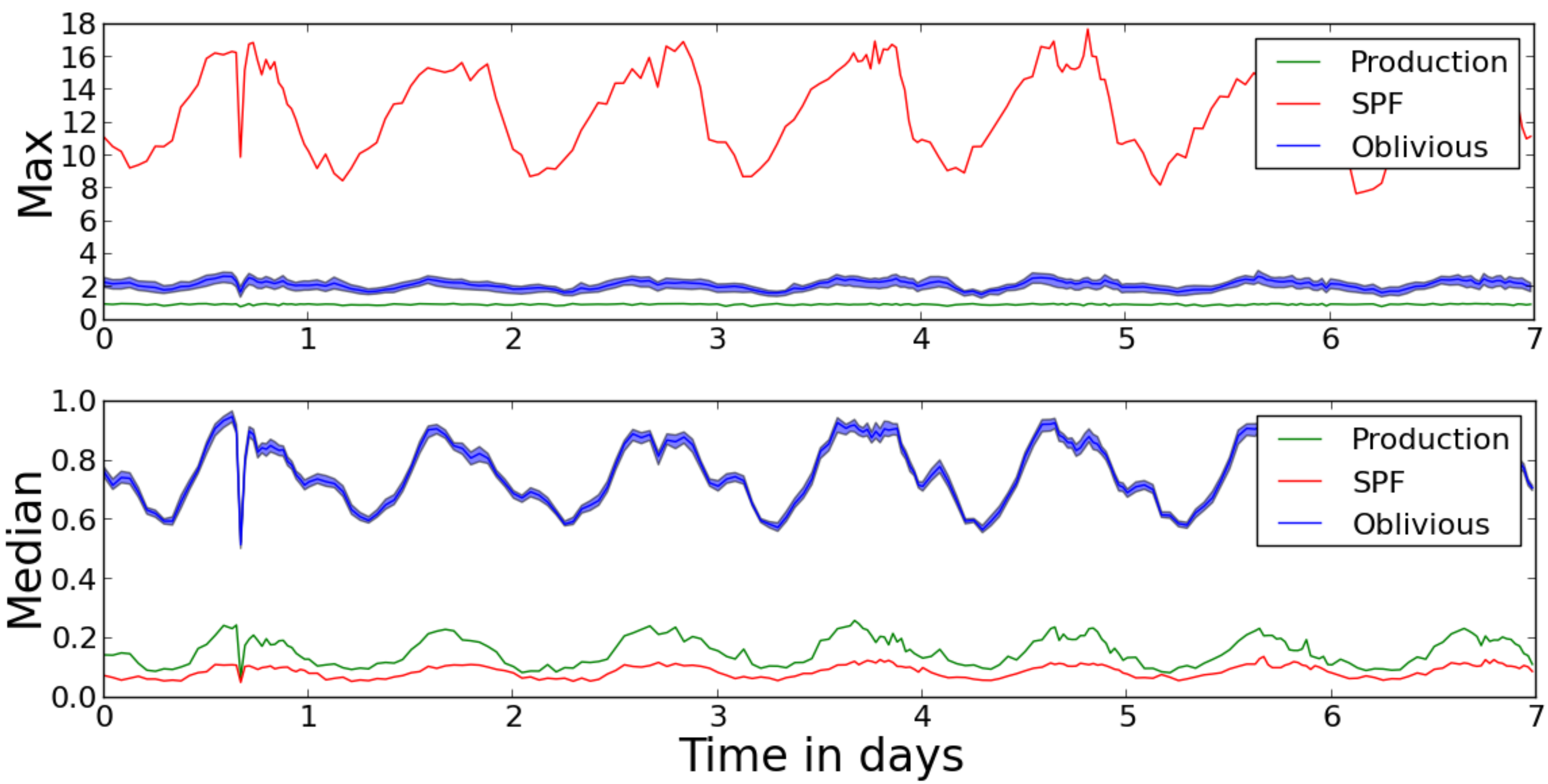
SDN Implementation



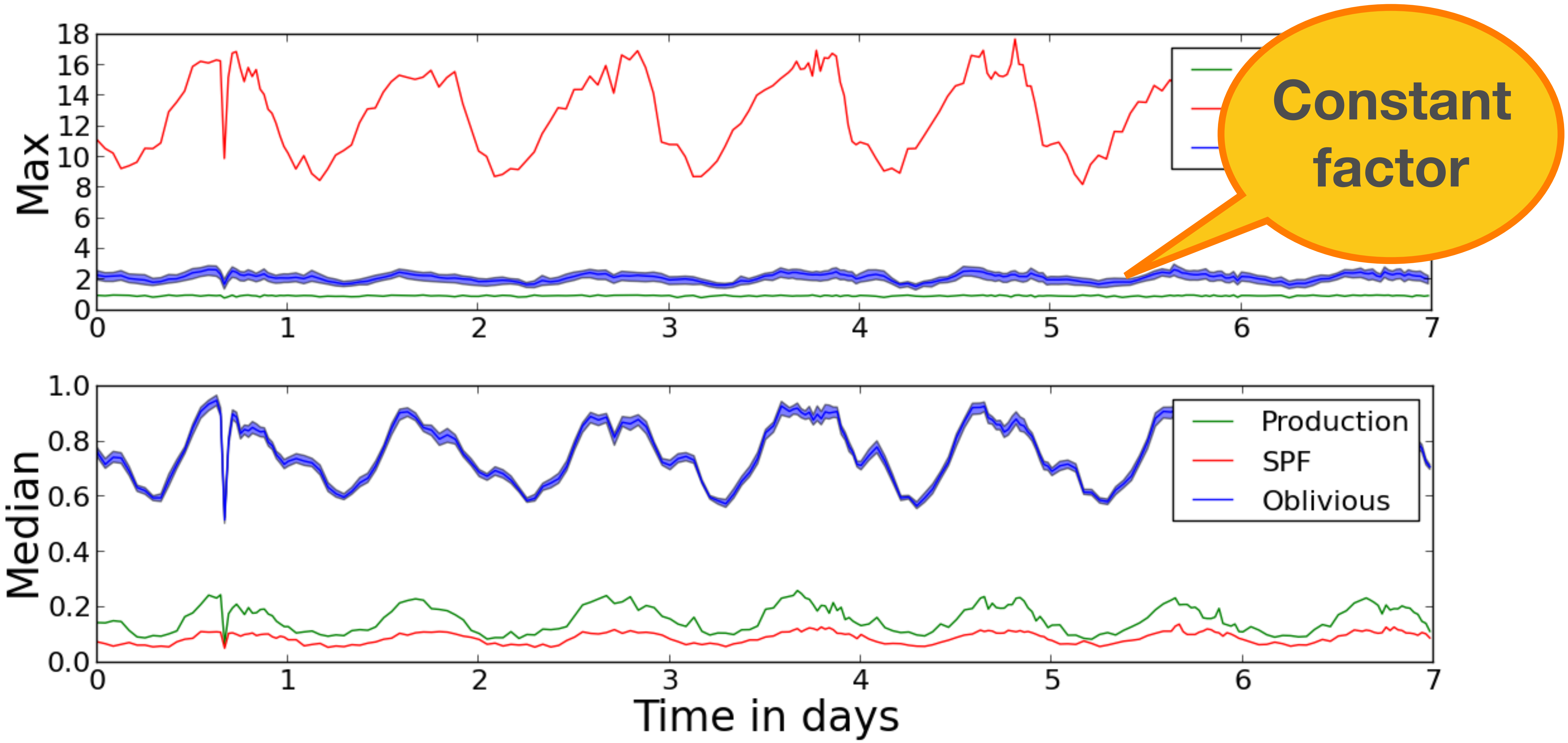
