Toward Self-* Networks

Stefan Schmid (Uni Vienna)
A Great Time to Be a Networking Researcher!

Innovation

Rhone and Arve Rivers, Switzerland

Credits: George Varghese.
Flexibilities: Along 3 Dimensions
Flexibilities: Along 3 Dimensions
Flexibilities: Along 3 Dimensions

Enabler: SDN
Enabler: Virtualization
Enabler: Optics
Rewinding the clock of the Internet to a decade ago...
Rewinding the clock of the Internet...

Shortest path routing only

Indirect control: via weights only

Proprietary, blackbox implementations

Difficult and slow innovation

Kudos to: Pedro Casas
Opportunity: Flexible Routing

- **Direct control** over paths
  - Traditionally: indirect control *via weights* based on which *shortest paths* are computed

- **More general routes**
  - Beyond shortest paths, even *beyond „paths“*
  - E.g., steer traffic through (virtualized) middleboxes to compose new services like *service chains* („walk“)

- **General match-action**
  - SDN allows to match *L2-L4* and route, e.g., HTTP traffic differently (e.g., to cache)

---

Opportunity: Flexible Embedding

Guest (e.g., VNet)

- Flexibly allocate (virtualized) network functions or map (virtualized) communication partners...
- ... to improve utilization, minimize latency and load, etc.

Enabler: Virtualization

Cost 5

Host

Charting the Complexity Landscape of Virtual Network Embeddings. Rost et al. IFIP Networking, 2018.
Opportunity: Flexible Embedding

Guest (e.g., VNet)

- Flexibly allocate (virtualized) network functions or map (virtualized) communication partners...
- ... to improve utilization, minimize latency and load, etc.

Host

Cost 1

Enabler: Virtualization

Charting the Complexity Landscape of Virtual Network Embeddings. Rost et al. IFIP Networking, 2018.
The Internet: Capable of Change on All Layers!

- Application Layer
- Transport Layer
- Network Layer
- Link + Physical Layer
The Internet: Capable of Change on All Layers!

Based on free-space optics, 60GHz, optical circuit switches, movable antennas and mirrors, etc.
Opportunity: Flexible Topology Programming

- Reconfigure networks towards needs

Enabler: Free-space optics

Opportunity: Flexible Topology Programming

• **Reconfigure** networks towards needs

Enabler: Free-space optics

---

**Toward Demand-Aware Networking: A Theory for Self-Adjusting Networks.**
Timeline

Reconfiguration time: from milliseconds to microseconds (and decentralized).

<table>
<thead>
<tr>
<th>Opportunity</th>
<th>Challenge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Great <strong>optimization opportunities</strong></td>
<td>Operating networks may become more <strong>complex</strong>, e.g.: traversal of firewall not mapped to „edge“</td>
</tr>
<tr>
<td>In principle flexibilities can be exploited „fast“: <strong>open interfaces</strong> (<em>bring your own algorithm!</em></td>
<td>Exploiting <strong>online optimization</strong> at runtime hard at <strong>human</strong> time scale: difficult algorithmic problems</td>
</tr>
<tr>
<td>Also easier to <strong>collect data</strong>: programmable networks, <strong>telemetry</strong></td>
<td><strong>Modelling</strong> can be <strong>more difficult</strong> too: new components like <strong>hypervisor</strong> can affect performance</td>
</tr>
</tbody>
</table>
You: I invented a great new algorithm to route and embed service chains at low resource cost and providing minimal bandwidth guarantees!

Boss: So can I promise our customers a predictable performance?

You: hmm…
To enable multi-tenancy, take existing network hypervisor (e.g. Flowvisor, OpenVirteX): provides network abstraction and control plane translation!

Recall Andi Blenk’s Talk

An Experiment: 2 vSDNs with bw guarantee!
To enable **multi-tenancy**, take existing network **hypervisor** (e.g. Flowvisor, OpenVirteX): provides network abstraction and control plane translation!

An Experiment: 2 vSDNs with bw guarantee!
• Exploiting network flexibilities is non-trivial, especially if fine-grained and fast reactions are desired

• Also modelling such networked systems is challenging: details of interference, demand, etc. will only be available at runtime

Optimal operation of flexible networks too complex for humans.
Let’s give up control: self-* networks!

Self-observing, self-adjusting, self-repairing, “self-driving”, ...
Roadmap

• Opportunities of self-* networks
  – Example 1: Demand-aware, self-adjusting networks
  – Example 2: Self-repairing networks

• Challenges of designing self-* networks
Roadmap

- Opportunities of self-* networks
  - Example 1: Demand-aware, self-adjusting networks
  - Example 2: Self-repairing networks

- Challenges of designing self-* networks
Why Demand-Aware...?

Case study: datacenter networks
Explosive Growth of Demand...

Batch processing, web services, distributed ML, ...: data-centric applications are distributed and interconnecting network is critical

... But Much Structure!

Spatial (sparse!) and temporal locality

Understanding Data Center Traffic Characteristics @ WREN 2009

ProjecToR @ SIGCOMM 2016
Explosive Growth of Demand...

Batch processing, web services, distributed ML, ...: data-centric applications are distributed and interconnecting network is critical.

... But Much Structure!

Demand-Aware Networks (DANs) can exploit this structure by adapting to it: „DANs can provide same performance as demand-oblivious networks at 25-40% lower costs.“ Firefly, SIGCOMM CCR, 2014.

Source: Jupiter Rising. SIGCOMM 2015.

