NetBouncer: Active Device and Link Failure Localization in Data Center Networks

Presented by

Akash Kulkarni

Problems that may occur in Data Center

- Routing misconfigurations
- Network device hardware failures
- Network device software bugs
- Gray Failures (subtle or partial malfunctions):
  - Drop packets probabilistically (can not be detected by evaluating connectivity)

Problems in Traditional Failure Localization System

1. Traditional Systems which query switches for packet loss are unable to observe gray failures.
2. Previous Systems need special hardware support, for eg, tweaking standard bits on network packets – making it unable to be readily deployed.
3. Some prior systems can only pinpoint a region which has the failures. Extra efforts to discover actual error.

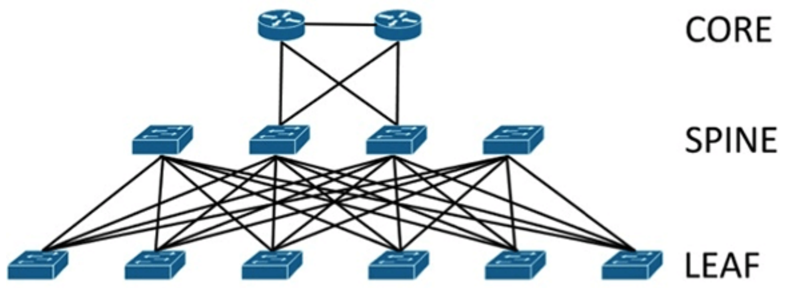
Failure Localization System must satisfy three requirements

1. Failure localization system needs an end-host’s perspective.
2. Should be readily deployable in practice – compatible with hardware, existing software stack and networking protocols.
3. Localizing failures should be precise and accurate (pinpointing towards link or device failures). Should incur less false positives and false negatives.

NetBouncer introduces:

- Efficient and compatible path probing method
- A probing plan to distinguish device failures
- A link failure inference algorithm

Clos network

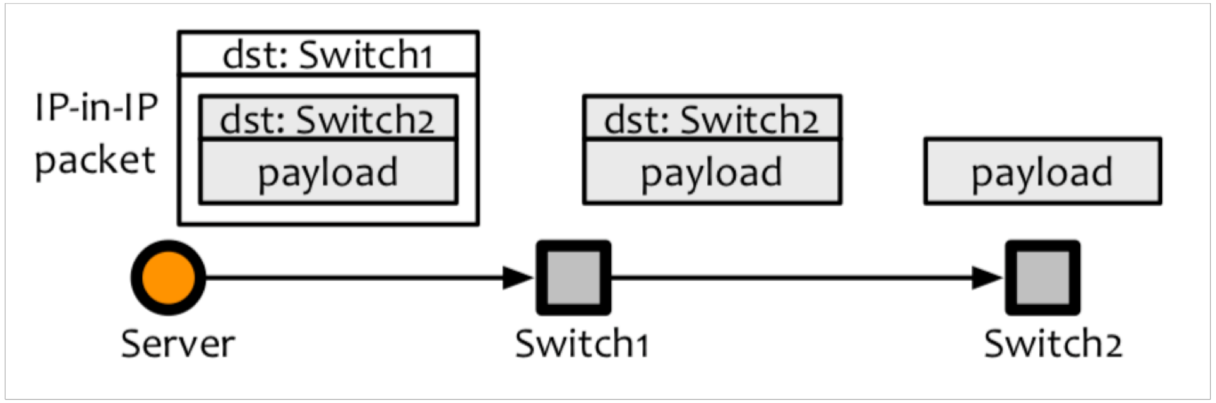


Probing Plan

- Probing scheme should satisfy two requirements:
  1. Pinpoint the routing path of probing packets
  2. Consume less network resources – such as bandwidth.