Hardware Testbed



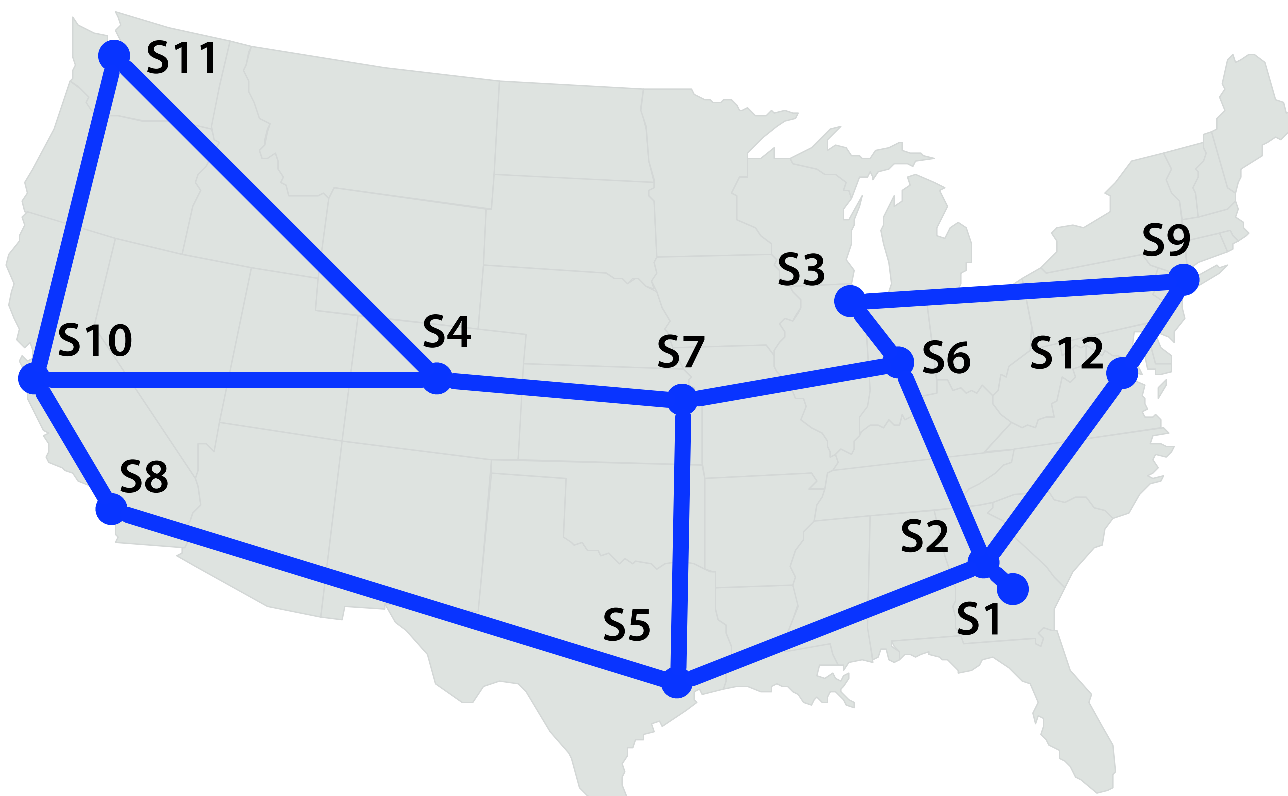
Facebook Backbone: Simulation