Aggregate server traffic in Google’s datacenter fleet

Source: Understanding Data Center Traffic Characteristics @ WREN 2009
Datacenter Networks

Traditionally: demand-oblivious:
Datacenter Networks

Traditionally: demand-obliviious:
Datacenter Networks

Traditional datacenter network

- Usually optimized for the “worst-case” (all-to-all communication)
- Example, fat-tree topologies: provide full bisection bandwidth
Datacenter Networks

Traditional datacenter network

- Usually optimized *for the “worst-case”* (all-to-all communication)
- Example, fat-tree topologies: provide **full bisection bandwidth**

*Reconfigurable* datacenter network

- Optimized *toward the workload* it serves (e.g., route length)
- Statically or **even dynamically**

**Mirrors →**

**Lasers →**

**TOR switches →**
Datacenter Networks

Traditional datacenter network

Usually optimized for the “worst-case” (all-to-all communication).

- Example, fat-tree topologies: provide full bisection bandwidth.

Reconfigurable datacenter network

- Optimized toward the workload it serves (e.g., route length).
- Statically or even dynamically.

Graph theory: logarithmic route lengths in constant-degree datacenter networks.
Datacenter Networks

Traditional datacenter network

- Usually optimized for the "worst-case" (all-to-all communication).
- Example, fat-tree topologies: provide full bisection bandwidth.

Reconfigurable datacenter network

- Mirrors → Lasers → TOR switches →
- What can be achieved in DANs? Metrics?
  - Optimized towards network and it serves (e.g., route length).
  - Statically or even dynamically.

Graph theory: logarithmic route lengths in constant-degree datacenter networks.
DAN Design: New Types of Problems

Input: Workload

Output: DAN

Demand matrix: joint distribution... of constant degree (scalability)
DAN Design: New Types of Problems

Input: Workload

<table>
<thead>
<tr>
<th>Sources</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2/65</td>
<td>1/13</td>
<td>1/65</td>
<td>1/65</td>
<td>2/65</td>
<td>3/65</td>
</tr>
<tr>
<td>2</td>
<td>2/65</td>
<td>0</td>
<td>1/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2/65</td>
</tr>
<tr>
<td>3</td>
<td>1/13</td>
<td>65</td>
<td>0</td>
<td>2/65</td>
<td>0</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>1/65</td>
<td>0</td>
<td>2/65</td>
<td>0</td>
<td>4/65</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1/65</td>
<td>0</td>
<td>3/65</td>
<td>4/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3/65</td>
</tr>
<tr>
<td>7</td>
<td>3/65</td>
<td>2/65</td>
<td>1/13</td>
<td>0</td>
<td>0</td>
<td>3/65</td>
<td>0</td>
</tr>
</tbody>
</table>

Destinations

Demand matrix: joint distribution

Output: DAN

Much from 4 to 5.

design

Makes sense to add link!

... of constant degree (scalability)
DAN Design: New Types of Problems

Input: Workload
Output: DAN

Demand matrix: joint distribution... of constant degree (scalability)

1 communicates to many.

Bounded degree: route to 7 indirectly.

Sources

Destinations

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2/65</td>
<td>1/13</td>
<td>1/65</td>
<td>1/65</td>
<td>2/65</td>
<td>3/65</td>
</tr>
<tr>
<td>2</td>
<td>2/65</td>
<td>0</td>
<td>1/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2/65</td>
</tr>
<tr>
<td>3</td>
<td>1/13</td>
<td>1/65</td>
<td>0</td>
<td>2/65</td>
<td>0</td>
<td>0</td>
<td>1/13</td>
</tr>
<tr>
<td>4</td>
<td>1/65</td>
<td>0</td>
<td>2/65</td>
<td>0</td>
<td>4/65</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1/65</td>
<td>0</td>
<td>3/65</td>
<td>4/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3/65</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>3/65</td>
<td>2/65</td>
<td>1/13</td>
<td>0</td>
<td>0</td>
<td>3/65</td>
<td>0</td>
</tr>
</tbody>
</table>
DAN Design: New Types of Problems

Input: Workload

<table>
<thead>
<tr>
<th>Sources</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2/65</td>
<td>1/13</td>
<td>1/65</td>
<td>1/65</td>
<td>1/65</td>
<td>2/65</td>
</tr>
<tr>
<td>2</td>
<td>2/65</td>
<td>0</td>
<td>1/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2/65</td>
</tr>
<tr>
<td>3</td>
<td>1/13</td>
<td>1/65</td>
<td>0</td>
<td>2/65</td>
<td>0</td>
<td>0</td>
<td>1/13</td>
</tr>
<tr>
<td>4</td>
<td>1/65</td>
<td>0</td>
<td>2/65</td>
<td>0</td>
<td>4/65</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1/65</td>
<td>0</td>
<td>3/65</td>
<td>4/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2/65</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>3/65</td>
<td>2/65</td>
<td>1/13</td>
<td>0</td>
<td>0</td>
<td>3/65</td>
<td>0</td>
</tr>
</tbody>
</table>

Demand matrix: joint distribution

Destinations

Output: DAN

... but “extra” link still makes sense: not a subgraph.

4 and 6 don’t communicate...

... of constant degree (scalability)
More Formally: DAN Design Problem

**Input:**

\[\mathcal{D}[p(i, j)]: \text{joint distribution, } \Delta\]

**Output:**

\(N: \text{DAN}\)

Bounded degree \(\Delta = 3\)

**Objective:**

**Expected Path Length (EPL):**

Demand-weighted route length

\[\text{EPL}(\mathcal{D}, N) = \sum_{(u,v) \in \mathcal{D}} p(u, v) \cdot d_N(u, v)\]

Path length on DAN \(N\).
Sometimes, DANs can be much better!