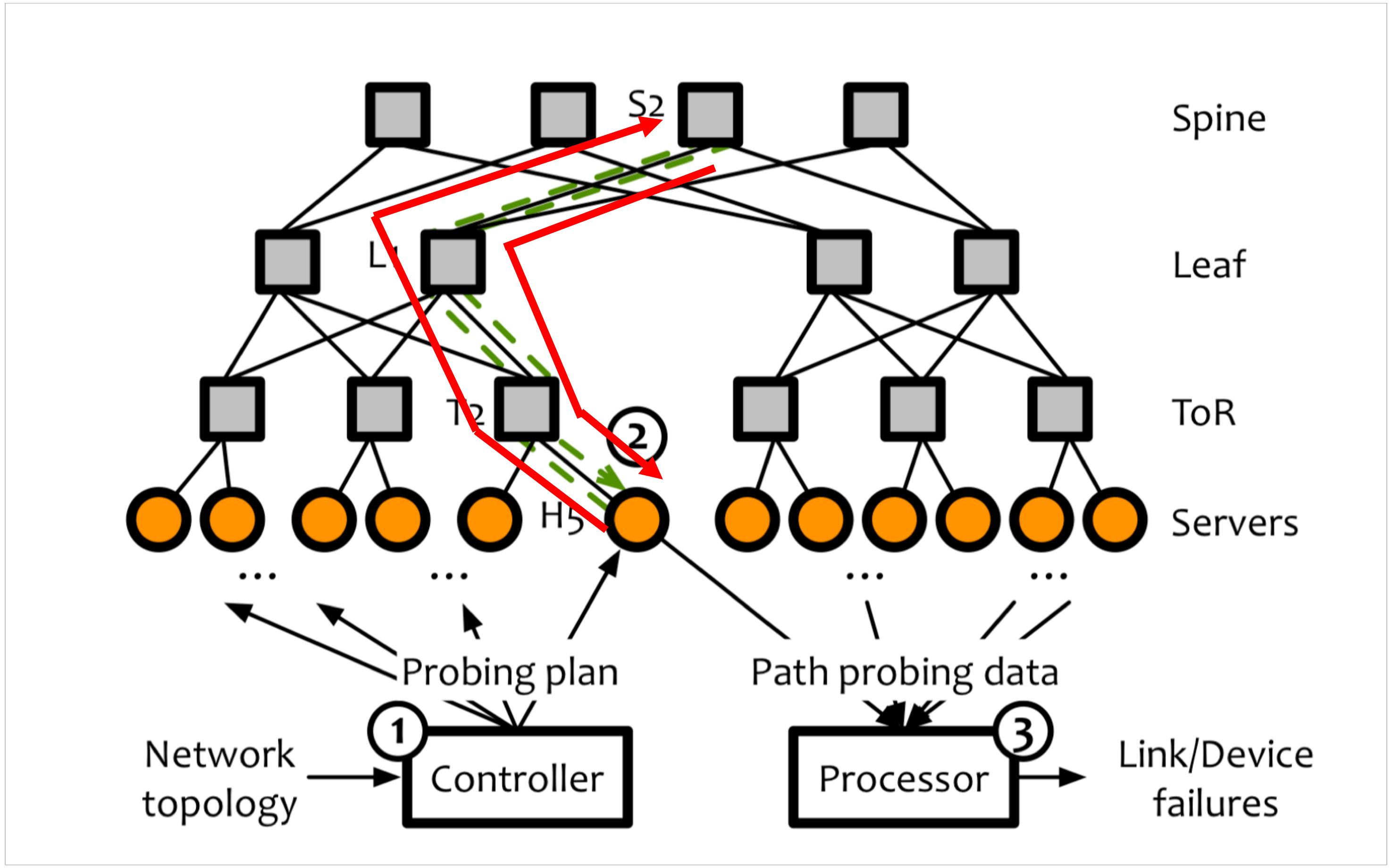
NetBouncer's Path Probing via Packet Bouncing

- IP-in-IP protocol



- Because the target network is Clos Network:
  1. Minimizes number of IP-in-IP headers (because less and smart connections)
  2. Links are evaluated bidirectionally – allowing the graph to be undirected.
  3. Sender and receiver are on the same server – less complicated.