Facebook Backbone: Simulation

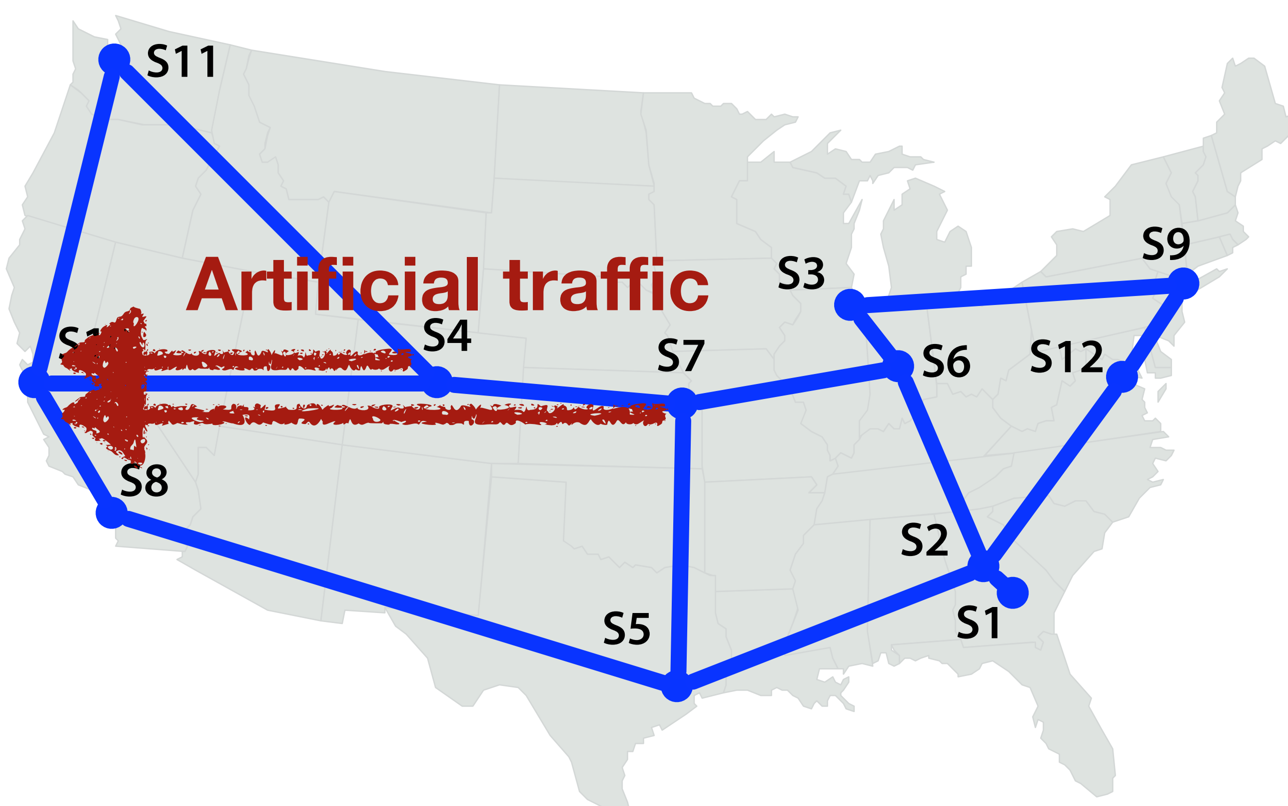


Abilene Topology



