The Quest for Static Resilient Forwarding Tables

MARCO CHIESA
UNIVERSITY OF LOUVAIN, BELGIUM

JOINT WORK WITH:

I. NIKOLAEVSKIY (AALTO),
S. MITROVIC (EPFL),
A. GURTOV (HIIT),
A. MADRY (MIT),
A. PANDA (UC BERKELEY),
M. SCHAPIRA (HUJI),
S. SHENKER (ICSI, UC BERKELEY)
The Quest for Static Resilient Forwarding Tables

How many failures can be tolerated without recomputing the forwarding tables?
How many failures can be tolerated without recomputing the forwarding tables?

The Quest for Static Resilient Forwarding Tables

correct delivery of packets, ie, no blackholes, no forwarding loops
How many failures can be tolerated without recomputing the forwarding tables?
How many failures can be tolerated without recomputing the forwarding tables?

recomputing takes time!
meanwhile packets are dropped
Dropping packets is **unacceptable** in **real-time** applications.

Today’s requirements:

- high reliability
- low latency
Precomputing the forwarding state
Precomputing the forwarding state is hard

Failures are **unpredictable** events

- mean-time between failures
- fiber cuts
Precomputing the forwarding state is hard

Failures are *unpredictable* events

- mean-time between failures
- fiber cuts
- … shark attacks!

**Challenge**

Precompute backup forwarding paths *for all the possible sets* of link failures
Forwarding loop example
Forwarding loop example
Forwarding loop example
Forwarding loop example
What information is **locally** available in a switch for handling a packet?
What information is **locally** available in a switch? The **forwarding table** match -\(\rightarrow\) action
What information is **locally** available in a switch? The **header** of the received packet
What information is **locally** available in a switch? The **incoming port** of the received packet.
What information is *locally* available in a switch? The outgoing port depends on the *failed links*.
State of the art

Multiple link failures:

- packet header rewriting
  [Lakshminarayanan07, Elhourani14, Borokovich13-14, Stephens16]

- packet duplication
  [Vulimiri13]
Limitations of traditional approaches

Multiple link failures:

- packet header rewriting
  [Lakshminarayanan07, Elhourani14, Borokovich13-14, Stephens16]
  packet header and forwarding tables overhead

- packet duplication
  [Vulimiri13]
  bandwidth overhead
What is the robustness of even the simplest possible static forwarding tables?
We consider three forwarding table models

**BASIC** routing

- Match: destination id + incoming port
- Action: forwarding on a single outgoing port

**HEADER** rewriting

- Match: BASIC + packet header
- Action: BASIC + rewriting of the packet header

**DUPPLICATION** routing

- Match: BASIC
- Action: forwarding on multiple outgoing ports
How many link failures can be tolerated without recomputing the forwarding tables?
The answer clearly depends on the **structural** properties of the network.
The connectivity of a network $N$ : the minimum number of link deletions that partitions $N$
The **connectivity** of this network is four.
Natural design goal for resilient forwarding tables

A $k$-connected network cannot be partitioned by $k - 1$ link deletions

Packets should not be dropped if $k - 1$ link fails
A $k$-connected network cannot be partitioned by $k - 1$ link deletions.

Packets should not be dropped if $k - 1$ link fails.

*Definition*: The forwarding tables are *connectivity-resilient*.
## Summary of our results

<table>
<thead>
<tr>
<th>Model</th>
<th>Connectivity-resilient</th>
<th>Header overhead</th>
<th>Path stretch</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>yes, up to 5-connected</td>
<td>none</td>
<td>low</td>
</tr>
<tr>
<td>HEADER</td>
<td>yes</td>
<td>3 bits</td>
<td>high</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log(k) bits</td>
<td>low</td>
</tr>
<tr>
<td>DUPLICATION limited bandwidth</td>
<td>yes</td>
<td>none</td>
<td>high</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log(k) bits</td>
<td>very low</td>
</tr>
</tbody>
</table>
A k-connected network contains k arc-disjoint spanning trees similarly to [Elhourani-INFOCOM14]
The first spanning tree
The second spanning tree
The third spanning tree
The fourth spanning tree
Each link is contained in at most two trees, but in opposite directions
General technique: routing along the same tree
General technique: routing along the same tree
When a failed link is hit …
... how do we choose the next tree?
We consider two simple, yet powerful, alternatives:

- **Circular-tree routing**: switch to the next tree based on an ordering of the trees.
  - This technique is \((k/2 - 1)\)-resilient

... how do we choose the next tree?
... how do we choose the next tree?