NetBouncer workflow



Mathematical Notations

- Each link has a success probability, denoted by x_i for the i^{th} link.
- Path success probability of j^{th} path , denoted by y_j, described as

$$y_j = \prod_{i:\ \text{link}_i \in \text{path}_j} x_i, \forall j,$$

- Data inconsistency
  - Imperfect measurements
  - Accidental packet loss
- Latent factor model

$$\begin{aligned} \text{minimize} \quad & \sum_j \Big(y_j - \prod_{i:\text{link}_i \in \text{path}_j} x_i\Big)^2 + \boxed{\lambda \sum_i x_i(1-x_i)} \\ \text{subject to} \quad & 0 \leq x_i \leq 1, \forall i \end{aligned}$$

Algorithm running on NetBouncer's Processor

Define:

devs : all devices

$Y : path \rightarrow [0,1]$ // a map from a path to its success probability

1: **procedure** PROCESSOR( )
2: (1) Collect probing data from agents as Y
3: (2) $badDev \leftarrow$ DETECTBADDEVICES(Y) // line 9
4: // eliminate the unsolvable subgraph
5: (3) $Y \leftarrow Y \setminus \{\text{path}_r \mid \text{path}_r \text{ passes any device in } badDev\}$
6: (4) $badLink \leftarrow$ DETECTBADLINKS(Y) // in Figure 5, §5.2
7: **return** $badDev, badLink$
8:
9: **procedure** DETECTBADDEVICES(Y)
10: $badDev \leftarrow \{\}$
11: **for** dev_p in $devs$:
12: $goodPath \leftarrow$ False
13: **for** all path_q passes dev_p :
14: **If** $Y[\text{path}_q] = 1$ **then** $goodPath \leftarrow$ True; break
15: **If** not $goodPath$ **then** $badDev += \text{dev}_p$
16: **return** $badDev$

Algorithm running on NetBouncer's Processor

Define:

$X \leftarrow all\ x_i, \quad Y \leftarrow all\ y_i$

$$f(X,Y) \leftarrow \sum_j (y_j - \prod_{i:\text{link}_i \in \text{path}_j} x_i)^2 + \lambda \sum_i x_i(1-x_i)$$

1: **procedure** DetectBadLinks(Y)

2: $X \leftarrow$ InitLinkProbability(Y) // line 12

3: $L_0 \leftarrow f(X,Y)$ // initial value for target function f

4: **for** iteration $k = 1, \cdots,$ MaxLoop :

5: **for** each x_i in X :

6: $x_i \leftarrow \underset{x_i}{argmin}\ f(X,Y)$

7: project x_i to $[0,1]$

8: $L_k \leftarrow f(X,Y)$

9: **If** $L_{k-1} - L_k < \varepsilon$ **then** break the loop

10: **return** $\{(i, x_i) | x_i \leq bad\ link\ threshold\}$

11:

12: **procedure** InitLinkProbability(Y)

13: $X \leftarrow \{\}$

14: **for** link_i in $links$:

15: // initialize link success probability

16: $x_i \leftarrow avg(\{y_j \mid \text{link}_i \in \text{path}_j\})$

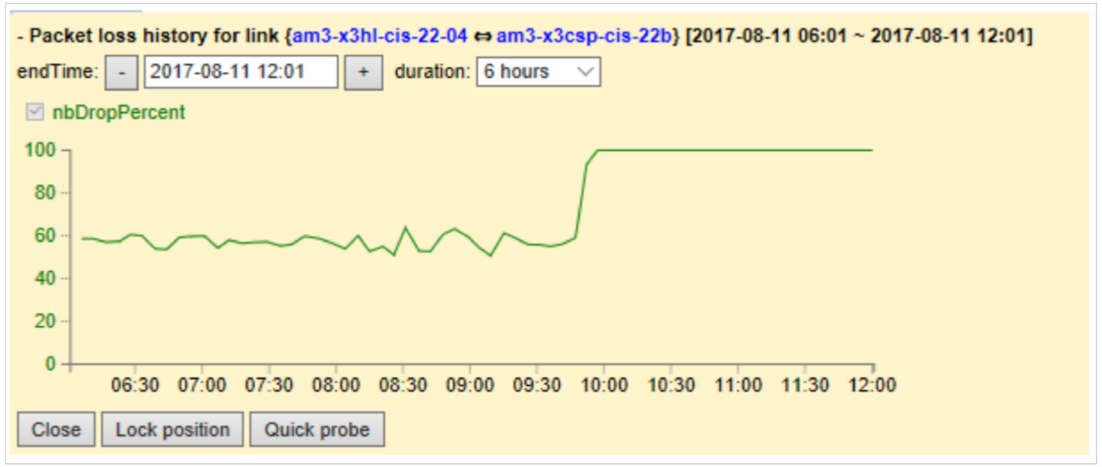
17: **return** X

Implementation

- Controller:
  - Takes network topology as input and generates probing plan.
  - Plan contains number of packets to send, packet size, UDP source destination port, probe frequency, TTL etc
- Agent:
  - Fetches probing plan from Controller which contains the paths to be probed.
  - Generates record containing path, packet length, total number of packets sent, number of packet drops, RTTs etc.
  - CPU and traffic delays are negligible because of IP-in-IP technique.

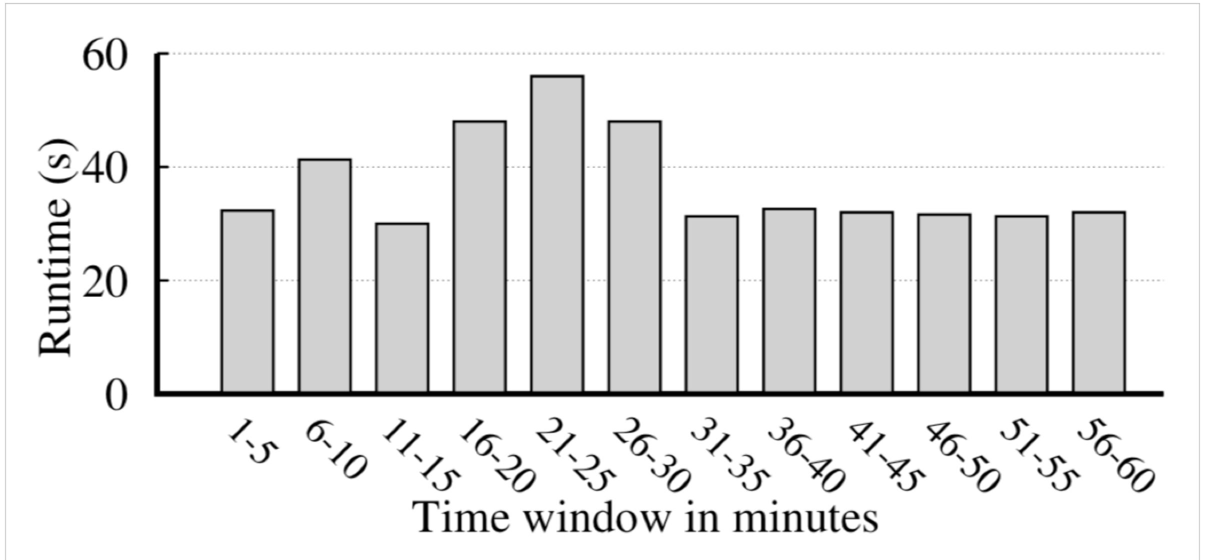
Implementation

- Processor:
  - Front End: collects records from agent.
  - Back End: runs detection algorithm.
- Result verification and visualization tool:
  - Shows packet drop history of detected links for visualization.