**Example 1:** low-degree demand
- Already low degree: degree-4 DAN can serve this **at cost 1**.

**Example 2:** high-degree but skewed demand
- If sufficiently skewed: constant-degree DAN can serve it at cost $O(1)$.
So on what does it depend?
So on what does it depend?

We argue (but still don’t know!): on the “entropy” of the demand!
An Analogy to Coding

if demand *arbitrary* and *unknown*

<table>
<thead>
<tr>
<th>worst case network:</th>
<th>worst case coding:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full BWV</td>
<td>00, 01, 10, 11</td>
</tr>
</tbody>
</table>

*log diameter*

*log # bits / symbol*
An Analogy to Coding

if demand **arbitrary** and **unknown**

DAN!

if demand **known** and **fixed**

entropy?  
entropy / symbol

```
static case network: Full BW
worst case coding: 00, 01, 10, 11
log diameter
```

```
static Demand-Aware Nets
static Huffman: 1, 01, 001, 000
log # bits / symbol
```

„Coming to IFIP Networking in Warsaw?“
An Analogy to Coding

if demand **arbitrary** and **unknown**

DAN!

if demand **known** and **fixed**

if demand **unknown** but **reconfigurable**

SAN!

 Dynamic DANs: Aka. Self-Adjusting Networks (SANs)!

• **worst case network:** Full BW
  - log diameter

• **worst case coding:** 00, 01, 10, 11
  - log # bits / symbol

• **entropy?**

• **entropy / symbol**

• **static Demand-Aware Nets**
  - static Huffman: 1, 01, 001, 000

• **dynamic Demand-Aware Nets**
  - dynamic Huffman codes

„Coming to IFIP Networking in Warsaw?“
An Analogy to Coding

DAN!

if demand **arbitrary** and **unknown**

worst case network: Full BW
worst case coding: 00, 01, 10, 11

log diameter
log # bits / symbol

Dynamic DANs: Aka. Self-Adjusting Networks (SANs)!

SAN!

if demand **known** and **fixed**

Can exploit **spatial locality**!

if demand **unknown** but **reconfigurable**

Can exploit **temporal locality**!

if demand **unknown** but **reconfigurable**

**Coming to IFIP Networking in Warsaw?**
An Analogy to Coding

if demand *arbitrary* and *unknown*

if demand *known* or *fixed*

if demand *unknown* but *reconfigurable*

DAN!  
Dynamic DANs: Aka. Self-Adjusting Networks (SANs)!

SAN!  
Need online algorithms!

Additionally exploit *temporal locality*!

"Cheating": need to know demand!

Can exploit *spatial* locality!
Analogous to **Datastructures**: Oblivious...

- Traditional, **fixed** BSTs do not rely on any assumptions on the demand
- Optimize for the **worst-case**
- Example **demand**: 
  \[1, \ldots, 1, 3, \ldots, 3, 5, \ldots, 5, 7, \ldots, 7, \ldots, \log(n), \ldots, \log(n)\]
- Items stored at **\(O(\log n)\)** from the root, **uniformly** and **independently** of their frequency

Corresponds to **max possible demand!**
Demand-aware fixed BSTs can take advantage of spatial locality of the demand.

E.g.: place frequently accessed elements close to the root.

E.g., Knuth/Mehlhorn/Tarjan trees.

Recall example demand: 1,...,1,3,...,3,5,...,5,7,...,7,...,log(n),...,log(n)

- Amortized cost $O(\log\log n)$

Amortized cost corresponds to empirical entropy of demand!
• **Demand-aware reconfigurable** BSTs can additionally take advantage of *temporal locality*

• By moving accessed element to the root: amortized cost is **constant**, i.e., $O(1)$
  - Recall example demand: $1, \ldots, 1, 3, \ldots, 3, 5, \ldots, 5, 7, \ldots, 7, \ldots, \log(n), \ldots, \log(n)$
Datastructures

Oblivious

Demand-Aware

Self-Adjusting

Lookup
$O(\log n)$

Exploit spatial locality:
empirical entropy $O(\log\log n)$

Exploit temporal locality as well:
$O(1)$
Analogously for Networks

Oblivious

Const degree (e.g., expander):
route lengths $O(\log n)$

DAN

Exploit spatial locality

SAN

Exploit temporal locality as well

Intuition: Entropy Lower Bound
Lower Bound Idea: Leverage Coding or Datastructure

- DAN just for a single (source) node 1: cannot do better than Δ-ary Huffman tree (or a biased BST) for its destinations

- How good can this tree be?

