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
Abstract
Software-defined networking (SDN) programs must simultaneously describe stateful forwarding behavior and dynamic updates to response to network 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 stateful updates that are guaranteed to preserve well-defined behaviors when transitioning between configurations in response to events. We propose network event structures (NESs) and model continuity on updates, such as which events can be stable simultaneously and causal dependences between events. We further an extension of the NetKAT language with mutable state, eliminating the need for synchronization, side effects, and complex error handling. We develop 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.2.2 [Programming Languages]: Language Classifications—Specialized application languages; D.3.4 [Programming Languages]: Procedures—Concurrency

Keywords
network update; consumer update; event structure; software-defined networking; NES; NetKAT
Abstract. Event-driven networking (SDN) programs must simultaneously describe state forwarding behavior and dynamic updates to network resources. Even 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 novel concurrent update operators that are proven to preserve well-defined behavior when transitioning between configurations in response to events. We propose network event structures (NESs) as model structures for invariants, such as which events can be enabled simultaneously and causal dependencies between events. We define an extension of the NetKAT language with stateful SESs, give semantics to stateful programs using NESs, and discuss provably-correct strategies for implementing NESs in SDNs. Finally, we evaluate our approach on complex network scenarios, showing that our techniques are effective for programming networks with thousands of parallel events, processing concurrent events, managing asynchronous events, dealing with untrusted entities, etc. To address these challenges, we present a novel event-driven consistent updates framework, build on a domain-specific network programming language, and apply our techniques to real-world scenarios.

Keywords: Event Driven Programming, Event Driven Networking, Concurrent Programming, Network Event Structures, Network Abstractions, Network Invariants

1 Introduction

Software-defined networking (SDN) allows network behavior to be specified using logically-centralized programs that enforce behavior policies and apply them across network elements. The network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic. To implement correct dynamic SESs applications to date, the most effective option is to use low-level APIs, allowing the benefits of higher-level languages entirely.

Example [Stanford Foreword]. To illustrate the challenges that arise when implementing dynamic applications, consider a topology where an internal host I1 is connected to a switch s1, and switches s1, s2, and s3 are connected to each other (see Fig.

[ESOP ’16]

Event-Driven Network Programming

Jedidiah McClurg
CU Boulder, USA
jmclurg@colorado.edu

Hossein Hajjat
Cornell University, USA
hajjat@cs.cornell.edu

Nate Foster
Cornell University, USA
nfoster@cs.cornell.edu

Pavol Cerny
University of Pennsylvania
pavol.cerny@colorado.edu

Abstract

Event-driven networking (SDN) programs must simultaneously describe state-forwarding behavior and dynamic updates to network resources. Even-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 novel concurrent update operators that are proven to preserve well-defined behavior when transitioning between configurations in response to events. We propose network event structures (NESs) as model structures for invariants, such as which events can be enabled simultaneously and causal dependencies between events. We define an extension of the NetKAT language with stateful SESs, give semantics to stateful programs using NESs, and discuss provably-correct strategies for implementing NESs in SDNs. Finally, we evaluate our approach on complex network scenarios, showing that our techniques are effective for programming networks with thousands of parallel events, processing concurrent events, managing asynchronous events, dealing with untrusted entities, etc. To address these challenges, we present a novel event-driven consistent updates framework, build on a domain-specific network programming language, and apply our techniques to real-world scenarios.

Keywords: Event Driven Programming, Event Driven Networking, Concurrent Programming, Network Event Structures, Network Abstractions, Network Invariants

1 Introduction

Software-defined networking (SDN) allows network behavior to be specified using logically-centralized programs that enforce behavior policies and apply them across network elements. The network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic. To implement correct dynamic SESs applications to date, the most effective option is to use low-level APIs, allowing the benefits of higher-level languages entirely.

Example [Stanford Foreword]. To illustrate the challenges that arise when implementing dynamic applications, consider a topology where an internal host I1 is connected to a switch s1, and switches s1, s2, and s3 are connected to each other (see Fig.