Observations

- NetBouncer’s probing plan achieves the same performance as hop-by-hop probing plan while it remarkably reduces the number of paths to be probed.
- Time to detection for failures < 60 seconds.



Observations

Table 1: Variance of NetBouncer with setup

Faulty link%	Hop-by-hop #FN	Hop-by-hop #FP	w/o DFD #FN	w/o DFD #FP	Convex #FN	Convex #FP	L_1 (λ=0.5) #FN	L_1 (λ=0.5) #FP	L_1 (λ=0.5) Err	L_1 (λ=1) #FN	L_1 (λ=1) #FP	L_1 (λ=1) Err	L_1 (λ=2) #FN	L_1 (λ=2) #FP	L_1 (λ=2) Err	NetBouncer (λ=1) #FN	NetBouncer (λ=1) #FP	NetBouncer (λ=1) Err
0.1%	0	0	135.3	0	0	46.9	0	48.5	0.01	0	0	0.03	0.3	0	0.14	0	0	0.01
1%	0	0	164.0	0	1.9	522.7	0	81.1	0.07	0	0	0.32	1.1	0	1.41	0	0	0.11
10%	0.6	0	123.3	0	257.6	4.1k	0	695.7	0.91	0.1	0.6	3.80	25.6	0	15.88	0.3	0.2	1.43

Table 2: Comparison of CD and SGD

OptMethod	Learning rate	#round	Time(s)
CD	–	4	14.8
SGD-lazy	0.001	145	513.3
SGD-lazy	0.005	45	157.5
SGD-lazy	0.01	161	569.9

Table 3: Comparison of NetBouncer with existing schemes

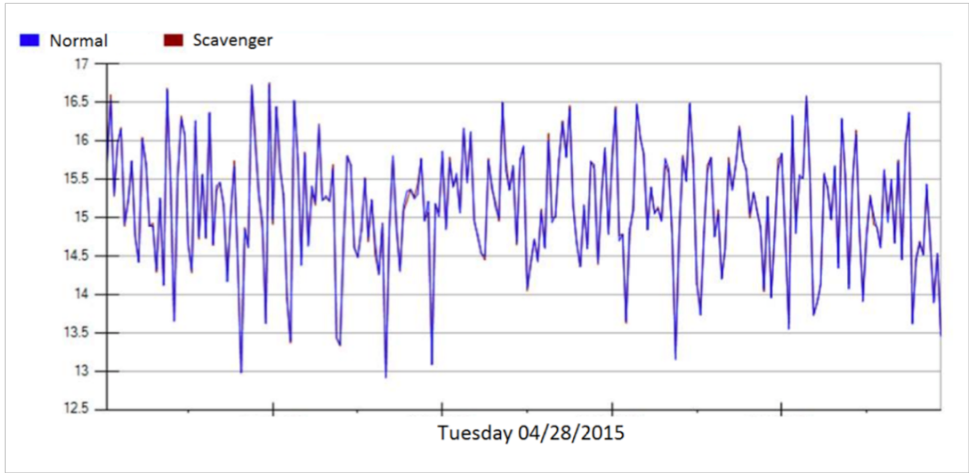
Faulty link%	0.1% #FN	0.1% #FP	1% #FN	1% #FP	10% #FN	10% #FP
NetBouncer	0	0	0	0	0.3	0.2
deTector (0.6)	187.5	6.0k	215.5	7.2k	204.0	22.8k
deTector (0.9)	204.5	0.7	191.5	0.4	208.0	21.7
NetScope (0.1)	0	9.1k	3.0	10.8k	167.5	12.6k
NetScope (1)	0.3	43.7	10.2	395.5	319.5	3.8k
NetScope (10)	28.7	6.3	291.5	86.7	2.4k	1.2k
KDD14	7.8	21.0	76.6	433.2	213.8	3.0k

Deployment experiences

- Clear improvements:
  1. Reduces detection time of gray failures from hours to minutes
  2. Deepened understanding of the reasons why packer drops happen – silent packer drops, link congestion, link flapping, switch unplanned reboot, packet blackholes etc.