**Entropy** lower bound on EPL known for binary trees, e.g. Mehlhorn 1975 for BST

<table>
<thead>
<tr>
<th>Sources</th>
<th>Destinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>2\text{ 65} 0 \text{ 65} 0 0 0 2\text{ 65}</td>
</tr>
<tr>
<td>3</td>
<td>1\text{ 13} 1\text{ 65} 0 0 0 1\text{ 13}</td>
</tr>
<tr>
<td>4</td>
<td>1\text{ 65} 0 2\text{ 65} 4\text{ 65} 0 0</td>
</tr>
<tr>
<td>5</td>
<td>1\text{ 65} 0 3\text{ 65} 4\text{ 65} 0 0 0</td>
</tr>
<tr>
<td>6</td>
<td>2\text{ 65} 0 0 0 0 0 3\text{ 65}</td>
</tr>
<tr>
<td>7</td>
<td>3\text{ 65} 2\text{ 65} 1\text{ 13} 0 0 3\text{ 65} 0</td>
</tr>
</tbody>
</table>
Lower Bound Idea:
Leverage Coding or Datastructure

<table>
<thead>
<tr>
<th>Sources</th>
<th>Destinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 2/65 1/65 1/65 1/65 2/65 3/65</td>
</tr>
<tr>
<td>2</td>
<td>2/65 0 1/65 0 0 0 2/65</td>
</tr>
<tr>
<td>3</td>
<td>1/13 1/65 0 2/65 0 0 1/13</td>
</tr>
<tr>
<td>4</td>
<td>1/65 0 2/65 0 4/65 0 0</td>
</tr>
<tr>
<td>5</td>
<td>1/65 0 3/65 4/65 0 0 0</td>
</tr>
<tr>
<td>6</td>
<td>2/65 0 0 0 0 0 3/65</td>
</tr>
<tr>
<td>7</td>
<td>3/65 2/65 1/13 0 0 3/65 0</td>
</tr>
</tbody>
</table>

- DAN just for a single (source) node 1: cannot do better than $\Delta$-ary Huffman tree (or a biased BST) for its destinations.
- How good can this tree be?

**Entropy** lower bound on EPL known for binary trees, e.g. Mehlhorn 1975 for BST.
So: Entropy of the *Entire* Demand

- **Proof idea** \((\text{EPL} = \Omega(H_\Delta(Y|X)))\):
  - Compute *ego-tree* for each source node
  - Take *union* of all *ego-trees*
  - Violates *degree restriction* but valid lower bound
Entropy of the * Entire Demand: Sources and Destinations*

Do this in **both dimensions**: 

\[ \text{EPL} \geq \Omega(\max\{H_{\Delta}(Y|X), H_{\Delta}(X|Y)\}) \]
Intuition: Reaching Entropy Limit in Datacenters
Ego-Trees Revisited

- ego-tree: optimal tree for a row (= given source)
Ego-Trees Revisited

- ego-tree: optimal tree for a row (= given source)

Can we merge the trees *without distortion* and *keep degree low*?
Ego-Trees Revisited

- ego-tree: optimal tree for a row (= given source)

For sparse demands yes: enough low-degree nodes which can serve as “helper nodes”!

Can we merge the trees without distortion and keep degree low?
Ego-Trees Revisited

- ego-tree: optimal tree for a row (= given source)

Ego-tree can also be *dynamic*, i.e. *self-adjusting*!

Can we merge the trees *without distortion* and *keep degree low*?
Other metrics for self-adjusting networks?
A Taxonomy: Reconfigurable Networks

Static Optimality:
“Not worse than static which knows demand ahead of time!”

\[ \rho = \frac{\text{Cost(ON)}}{\text{Cost(STAT)}} \]
is constant.
A Taxonomy: Reconfigurable Networks

Static Optimality:

“Not worse than static which knows demand ahead of time!”

$\rho = \frac{\text{Cost(ON)}}{\text{Cost(STAT*)}}$ is constant.

**Note:** may be <<1. ON has advantage of adjusting, but the disadvantage of not knowing the workload. E.g. if much temporal locality.
A Taxonomy: Reconfigurable Networks

Working Set Property:

“Topological distance between nodes proportional to how recently they communicated!”

Revealed over time: learning or online algorithm

Off

Online

Static Optimality

Working Set

Demand-Aware

Reconfigurable

Topology

Input

Algorithm

Property
A Taxonomy: Reconfigurable Networks

Dynamic Optimality:

“No worse than an offline algorithm which knows the sequence!”

\[ \rho = \frac{\text{Cost(ON)}}{\text{Cost(OFF*)}} \]

is constant.

Always \( \geq 1 \).

The holy grail!
So: How much structure/entropy is there?

How to measure it?
How to distinguish between temporal and non-temporal structure?
More tricky!
Often only intuitions in the literature...

“less than 1% of the rack pairs account for 80% of the total traffic”

“only a few ToRs switches are hot and most of their traffic goes to a few other ToRs”

“over 90% bytes flow in elephant flows”
... and it is intuitive!
Non-temporal Structure

Traffic matrix of two different distributed ML applications (GPU-to-GPU):
Which one has more structure?
... and it *is* intuitive!

Non-temporal Structure

More uniform

More skewed

Traffic matrix of two different *distributed ML* applications (GPU-to-GPU):

Which one has *more structure*?
Two different ways to generate *same traffic matrix* (same non-temporal structure):
Which one has *more structure*?

... and it *is* intuitive!

Temporal Structure

VS
Two different ways to generate *same traffic matrix* (same non-temporal structure):
Which one has *more structure*?
... and it *is* intuitive!

Temporal Structure

Two different ways to generate *same traffic matrix* (same non-temporal structure):

Which one has *more structure*?

Quite intuitive: but how to define and measure systematically?

More bursty

More random
The Trace Complexity

- An **information-theoretic**: what is the entropy (rate) of a traffic trace?

- Systematic „**shuffle&compress““ methodology
  - Remove structure by iterative *randomization*
  - Difference of compression *before and after* randomization: structure
The Trace Complexity

Original src-dst trace → Randomize rows → Randomized columns → Uniform trace

Increasing complexity (systematically randomized)

More structure (compresses better)
The Trace Complexity

Original src-dst trace

Randomize rows

Randomized columns

Uniform trace

Remove temporal locality

Break src-dst pairs

Remove non-temporal locality

Difference in compression?