[ESOP ’16]

Event-Driven Network Programming

Jedidiah McClurg
CU Boulder, USA
jmclurg@colorado.edu

Hossein Hajjat
Cornell University, USA
hajjat@cs.cornell.edu

Nate Foster
Cornell University, USA
nfoster@cs.cornell.edu

Pavol Cerny
University of Pennsylvania
pavol.cerny@colorado.edu

Abstract

Event-driven networking (SDN) programs must simultaneously describe state-forwarding behavior and dynamic updates to network resources. Even-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 novel concurrent update operators that are proven to preserve well-defined behavior when transitioning between configurations in response to events. We propose network event structures (NESs) as model structures for invariants, such as which events can be enabled simultaneously and causal dependencies between events. We define an extension of the NetKAT language with stateful SESs, give semantics to stateful programs using NESs, and discuss provably-correct strategies for implementing NESs in SDNs. Finally, we evaluate our approach on complex network scenarios, showing that our techniques are effective for programming networks with thousands of parallel events, processing concurrent events, managing asynchronous events, dealing with untrusted entities, etc. To address these challenges, we present a novel event-driven consistent updates framework, build on a domain-specific network programming language, and apply our techniques to real-world scenarios.

Keywords: Event Driven Programming, Event Driven Networking, Concurrent Programming, Network Event Structures, Network Abstractions, Network Invariants

1 Introduction

Software-defined networking (SDN) allows network behavior to be specified using logically-centralized programs that enforce behavior policies and apply them across network elements. The network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic. To implement correct dynamic SESs applications to date, the most effective option is to use low-level APIs, allowing the benefits of higher-level languages entirely.

Example [Stanford Foreword]. To illustrate the challenges that arise when implementing dynamic applications, consider a topology where an internal host I1 is connected to a switch s1, and switches s1, s2, and s3 are connected to each other (see Fig.

[ESOP ’16]

Event-Driven Network Programming

Jedidiah McClurg
CU Boulder, USA
jmclurg@colorado.edu

Hossein Hajjat
Cornell University, USA
hajjat@cs.cornell.edu

Nate Foster
Cornell University, USA
nfoster@cs.cornell.edu

Pavol Cerny
University of Pennsylvania
pavol.cerny@colorado.edu

Abstract

Event-driven networking (SDN) programs must simultaneously describe state-forwarding behavior and dynamic updates to network resources. Even-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 novel concurrent update operators that are proven to preserve well-defined behavior when transitioning between configurations in response to events. We propose network event structures (NESs) as model structures for invariants, such as which events can be enabled simultaneously and causal dependencies between events. We define an extension of the NetKAT language with stateful SESs, give semantics to stateful programs using NESs, and discuss provably-correct strategies for implementing NESs in SDNs. Finally, we evaluate our approach on complex network scenarios, showing that our techniques are effective for programming networks with thousands of parallel events, processing concurrent events, managing asynchronous events, dealing with untrusted entities, etc. To address these challenges, we present a novel event-driven consistent updates framework, build on a domain-specific network programming language, and apply our techniques to real-world scenarios.

Keywords: Event Driven Programming, Event Driven Networking, Concurrent Programming, Network Event Structures, Network Abstractions, Network Invariants

1 Introduction

Software-defined networking (SDN) allows network behavior to be specified using logically-centralized programs that enforce behavior policies and apply them across network elements. The network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic. To implement correct dynamic SESs applications to date, the most effective option is to use low-level APIs, allowing the benefits of higher-level languages entirely.

Example [Stanford Foreword]. To illustrate the challenges that arise when implementing dynamic applications, consider a topology where an internal host I1 is connected to a switch s1, and switches s1, s2, and s3 are connected to each other (see Fig.
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

West

East

Device Limitations
Challenges

- Sporadic shortcuts
- Device limitations
Challenges

- Sporadic shortcuts
- Sparse bisection
- Device limitations
Challenges

- Sporadic shortcuts
- Sparse bisection
- Unexpected Failures
- Device Limitations
Challenges

- Sporadic shortcuts
- Sparse bisection
- Device Limitations
- Unexpected Failures
- Misprediction & Bursts
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?
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

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

SWAN & B4 [SIGCOMM ’13]
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
1. Pre-compute a set of least-cost paths
2. Identify flows by hashing packet header fields
3. Randomly forward along least cost paths
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
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 a 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 $\Omega(\log(n)/\log (\log(n)))$ competitive ratio
- Realistic workloads are different from worst-case
SDN Implementation & Evaluation
Kulfi Framework