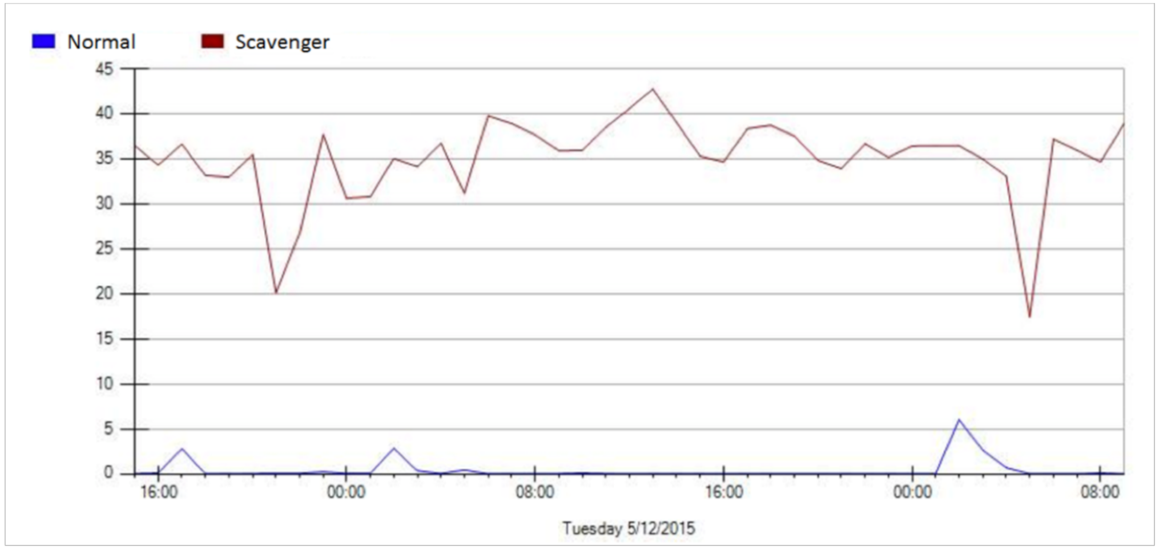
Deployment experience

- Case 1: Spine router gray failure
  - Switch silently dropping packets .
  - Led to packet drops and latency increases.
  - Traditional systems detected end-to-end latency issues.
  - Clear that one or more switches were dropping packets. But which one?
  - NetBouncer detected lossy links.



Deployment experience

- Case 2: Polarized traffic
  - Switch firmware bug – polarized traffic load onto a single link
  - NetBouncer observed that the Scavenger traffic was dropped at a probability of 35% - causing congestion.



Deployment experience

- Case 3: Miscounting TTL
  - Supposed to be decremented by one though each switch
  - NetBouncer detected that certain set of switches were decrementing by two.
  - Manifests as a “false positive” by misclassifying affected good links as bad.
  - Verified and visualized to realize it was false positive.
  - Further analysis of detected devices and links – internal switch firmware bug.

Deployment experiences – failed cases

- DHCP booting failure.
  - Servers could send DHCP DISCOVER packets but could not receive responding DHCP OFFER packets.
  - NetBouncer did not detect packet drops. However, the real problem was caused by NIC.
- Misconfigured switch ACL (ACL filters packet)
  - Packets drop for limited set of IP addresses.
  - NetBouncer scanned wide range of IP addresses – so signal detected was weak.
- Firewall rules – wrongly applied.

Limitations of NetBouncer

- Assumes probing packets experiences same failures as real applications.
- Does not guarantee zero false positives or negatives.
- Assumes failures are independent (might lead to wrong detection)
- Only detects persistent congestion (depends on the probing frequency)

NetBouncer - running in Microsoft Azure’s data centers for three years!

Thank You