Difference in compression?

Difference in compression?
The Trace Complexity

Original src-dst trace

Randomize rows

Randomized columns

Uniform trace

Difference in compression?

Difference in compression?

Difference in compression?

Can be used to define a „complexity map“!
The Complexity Map

Complexity Map: Entropy ("complexity") of traffic traces.

The Complexity Map

Complexity Map: Entropy ("complexity") of traffic traces.

The Complexity Map

Complexity Map: Entropy ("complexity") of traffic traces.

Size = product of entropy

Uniform: Today’s datacenters

• Traditional networks are optimized for the “worst-case” (all-to-all communication traffic)
• Example, fat-tree topologies: provide full bisection bandwidth
The Complexity Map

Good in the worst case but: cannot leverage different temporal and non-temporal structures of traffic traces!
Non-temporal structure could be exploited already with static demand-aware networks! Good in the worst case but: cannot leverage different temporal and non-temporal structures of traffic traces!
To exploit **temporal** structure, need **adaptive demand-aware** ("self-adjusting") networks.

Non-temporal structure could be exploited already with **static demand-aware networks**!

**Good** in the worst case **but**: cannot leverage different **temporal** and **non-temporal** structures of traffic traces!
The Complexity Map

**Observation**: different applications feature quite significant (and different!) temporal and non-temporal structures.

- **Facebook** clusters: DB, WEB, HAD
- **HPC** workloads: CNS, Multigrid
- Distributed **Machine Learning** (ML)
- Synthetic traces like **pFabric**
The Complexity Map

Goal: Design self-adjusting networks which leverage both dimensions of structure!

Both structures!
The Complexity Map

Potential gain / tax of self-adjusting networks!

No structure!

Goal: Design self-adjusting networks which leverage both dimensions of structure!

Both structures!
Roadmap

• Opportunities of self-* networks
  – Example 1: Demand-aware, self-adjusting networks
  – Example 2: Self-repairing networks

• Challenges of desinging self-* networks
Reasoning About Failures is Hard

Example: BGP in Datacenter (!)

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.
Reasoning About Failures is Hard

Example: BGP in Datacenter (!)

Cluster with services that should be **globally reachable**.

Cluster with services that should be accessible **only internally**.

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.
Reasoning About Failures is Hard

Example: X and Y *announce* to Internet what is from G* (prefix).

X and Y *block* what is from P*.

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.
Reasoning About Failures is Hard

Example: Datacenter

X and Y announce to Internet what is from G* (prefix).

X and Y block what is from P*.

What can go wrong?

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.
Reasoning About Failures is Hard

Example: BGP in Datacenter

X and Y *announce* to Internet what is from G* (prefix).

X and Y *block* what is from P*.

If link (G,X) fails and traffic from G is rerouted via Y and C to X: X announces (does not block) G and H as it comes from C. (Note: BGP.)

Credits: Beckett et al. (SIGCOMM 2016): Bridging Network-wide Objectives and Device-level Configurations.
Managing Complex Networks is Hard for Humans

Another Case for Automation!
Case Study: Self-Repairing MPLS Networks

- MPLS: forwarding based on top label of label stack
Case Study: Self-Repairing MPLS Networks

- MPLS: forwarding based on top label of label stack

Diagram:
- Default routing of two flows
- Flow 1 and Flow 2
- Nodes V1, V2, V3, V4, V5, V6, V7, V8
Case Study: Self-Repairing MPLS Networks

- MPLS: forwarding based on top label of label stack

\[ \text{Default routing of two flows} \]
Fast Reroute Around 1 Failure

- MPLS: forwarding based on top label of label stack

![Diagram showing network topology and label manipulation]

If \((v_2, v_3)\) failed, push 30 and forward to \(v_6\).

For failover: push and pop label

Default routing of two flows

One failure: push 30: route around \((v_2, v_3)\)
Fast Reroute Around 1 Failure

- MPLS: forwarding based on top label of label stack

  - For failover: push and pop label

  What about multiple link failures?

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

  Default routing of two flows
2 Failures: Push *Recursively*

**Original Routing**

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

**Two failures:**
- first push 30: route around \((v_2, v_3)\)
- **Push recursively** 40: route around \((v_2, v_6)\)
2 Failures: Push \textit{Recursively}

Original Routing

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

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

Push recursively 40: route around \((v_2, v_6)\)
2 Failures: Push *Recursively*

One failure: push 30:

But masking links one-by-one can be inefficient:

(v_7, v_3, v_8) could be shortcut to (v_7, v_8).

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

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

Push recursively 40:
route around (v_2, v_6)