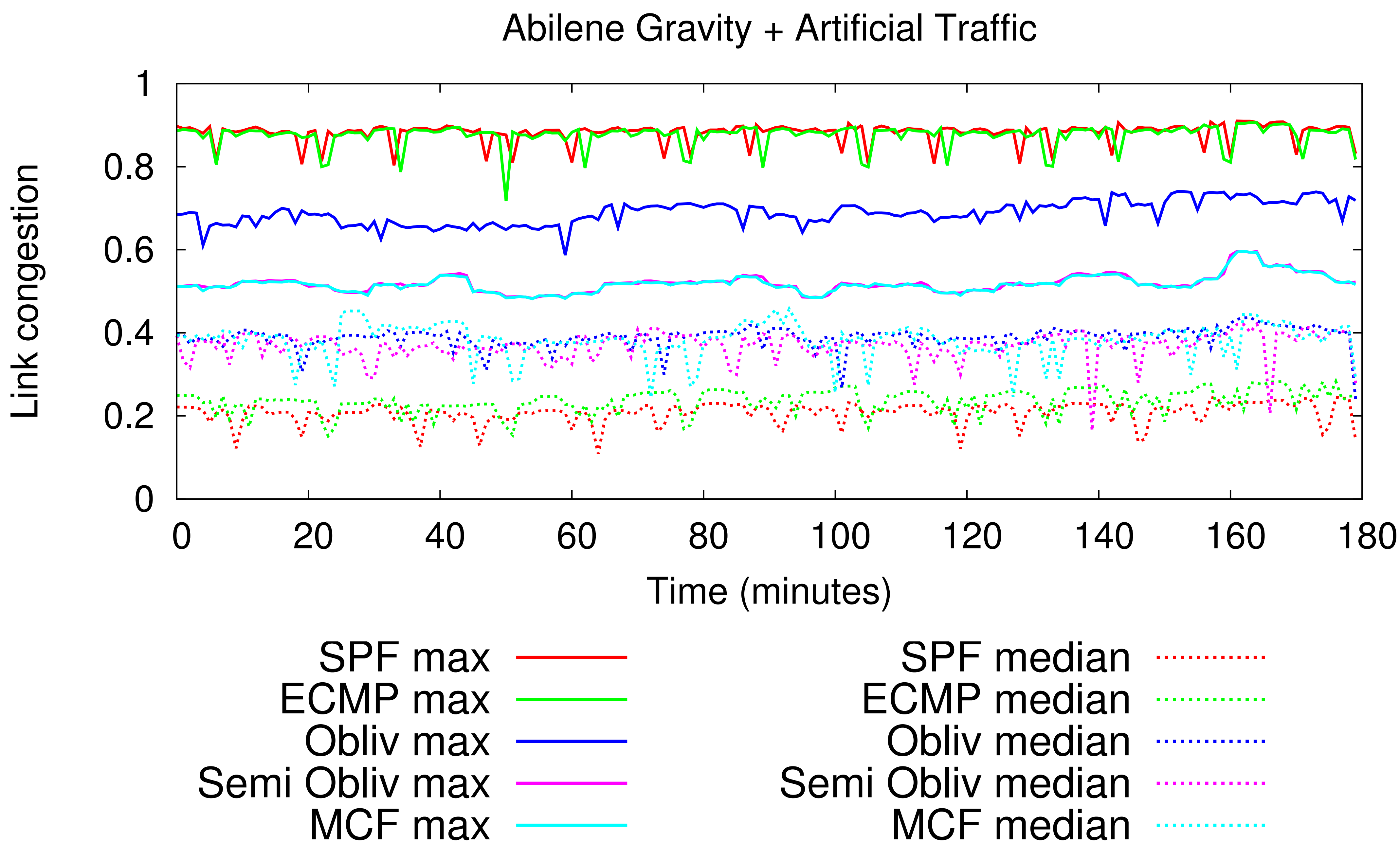
- Emulated Abilene topology in hardware test bed
- Used real-world and worst case traffic scenarios
- Compared shortest-path, ECMP, MCF, oblivious, and semi-oblivious