We consider two simple, yet powerful, alternatives:

- **Circular-tree routing**: switch to the next tree based on an ordering of the trees.
  - This technique is \((k/2 - 1)\)-resilient

- **Bounce-tree routing**: reroute on the tree that shares the failed link
We consider two simple, yet powerful, alternatives:

- **Circular-tree routing**: switch to the next tree based on an ordering of the trees.
  - This technique is $\left(\frac{k}{2} - 1\right)$-resilient

- **Bounce-tree routing**: reroute on the tree that shares the failed link
We consider two simple, yet powerful, alternatives:

- **Circular-tree routing**: switch to the next tree based on an ordering of the trees.
  - This technique is \((k/2 - 1)\)-resilient

- **Bounce-tree routing**: reroute on the tree that shares the failed link

... how do we choose the next tree?
... how do we choose the next tree?

We consider two simple, yet powerful, alternatives:

- **Circular-tree routing**: switch to the next tree based on an ordering of the trees.
  - This technique is \( (k/2 - 1) \)-resilient

- **Bounce-tree routing**: reroute on the tree that shares the failed link
We consider two simple, yet powerful, alternatives:

- **Circular-tree routing**: switch to the next tree based on an ordering of the trees.
  - This technique is \((k/2 - 1)\)-resilient

- **Bounce-tree routing**: reroute on the tree that shares the failed link
  - very easy to create a forwarding loop

... how do we choose the next tree?
We consider two simple, yet powerful, alternatives:

- **Circular-tree routing**: switch to the next tree based on an ordering of the trees.
  - This technique is \((k/2 - 1)\)-resilient

- **Bounce-tree routing**: reroute on the tree that shares the failed link.
  - very easy to create a forwarding loop
Our key property

There always exists a tree $T$ such that a packet either
- reaches the destination through $T$ or
- bounces on a failed link and it reaches the destination
Our key property

There always exists a tree $T$ such that a packet either

- reaches the destination through $T$ or
- bounces on a failed link and it reaches the destination
Our key property

There always exists a tree $T$ such that a packet either
- reaches the destination through $T$ or
- bounces on a failed link and it reaches the destination

Packet-delivery sufficient condition:
- route on tree 1 + bounce
- route on tree 2 + bounce
- ...
- route on tree $k$ + bounce
Bouncing packets without marking any bits in the header is dangerous!

Tree ids 1 2 3 4

circular-tree routing
Bouncing packets without marking any bits in the header is dangerous!

Tree ids:
1 2 3 4

circular-tree routing

bounce-tree routing
## BASIC

<table>
<thead>
<tr>
<th>Model</th>
<th>Connectivity-resilient</th>
<th>Header overhead</th>
<th>Path stretch</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>yes, up to 5-connected</td>
<td>none</td>
<td>low</td>
</tr>
<tr>
<td>HEADER</td>
<td>yes</td>
<td>3 bits</td>
<td>high</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log(k) bits</td>
<td>low</td>
</tr>
<tr>
<td>DUPLICATION limited</td>
<td>yes</td>
<td>none</td>
<td>medium</td>
</tr>
<tr>
<td>bandwidth</td>
<td></td>
<td>log(k) bits</td>
<td>very low</td>
</tr>
</tbody>
</table>
BASIC routing 4-resiliency

Based only on circular-tree routing

- bounced-tree routing is risky
- we construct specific trees such that 4-resiliency is guaranteed

More details in the paper
# Header with $\log(k)$ bits

<table>
<thead>
<tr>
<th>Model</th>
<th>Connectivity-resilient</th>
<th>Header overhead</th>
<th>Path stretch</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>yes, up to 5-connected</td>
<td>none</td>
<td>low</td>
</tr>
<tr>
<td>HEADER</td>
<td>yes</td>
<td>3 bits</td>
<td>high</td>
</tr>
<tr>
<td>DUPLICATION limited bandwidth</td>
<td>yes</td>
<td>none</td>
<td>medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\log(k)$ bits</td>
<td>very low</td>
</tr>
</tbody>
</table>
# Header with $\log(k)$ bits

<table>
<thead>
<tr>
<th>Model</th>
<th>Connectivity-resilient</th>
<th>Header overhead</th>
<th>Path stretch</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>yes, up to 5-connected</td>
<td>none</td>
<td>low</td>
</tr>
<tr>
<td>HEADER</td>
<td>yes</td>
<td>3 bits</td>
<td>high</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\log(k)$ bits</td>
<td>low</td>
</tr>
<tr>
<td>DUPLICATION</td>
<td>yes</td>
<td>none</td>
<td>medium</td>
</tr>
<tr>
<td>limited</td>
<td></td>
<td>$\log(k)$ bits</td>
<td>very low</td>
</tr>
<tr>
<td>bandwidth</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
HEADER with $\log(k)$ bits
store in the header the circular-tree information
HEADER with $\log(k)$ bits
...start routing from the red tree
HEADER with $\log(k)$ bits
…until hitting a failed link
HEADER with $\log(k)$ bits
...bounce on the opposite tree