Original Routing

Pushing around (v_2, v_3)
Reasoning About Low-Level Rules is Hard

Flow Table

<table>
<thead>
<tr>
<th>FT</th>
<th>In-I</th>
<th>In-Label</th>
<th>Out-I</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ_{v_1}</td>
<td>\textit{in}_1</td>
<td>\perp</td>
<td>(v_1, v_2)</td>
<td>push</td>
</tr>
<tr>
<td></td>
<td>\textit{in}_2</td>
<td>\perp</td>
<td>(v_1, v_2)</td>
<td>push</td>
</tr>
<tr>
<td>τ_{v_2}</td>
<td>(v_1, v_2)</td>
<td>10</td>
<td>(v_2, v_3)</td>
<td>swap(11)</td>
</tr>
<tr>
<td></td>
<td>(v_1, v_2)</td>
<td>20</td>
<td>(v_2, v_3)</td>
<td>swap(21)</td>
</tr>
<tr>
<td>τ_{v_3}</td>
<td>(v_2, v_3)</td>
<td>11</td>
<td>(v_3, v_4)</td>
<td>swap(12)</td>
</tr>
<tr>
<td></td>
<td>(v_2, v_3)</td>
<td>21</td>
<td>(v_3, v_8)</td>
<td>swap(22)</td>
</tr>
<tr>
<td></td>
<td>(v_7, v_3)</td>
<td>11</td>
<td>(v_3, v_4)</td>
<td>swap(12)</td>
</tr>
<tr>
<td></td>
<td>(v_7, v_3)</td>
<td>21</td>
<td>(v_3, v_8)</td>
<td>swap(22)</td>
</tr>
<tr>
<td>τ_{v_4}</td>
<td>(v_3, v_4)</td>
<td>12</td>
<td>\textit{out}_1</td>
<td>pop</td>
</tr>
<tr>
<td>τ_{v_5}</td>
<td>(v_3, v_4)</td>
<td>40</td>
<td>\textit{out}_1</td>
<td>pop</td>
</tr>
<tr>
<td>τ_{v_6}</td>
<td>(v_3, v_4)</td>
<td>12</td>
<td>\textit{out}_1</td>
<td>pop</td>
</tr>
<tr>
<td>τ_{v_7}</td>
<td>(v_5, v_6)</td>
<td>11</td>
<td>(v_6, v_7)</td>
<td>swap(62)</td>
</tr>
<tr>
<td></td>
<td>(v_6, v_7)</td>
<td>31</td>
<td>(v_7, v_3)</td>
<td>pop</td>
</tr>
<tr>
<td></td>
<td>(v_6, v_7)</td>
<td>62</td>
<td>(v_7, v_3)</td>
<td>swap(11)</td>
</tr>
<tr>
<td></td>
<td>(v_6, v_7)</td>
<td>72</td>
<td>(v_7, v_8)</td>
<td>swap(22)</td>
</tr>
<tr>
<td>τ_{v_8}</td>
<td>(v_3, v_8)</td>
<td>22</td>
<td>\textit{out}_2</td>
<td>pop</td>
</tr>
<tr>
<td></td>
<td>(v_7, v_8)</td>
<td>22</td>
<td>\textit{out}_2</td>
<td>pop</td>
</tr>
</tbody>
</table>

Protected link

Alternative link

Label

Version which does not mask links individually!

Failover Tables

Tables for our example
MPLS Tunnels in Today’s ISP Networks
Responsibilities of a Sysadmin

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

- **Reachability**: Can traffic from ingress port A reach egress port B?
- **Loop-freedom**: Are the routes implied by the forwarding rules loop-free?
- **Non-reachability**: Is it ensured that traffic originating from A never reaches B?
- **Waypoint assurance**: Is it ensured that traffic from A to B is always routed via a node C (e.g., a firewall)?
Responsibilities of a Sysadmin

Sysadmin responsible for:
• **Reachability**: Can traffic from ingress port A reach egress port B?

Reachability?
Responsibilities of a Sysadmin

**Sysadmin** responsible for:

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

**Sysadmin** responsible for:

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

E.g. **NORDUnet**: no traffic via Iceland (expensive!). Or no traffic through route reflectors.
Responsibilities of a Sysadmin

Sysadmin responsible for:

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

E.g. IDS, firewall
Sysadmin responsible for:

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

... and everything even under multiple failures?!
Can we automate such tests or even self-repair?
Can we automate such tests or even self-repair?

Yes! Encouraging: sometimes even fast: What-if Analysis Tool for MPLS and SR
Leveraging Automata-Theoretic Approach

MPLS configurations, Segment Routing etc.

What if...?!

Compilation

pX ⇒ qXX
pX ⇒ qYX
qY ⇒ rYY
rY ⇒ r
rX ⇒ pX

Interpretation

Pushdown Automaton and Prefix Rewriting Systems Theory
Leveraging Automata-Theoretic Approach

MPLS configurations, Segment Routing etc.

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

What if...?!

Compilation

\[
pX \Rightarrow qXX
\]

\[
pX \Rightarrow qYX
\]

\[
qY \Rightarrow rYY
\]

\[
rY \Rightarrow r
\]

\[
rX \Rightarrow pX
\]

Interpretation

Pushdown Automaton and *Prefix Rewriting Systems* Theory
Mini-Tutorial: A Network Model

- Network: a 7-tuple

\[ N = (V, E, I_{in}, I_{out}, \lambda, L, \delta_F) \]
Mini-Tutorial: A Network Model

- Network: a 7-tuple

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

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

\[ \lambda_v : I_v^{in} \cup I_v^{out} \rightarrow V \]

That is: \((\lambda_v(in), v) \in E\) and \((v, \lambda_v(out)) \in E\)
Mini-Tutorial: A Network Model

• Network: a 7-tuple

\[ N = (V, E, I^\text{in}_v, I^\text{out}_v, \lambda_v, L, \delta^F_v) \]

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

\[ \delta^F_v : I^\text{in}_v \times L^* \rightarrow 2(I^\text{out}_v \times L^*) \]

defines, for all incoming interfaces and packet headers, outgoing interfaces together with modified headers.
Packet routing sequence can be represented using sequence of tuples:

\[(v_i, in_i, h_i, out_i, h_{i+1}, F_i)\]

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

\[(v_1, in_1, h_1, out_1, h_2, F_1),\]

\[(v_2, in_2, h_2, out_2, h_3, F_2),\]

