Debugging Applications using PathDump and SwitchPointer API

Praveen Tammana
University of Edinburgh

Rachit Agarwal
Cornell University

Myungjin Lee
University of Edinburgh

1 Definitions

PathDump and SwitchPointer exposes a simple interface for network debugging; see Table 1. We assume that each switch and host has a unique ID. We use the following definitions:

- A `linkID` is a pair of adjacent `switchIDs` ($\langle S_i, S_j \rangle$);
- A `Path` is a list of `switchIDs` ($\langle S_i, S_j, \ldots \rangle$);
- A `flowID` is the usual 5-tuple ($\langle$srcIP, dstIP, srcPort, dstPort, protocol$\rangle$);
- A `Flow` is a ($\langle$flowID, Path$\rangle$) pair; this will be useful for cases when packets from the same `flowID` may traverse along multiple paths.
- An `epochID` represents an epoch in a switch.
- A `pathInfo` is a list of ($\langle$`linkID`, a list of `epochIDs`$\rangle$) pairs, where `linkID` represents a link that a specific flow traverse and the corresponding `epochIDs` at the first switch (S_i) in a `linkID`.
- A `flowStat` is a ($\langle$flowID, priority, byte counts$\rangle$) tuple.
- A `timeRange` is a pair of `timestamps` ($\langle t_i, t_j \rangle$);

PathDump and SwitchPointer support wildcard entries for `switchIDs` and `timestamps`. For instance, ($\langle \star, S_j \rangle$) is interpreted as all incoming links for switch S_j and ($\langle t_i, \star \rangle$) is interpreted as “since time t_i”.

2 Interface

Each host exposes the host API in Table 1 and returns results for “local” flows, that is, for flows that have this host as their `dstIP`. Similarly, each switch exposes switch API and returns the hostIDs seen in a particular epoch. To collect the results distributed across individual end-hosts and switches, the controller may use the controller API — to `execute` a query at end-hosts or switches, to `install` a query at end-hosts for periodic execution, or to `uninstall` a query at end-hosts. For instance, to debug a network event, a controller use the controller API to locate, retrieve, and correlate the necessary data.

```
getLinkEpochStats(linkID, epochID):
  SwitchID = linkID[0]  # first switch in the linkID
  linkEpochStats = []
  hostIDs = execute([switchID], Query = getHosts(epochID))
  linkEpochStats = execute(hostIDs, Query = getTelemetryData(linkID, epochID))
  return linkEpochStats


getContendingFlows(flowID, pathInfo):
  contendingFlows=[]
  for linkID, epochIDs in pathInfo:
    for eID in epochIDs:
      linkEpochStats = getLinkEpochStats(linkID, eID)
  contendingFlows.append((linkID, eID, linkEpochStats))
  return contendingFlows
```

Host API	Description
`getFlows(linkID, timeRange)`	Return list of `flows` that traverse `linkID` during specified `timeRange`.
`getPaths(flowID, linkID, timeRange)`	Return list of `Paths` that include `linkID`, and are traversed by `flowID` during specified `timeRange`.
`getPathInfo(flowID, timeRange)`	Return `pathInfo` of `flowID` in specified `timeRange`.
`getTelemetryData(linkID, epochID)`	Return list of `flowStats` traversing the `linkID` in an `epochID`.
`getCount(Flow, timeRange)`	Return packet and byte counts of a `flow` within a specified `timeRange`.
`getDuration(Flow, timeRange)`	Return the duration of a `flow` within a specified `timeRange`.
`getPoorTCPFlows(Threshold, timeRange)`	Return the `flowIDs` for which `protocol = TCP`, and the throughput in the current window is less than the threshold times previous window.
`getLossyFlows(Threshold)`	Return the `flowIDs` whose `protocol = TCP`, and the number of time-outs is greater than threshold times number of packets received so far.
`Alarm(flowID, Reason, pathInfo)`	Raise an alarm regarding `Flow` with a reason code (e.g., TCP performance alert (`POOR_PERF`)), and the `pathInfo` of `Flow`
Switch API	**Description**
`getHosts(epochID)`	Return list of `hostIDs` traverse a switch in the `epochID`.
`getHosts(timeRange)`	Return list of (⟨`epochID, hostIDs`⟩) pairs, where the `epochID` is in the `timeRange`.
Controller API	**Description**
`execute(List⟨HostID/SwitchID⟩,Query)`	Execute a `Query` once at each host or switch specified in list of `HostIDs` or `SwitchIDs`; a `Query` could be any of the ones from Host API or Switch API.
`install(List⟨HostID⟩,Query,Period)`	Install a `Query` at each host specified in list of `HostIDs` to be executed at regular `Periods`. If the `Period` is not set, the query execution is triggered by a new event (*e.g.*, receiving a packet).
`uninstall(List⟨HostID⟩,Query)`	Uninstall a `Query` from each host specified in list of `HostIDs`.
`getLinkEpochStats(linkID, epochID)`	Return list of `flowStats` traversing `linkID` in the `epochID`.
`getContendingFlows(flowID, pathInfo)`	Return list of `flowStats` contending with the `flowID` across all links in the `flowID` path.