Abilene Topology

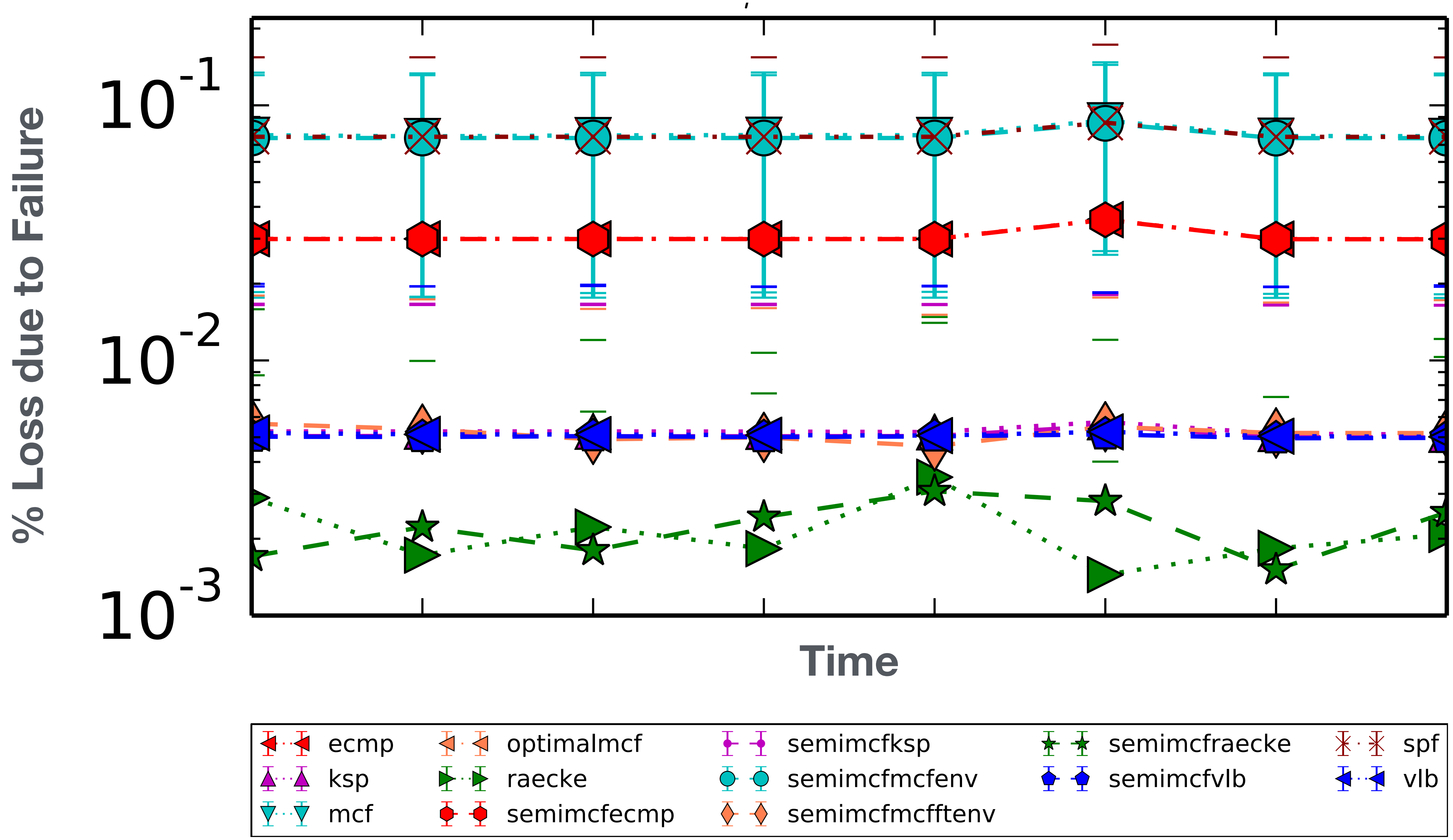


- Emulated Abilene topology in hardware test bed
- Used real-world and worst case traffic scenarios
- Compared shortest-path, ECMP, MCF, oblivious, and semi-oblivious