...
Example Rules: 

*Regular Forwarding* on Top-Most Label

**Push:**

\[(v, \text{in})\ell \rightarrow (v, \text{out}, 0)\ell'\ell \text{ if } \tau_v(\text{in}, \ell) = (\text{out}, \text{push}(\ell'))\]

**Swap:**

\[(v, \text{in})\ell \rightarrow (v, \text{out}, 0)\ell'\ell \text{ if } \tau_v(\text{in}, \ell) = (\text{out}, \text{swap}(\ell'))\]

**Pop:**

\[(v, \text{in})\ell \rightarrow (v, \text{out}, 0) \text{ if } \tau_v(\text{in}, \ell) = (\text{out}, \text{pop})\]
Example *Failover* Rules

**Failover-Push:**

\[(v, \text{out}, i)\ell \rightarrow (v, \text{out}', i + 1)\ell'\ell \text{ for every } i, \ 0 \leq i < k, \]

where \(\pi_v(\text{out}, \ell) = (\text{out}', \text{push}(\ell'))\)

**Failover-Swap:**

\[(v, \text{out}, i)\ell \rightarrow (v, \text{out}', i + 1)\ell' \text{ for every } i, \ 0 \leq i < k, \]

where \(\pi_v(\text{out}, \ell) = (\text{out}', \text{swap}(\ell'))\),

**Failover-Pop:**

\[(v, \text{out}, i)\ell \rightarrow (v, \text{out}', i + 1) \text{ for every } i, \ 0 \leq i < k, \]

where \(\pi_v(\text{out}, \ell) = (\text{out}', \text{pop})\).

**Example rewriting sequence:**

\[(v_1, \text{in}_1)h_1 \perp \rightarrow (v_1, \text{out}, 0)h_\perp \rightarrow (v_1, \text{out}', 1)h'_\perp \rightarrow (v_1, \text{out}'', 2)h''_\perp \rightarrow \ldots \rightarrow (v_1, \text{out}_1, i)h_{2} \perp\]
A Complex and Big Formal Language! Why Polynomial Time?!

• Arbitrary number $k$ of failures: How can I avoid checking all $\binom{n}{k}$ many options?!

• Even if we reduce to push-down automaton: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even undecidable!
A Complex and Big Formal Language! Why Polynomial Time?!

- Arbitrary number $k$ of failures: How can I avoid checking all $\binom{n}{k}$ many options?!

- Even if we reduce to push-down automaton: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even undecidable!

This is not how we will use the PDA!
A Complex and Big Formal Language!  
Why Polynomial Time?!

- Arbitrary number $k$ of failures: How can I avoid checking all $\binom{n}{k}$ many options?!

- Even if we reduce to push-down automaton: simple operations such as emptiness testing or intersection on Push-Down Automata (PDA) is computationally non-trivial and sometimes even undecidable!

The words in our language are sequences of pushdown stack symbols, not the labels of transitions.
Time for Automata Theory (from Switzerland)!

- Classic result by Büchi 1964: the set of all reachable configurations of a pushdown automaton \( a \) is regular set

- Hence, we can operate only on Nondeterministic Finite Automata (NFAs) when reasoning about the pushdown automata

- The resulting regular operations are all polynomial time
  - Important result of model checking

Julius Richard Büchi
1924-1984
Swiss logician
**Part 1:** Parses query and constructs Push-Down System (PDS)

- In Python 3

**Part 2:** Reachability analysis of constructed PDS

- Using *Moped* tool

Regular query language

\[ k \text{ <a> b <c>} \]
Example: Traversal Testing With 2 Failures

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

Query: \( k=2 \) \([\cdot]\) s1 \(\Rightarrow\) s5 \(\Rightarrow\) s7 \([\cdot]\)

YES!

(Gives witness!)
Formal methods are nice (give guarantees!)... But what about ML...?!
Speed Up Further and Synthesize: Deep Learning (s. talk by Fabien Geyer)

- Yes sometimes **without losing guarantees**
- Extend **graph-based neural networks**
- **Predict** counter-examples and **fixes**

Network topologies and MPLS rules

Network topologies and query
Roadmap

• Opportunities of self-* networks
  – Example 1: Demand-aware, self-adjusting networks
  – Example 2: Self-repairing networks

• Challenges of designing self-* networks
Challenge 1: Hard Problems

- Optimization problems are often **NP-hard**: hard *even for computers*!

Waypoint routing: **disjoint paths**

Embedding: **Minimum Lin. Arrangement**

Topology design: **Graph spanners**
It can get worse...: intractable!

(Simplified) MPLS rules:

prefix rewriting

\[ \text{in} \times L \rightarrow \text{out} \times \text{OP} \]

where \( \text{OP} = \{\text{swap, push, pop}\} \)

Rules of general networks (e.g., SDN):

arbitrary header rewriting

\[ \text{in} \times L^* \rightarrow \text{out} \times L^* \]
It can get worse...: intractable!

(Simplified) MPLS rules:

prefix rewrite: \( in \times L \rightarrow out \times \text{OP} \)

where \( \text{OP} = \{\text{swap}, \text{push}, \text{pop}\} \)

Rules of general networks (e.g., SDN):

arbitrary header rewriting:

\( in \times L^* \rightarrow out \times L^* \)

Rules match the header \( h \) of packets arriving at \( in \), and define to which port \( out \) to forward as well as new header \( h' \).

Polynomial time

Undecidable!
It can get worse...: intractable!