Table 1: Interface.

3 Debugging applications

Applications	Description	Pseudo code

Priority based flow contention	A low priority flow contends with one or more high priority flows at multiple switches present in the low priority flow's path. Debugging the poor performance of low priority flow requires maintaining temporal state (that is, flowIDs and packet priorities for all flows that the low priority flow contends with) and spatial state (same as temporal state but now at each switch).
	The analyze function in pseudo code checks for at-least one epochID is common between high and low priority flows. If present, the participating flows could be re-directed to different paths to avoid contention.
		Endhost:
		`tRange = (t1, t2)`
		`flowIDs = getPoorTCPFlows(threshold, tRange)`
		`for fID in flowIDs:`
		`  pathInfo = getPathInfo(flowID, tRange)`
		`  Alarm(fID, POOR_PERF, pathInfo)`
		Controller:
		`contendingFlows = getContendingFlows(fID,`
		`                                pathInfo)`
		`analyze(contendingFlows)`

Traffic cascades	A middle priority flow and a low priority flow contend at a downstream switch, because the middle priority flow packets are delayed at an upstream switch by a high priority flow. In the absence of the high priority flow, there would be no contention at both upstream and downstream switches.
	The analyze function in the pseudo code checks for traffic cascades: first it looks for common epochIDs between middle and high priority flows at the upstream switch, and then between middle and low priority flows at the downstream switch.
		Controller:
		`fID = midPriorityFlowID`
		`pathInfo = midPriotiyFlowPathInfo`
		`contendingFlows = getContendingFlows(fID,`
		`                                pathInfo)`
		`analyze(contendingFlows)`

Transient congestion diagnosis [7]	Identify flows responsible for transient congestion at a switch. It allows to check if flows of multiple applications (*e.g.*, a heavy hitter and incast traffic) traverse the switch at the same time. Such an event might fill up switch buffer, cause for high job finish time of an application that involves incast traffic.	**Controller:**
		`linkID=(switchID,*), tRange=(t1, t2)`
		`flowStats=[]`
		`result = execute([switchID], getHosts(tRange))`
		`for epochID, hostIDs in result:`
		`  result = execute(hostIDs,`
		`      getTelemetryData(linkID, epochID))`
		`  flowStats.append(result)`
		`analyze(flowStats)`

ECMP load imbalance diagnosis [2]	Through cross-comparison of the flow size distributions on the egress ports, the operator can tell the degree of load imbalance.	**Controller:**
		`result = ; binsize = 10000`
		`linkIDs = (l1, l2); tRange = (t1, t2)`
		`for lID in linkIDs:`
		`  switchID = lID[0]`
		`  result = execute([switchID],`
		`                   getHosts(tRange))`
		`for epochID, hostIDs in result:`
		`  flowStats = execute(hostIDs,`
		`      getTelemetryData(linkID, epochID))`
		`  for fStat in flowStats:`
		`    result[lID][fStat.bytes/binsize] += 1`
		`return result`

Table 2: Debugging applications.