Tree ids
1 2 3 4
HEADER with $\log(k)$ bits
...until hitting a failed link
HEADER with \( \log(k) \) bits
...route on the next tree after red
HEADER with $\log(k)$ bits
...a failed edge, bounce
HEADER with $\log(k)$ bits
...route on red until hitting a failed link.
HEADER with $\log(k)$ bits

...route on the next tree after blue
HEADER with $\log(k)$ bits
...packet delivered!
## Duplication limited bandwidth

<table>
<thead>
<tr>
<th>Model</th>
<th>Connectivity-resilient</th>
<th>Header overhead</th>
<th>Path stretch</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>yes, up to 5-connected</td>
<td>none</td>
<td>low</td>
</tr>
<tr>
<td>HEADER</td>
<td>yes</td>
<td>3 bits</td>
<td>high</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log(k) bits</td>
<td>low</td>
</tr>
<tr>
<td>DUPLICATION limited bandwidth</td>
<td>yes</td>
<td>none</td>
<td>medium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log(k) bits</td>
<td>very low</td>
</tr>
</tbody>
</table>
There always exists a tree $T$ such that a packet either
- reaches the destination through $T$ or
- bounces on a failed link and it reaches the destination

Packet-delivery sufficient condition:
- route on tree 1 + bounce
- route on tree 2 + bounce
- …
- route on tree $k$ + bounce
When hitting a failed link, create two copies of a packet

1. send the 1\textsuperscript{st} copy along the next tree (circular-tree routing)

2. send the 2\textsuperscript{nd} copy through the bounced tree (bounced-tree routing, if exists)

One copy of a packet is guaranteed to reach the destination, but… the others may loop
DUPLICATION routing
...start from the red tree

Tree ids
1 2 3 4
DUPLICATION routing

...switch to the next tree, no bouncing
DUPLICATION routing

...switch to the next tree, no bouncing
DUPLICATION routing
…route through the yellow tree
DUPLICATION routing
...a failed link is hit
DUPLICATION routing
…create two copies of the packet
DUPLICATION routing
...one is routed through the green tree
DUPLICATION routing

…one is routed through the green tree
DUPLICATION routing
...one is routed through the green tree
DUPLICATION routing
...destination reached!
DUPLICATION routing
…the other one is bounced
DUPLICATION routing
…the same failed link is hit
DUPLICATION routing
…reroute though the next tree (again!)
DUPLICATION routing

...a failed link is hit (again)
DUPLICATION routing
…create two copies of the packet (again)
Bouncing packets without marking any bits in the header is dangerous!

Tree ids

1 2 3 4

- circular-tree routing
- bounce-tree routing
Partition the trees into two sets s.t. trees in the same partition do not share links

- our contribution: we show how to construct such trees!

DUPLICATION routing correct algorithm

\[ T_A \]

- \[ 1 \]
- \[ 2 \]

\[ T_B \]

- \[ 3 \]
- \[ 4 \]
Partitioning arc-disjoint trees

\[ T_A \quad T_B \]

1 2 3 4
RED does not share links with YELLOW
BLUE does not share links with GREEN
Partition the trees into two sets s.t. trees in the same partition do not share links

- our contribution: we show how to construct such trees!

\[ T_A \]

\[
\begin{array}{cc}
1 & 2 \\
3 & 4
\end{array}
\]

\[ T_B \]

Packet-delivery loop-free sufficient condition:

- route on tree 1 + bounce
- ... 
- route on tree k/2 + bounce

- route on tree k/2+1 (+drop)
- ... 
- route on tree k (+drop)
Partition the trees into two sets s.t. trees in the same partition do not share links

- our contribution: we show how to construct such trees!

Packet-delivery loop-free sufficient condition:
- route on tree 1 + bounce
- …
- route on tree \( k/2 \) + bounce
- route on tree \( k/2+1 \) (+drop)
- …
- route on tree \( k \) (+drop)
4-connected graphs with 3 link failures

Measure: path stretch w.r.t. the k-bits algorithm [Elhourani-INFOCOM14]
8-connected graphs with 4 and 7 link failures

Measure: path stretch w.r.t. the k-bits algorithm [Elhourani-INFOCOM14]
Conclusions

A better understanding of the resiliency of static forwarding tables
- impossibility results in the paper

Simple algorithms for achieving high-resiliency in different models
- easy to implement

Practical good path stretches (compared to state of the art resilient routing algorithms)