Rules match the header $h$ of packets arriving at $in$, and define to which port $out$ to forward as well as new header $h'$.

What is a good tradeoff between generality and performance?

(Simplified) MPLS rules:
- Prefix rewriting in $x \rightarrow out x \cdot L$
  - where $OP = \{swap, push, pop\}$
Challenge 2: Realizing Limits?

- Can a self-* network realize its limits?

- E.g., when quality of input data is not good enough?

- When to hand over to human? Or fall back to „safe/oblivious mode“?

- Can we learn from self-driving cars?
Challenge 3: Self-Stabilization

- Could be an attractive property of self-* network!

A **self-stabilizing** system guarantees that it *reconverges to a desirable configuration* or state, *from any initial state.*
Self-Stabilization

Self-stabilizing algorithms pioneered by Dijkstra (1973): for example self-stabilizing mutual exclusion.

“I regard this as Dijkstra’s most brilliant work. Self-stabilization is a very important concept in fault tolerance.”
Leslie Lamport (PODC 1983)

Some notable works by Perlman toward self-stabilizing Internet, e.g., self-stabilizing spanning trees.

Yet, many protocols in the Internet are not self-stabilizing. Much need for future work.
E.g., Self-Stabilizing SDN Control?

- Distributed SDN control plane which self-organizes management of switches?
- Especially challenging: inband control (how to distinguish traffic?)
Challenge 4: Uncertainties

• How to deal with uncertainties?

• How to maintain flexibilities?

• Use of principles from robotics? E.g., empowerment?
Conclusion

• **Flexibilities** in networks: great opportunities for **optimization** and automation

• **Demand-aware** and **self-adjusting** networks: beating the routing lower bounds of oblivious networks, **reaching entropy bounds**

• Potential of **self-repairing** networks, self-stabilizing networks, etc.

• Much work ahead: **tradeoff** generality vs efficiency? How to self-monitor and **fall-back** if needed? Use of **formal methods** and ML?
Reachability?

Thank you! Questions?
Flexibilities and Complexity

On The Impact of the Network Hypervisor on Virtual Network Performance
Andreas Blenk, Arsany Basta, Wolfgang Kellerer, and Stefan Schmid.
IFIP Networking, Warsaw, Poland, May 2019.

Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges (Invited Paper)
Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid.

Efficient Distributed Workload (Re-)Embedding
Monika Henzinger, Stefan Neumann, and Stefan Schmid.
ACM/IFIP SIGMETRICS/PERFORMANCE, Phoenix, Arizona, USA, June 201

Parametrized Complexity of Virtual Network Embeddings: Dynamic & Linear Programming Approximations
Matthias Rost, Elias Döhne, and Stefan Schmid.
Charting the Complexity Landscape of Virtual Network Embeddings (Best Paper Award)
Matthias Rost and Stefan Schmid.
IFIP Networking, Zurich, Switzerland, May 2018.

Tomographic Node Placement Strategies and the Impact of the Routing Model
Yvonne Anne Pignolet, Stefan Schmid, and Gilles Tredan.
ACM SIGMETRICS, Irvine, California, USA, June 2018. hmid.
Demand-Aware and Self-Adjusting Networks

Survey of Reconfigurable Data Center Networks: Enablers, Algorithms, Complexity
Klaus-Tycho Foerster and Stefan Schmid.
Toward Demand-Aware Networking: A Theory for Self-Adjusting Networks (Editorial)
Chen Avin and Stefan Schmid.
Demand-Aware Network Design with Minimal Congestion and Route Lengths
Chen Avin, Kaushik Mondal, and Stefan Schmid.
Documents: paper pdf, bibtex bib
Distributed Self-Adjusting Tree Networks
Bruna Peres, Otavio Augusto de Oliveira Souza, Olga Goussevskaia, Chen Avin, and Stefan Schmid.
Efficient Non-Segregated Routing for Reconfigurable Demand-Aware Networks
Thomas Fenz, Klaus-Tycho Foerster, Stefan Schmid, and Anais Villedieu.
IFIP Networking, Warsaw, Poland, May 2019.
DaRTree: Deadline-Aware Multicast Transfers in Reconfigurable Wide-Area Networks
Long Luo, Klaus-Tycho Foerster, Stefan Schmid, and Hongfang Yu.
Demand-Aware Network Designs of Bounded Degree
Chen Avin, Kaushik Mondal, and Stefan Schmid.
31st International Symposium on Distributed Computing (DISC), Vienna, Austria, October 2017.
SplayNet: Towards Locally Self-Adjusting Networks
Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker.
Characterizing the Algorithmic Complexity of Reconfigurable Data Center Architectures
Klaus-Tycho Foerster, Monia Ghobadi, and Stefan Schmid.
Further Reading

Self-Repairing Networks

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

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

**Renaissance: A Self-Stabilizing Distributed SDN Control Plane**
Marco Canini, Iosif Salem, Liron Schiff, Elad Michael Schiller, and Stefan Schmid.
38th IEEE International Conference on Distributed Computing Systems (**ICDCS**), Vienna, Austria, July 2018.

**Empowering Self-Driving Networks**
Patrick Kalmbach, Johannes Zerwas, Peter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid.
ACM SIGCOMM 2018 Workshop on Self-Driving Networks (**SDN**), Budapest, Hungary, August 2018.

**DeepMPLS: Fast Analysis of MPLS Configurations using Deep Learning**
Fabien Geyer and Stefan Schmid.
**IFIP Networking**, Warsaw, Poland, May 2019.