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

- 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

- 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

<table>
<thead>
<tr>
<th>Routing Algorithm</th>
<th>Description</th>
<th>Type</th>
<th>Path Diversity</th>
<th>Max Congestion</th>
<th>Overheads</th>
<th>Churn</th>
<th>Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MCF</strong></td>
<td>Multi-Commodity Flow solved with LP [20]</td>
<td>conscious</td>
<td>medium</td>
<td>least</td>
<td>high</td>
<td>slow</td>
<td></td>
</tr>
<tr>
<td><strong>MW</strong></td>
<td>Multi-Commodity Flow solved with Multiplicative Weights [17]</td>
<td>conscious</td>
<td>medium</td>
<td>least</td>
<td>high</td>
<td>slow</td>
<td></td>
</tr>
<tr>
<td><strong>SPF</strong></td>
<td>Shortest Path First</td>
<td>oblivious</td>
<td>least</td>
<td>high</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td><strong>ECMP</strong></td>
<td>Equal-Cost, Multi-Path</td>
<td>oblivious</td>
<td>low</td>
<td>high</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>KSP</strong></td>
<td>K-Shortest Paths</td>
<td>oblivious</td>
<td>medium</td>
<td>medium</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>Rücke</strong></td>
<td>Rücke [38]</td>
<td>oblivious</td>
<td>high</td>
<td>low</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>VLB</strong></td>
<td>Valiant Load Balancing [44]</td>
<td>oblivious</td>
<td>high</td>
<td>medium</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>SemiMCF-MCF</strong></td>
<td>MCF for paths</td>
<td>semi-oblivious</td>
<td>medium</td>
<td>least</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>SemiMCF-ECMP</strong></td>
<td>ECMP for paths</td>
<td>semi-oblivious</td>
<td>low</td>
<td>medium</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>SemiMCF-KSP</strong></td>
<td>KSP for paths</td>
<td>semi-oblivious</td>
<td>medium</td>
<td>medium</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>SemiMCF-Rücke</strong></td>
<td>Rücke for paths</td>
<td>semi-oblivious</td>
<td>high</td>
<td>low</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>SemiMCF-VLB</strong></td>
<td>VLB for paths</td>
<td>semi-oblivious</td>
<td>high</td>
<td>medium</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>SemiMCF-MCF-Env</strong></td>
<td>MCF over demand envelope for paths</td>
<td>semi-oblivious</td>
<td>medium</td>
<td>low</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td><strong>SemiMCF-MCF-FT-Env</strong></td>
<td>Multiple MCF-Env considering failures</td>
<td>semi-oblivious</td>
<td>high</td>
<td>medium</td>
<td>none</td>
<td>fast</td>
<td></td>
</tr>
</tbody>
</table>
### Kulfi Framework

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

- 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

- 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

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

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

- 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

- **Linux End Host**
  - **User-Space Agent**
  - **Netfilter Module**

- **SDN Controller**
  - **Traffic matrix + Path map**
  - **forwarding rules**
  - **data traffic**

- **SDN Switch**
  - **Historical Data**
    - Traffic statistics

- **Data Traffic**
  - Netfilter Module
  - User-Space Agent
  - Linux Kernel
  - Linux End Host
Hardware Testbed
Facebook Backbone: Simulation

![Graph showing simulation results with multiple lines representing Production, SPF, and Oblivious over time in days.](image)
Facebook Backbone: Simulation

Constant factor
Emulated Abilene topology in hardware test bed

Used real-world and worst case traffic scenarios

Compared shortest-path, ECMP, MCF, oblivious, and semi-oblivious
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

Abilene Gravity + Artificial Traffic

Time (minutes)

Link congestion

SPF max
ECMP max
Obliv max
Semi Obliv max
MCF max

SPF median
ECMP median
Obliv median
Semi Obliv median
MCF median
Topology Zoo: Failures

% Loss due to Failure

Time

- ecmp
- optimalmcf
- semimcfksp
- semimcfraecke
- spf
- ksp
- raecke
- semimcfmcfenv
- semimcfvlb
- vlb
- mcf
- semimcfecmp
- semimcfmcfftenv
Selected Topology Zoo: Latency

![Graph showing latency vs. fraction delivered for different topologies. The graph includes lines for various topologies such as JANET and Geant with markers for different protocols like ecmp, mcf, optimalmcf, raecke, semimcfksp, semimcfmcffttenv, semimcfraecke, and vlb. The x-axis represents latency, and the y-axis represents the fraction delivered.]
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