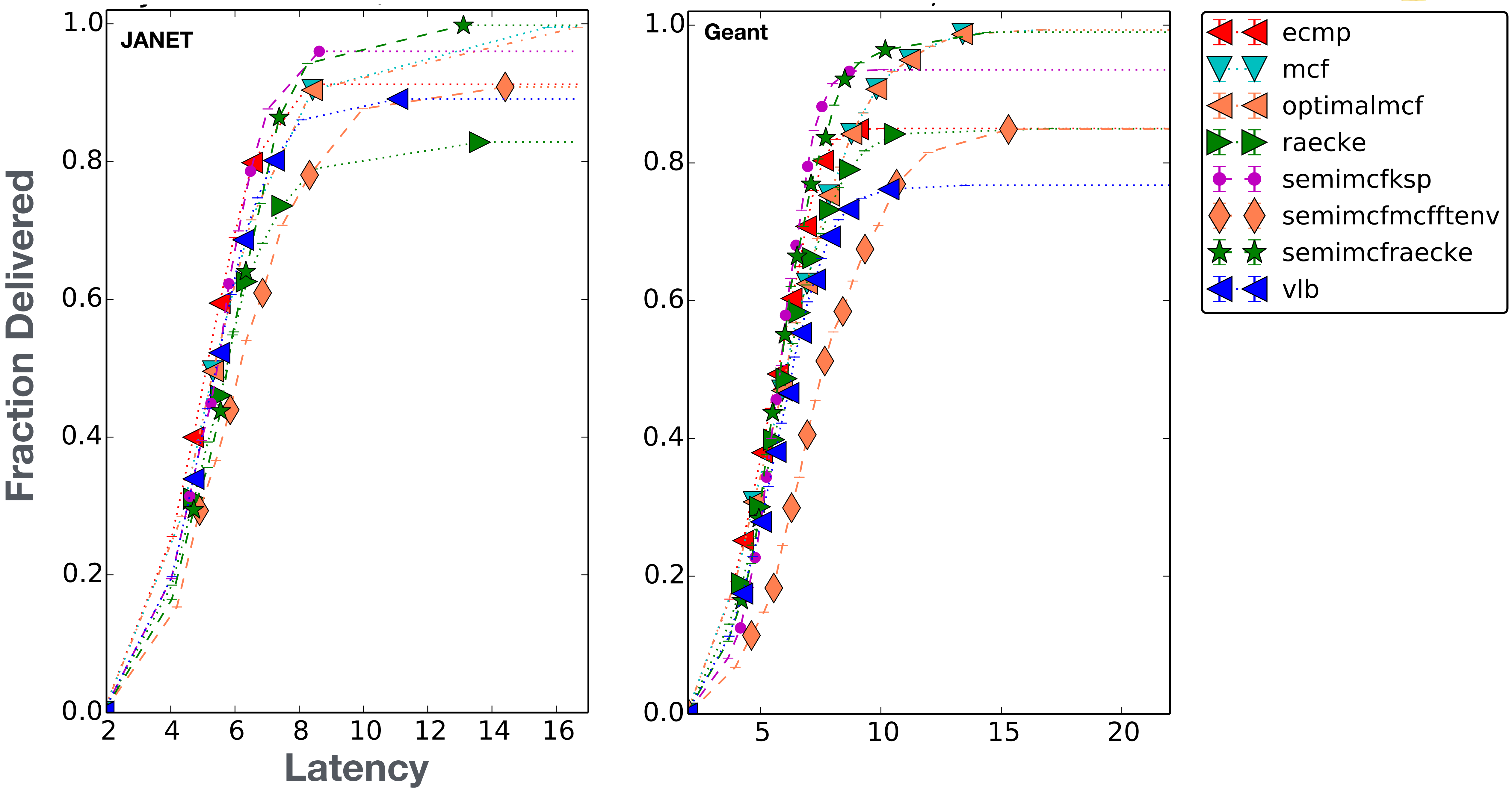
Abilene Topology: Simulated Workload



Topology Zoo: Failures



Selected Topology Zoo: Latency



Conclusions

- Randomization can dramatically simplify traffic engineering while balancing competing objectives
- Oblivious routing performs much better in practice than expected, avoids problems associated with churn, and load-balances better
- Semi-oblivious routing provides near-optimal performance in real-world scenarios, even in the presence of demand misprediction, traffic bursts, and failures
- Ongoing work: working with large ISP and content provider to further refine and evaluate Kulfi

Team Kulfi

Praveen Kumar

Yang Yuan

Chris Yu '15

Bobby Kleinberg

Robert Soulé

`https://github.com/merlin-lang/kulfi`

Topology Zoo, Traffic Burst