Applications	Description	Pseudo code

Number of active connections [8, 9]	Find the number of active flows converge on the same switch interface in a short time period. This information allows to detect incast [3] traffic at an egress port. Moreover, sum of flow bytes in a short period may suggests the queue utilization.	**Controller:**
		`linkID=(S1, S2), tRange=(t1, t2)`
		`flowIDs = [], bytes = 0,`
		`result = execute([S1], getHosts(tRange))`
		`for epochID, hostIDs in result:`
		`  result = execute(hostIDs,`
		`      getTelemetryData(linkID, epochID))`
		`  for fStat in result:`
		`    fStat.append(flowIDs)`
		`    bytes += fStat.bytes`

Silent random packet drop detection [13]	A faulty switch interface drops packets at random without updating the discarded packet counters at respective interfaces.	**Endhost:**
		`tRange = (t1, t2)`
		`flowIDs = getPoorTCPFlows(threshold, tRange)`
		`for fID in flowIDs:`
		`  pathInfo = getPaths(flowID, <*,*> tRange)`
		`  Alarm(fID, POOR_PERF, pathInfo)`
		Controller:
		`Runs localization algorithm such as MAX-COVERAGE on the received paths that potentially include faulty links.`

Silent blackhole diagnosis [13, 4]	PathDump greatly reduces the search space while debugging a network blackhole. In datacenters, to load balance the traffic along multiple paths, operators may enable ECMP forwarding or packet spraying in the network. PathDump exploits the multi-path forwarding, and pin-point to a small part of the network (that is, most likely problematic switches).	**Controller:**
		`tRange = (t1, t2), hostID = flowID.dIP`
		`allPaths = List(Paths that a flowID can traverse)`
		`goodPaths = execute([flowID.dIP], getPaths(flowID,`
		`                                 <*, *>, tRange)`
		`potentialCulprits =`
		`    linksIDs(allPaths)-linkIDs(goodPaths)`
		`for linkID in potentialCulprits:`
		`  switchID = linkID[0]`
		`  result = execute([switchID], getHosts(tRange))`
		`  if flowID.dstIP not in result.hIDs:`
		`    examine(switchID)`

Path conformance [6, 9]	Checks for policy violations on certain properties of the path taken by a particular `flowID` (*e.g.*, path length no more than 6, or packets must avoid `switchID`). The controller may install the following query at the end-hosts.	**Endhost:**
		`Paths = getPaths(flowID, <*, *>, *)`
		`for path in Paths:`
		`  if len(path)>=6 or switchID in path:`
		`    pathInfo = getPathInfo(flowID, *)`
		`    result.append (pathInfo)`
		`if len(result) > 0:`
		`  Alarm (flowID, PC_FAIL, result)`

Traffic measurement	PathDump allows to write queries for various measurements such as traffic matrix, heavy hitters, top-*k* flows, and so forth. To obtain top-100 flows of a cloud service, the controller can execute this query at desired set of end-hosts participating in the service.	**Endhost:**
		`h = []; linkID = (*, *); tRange = (t1, t2)`
		`flows = getFlows (linkID, tRange)`
		`for flow in flows:`
		`  (bytes, pkts) = getCount (flow, tRange)`
		`if len(h) < 100 or bytes > h[0][0]:`
		`  if len(h) == 100:  heapq.heappop (h)`
		`  heapq.heappush (h, (bytes, flow))`
		`return h`

Table 3: Debugging applications.

Applications	Description	Pseudo code

DDoS/ Superspreader detection	A Superspreader host talks with more than *k*-destinations in a specific time period. A DDoS victim is contacted by more than *k* different sources in a specific time period. A controller may install a query at end-hosts to be executed at regular intervals. The query perform aggregation on fields in flowID, and identify DDoS victims and Superspreaders.	**Endhost:**
		`tRange = (t1, t2)`
		`flows = getFlows(<*,*>, tRange)`
		`ss = [], ddos = [] hID = hostID`
		`for fID in flows:`
		  `if fID.sIP==hID and fID.dIP not in ss:`
		      `ss.append(fID.dIP)`
		  `if fID.dIP==hID and fID.sIP not in ddos:`
		    `ddos.append(fID.sIP)`
		`if len(ss) > k:`
		   `Alarm(flows, SS_FOUND, [])`
		`if len(ddos) > k:`
		   `Alarm(flows, DDoS_FOUND,[])`

Heavy hitter detection	Find the flows consuming more than a certain percentage of link bandwidth. A controller may install a query at endhosts (see pseudo code) to monitor flows and their size at regular intervals.	**Endhost:**
		`tRange = (t1, t2)`
		`hh_size = 12MB #10% of 1 Gbps link`
		`flows = getFlows(<*,*>, tRange)`
		`for fID in flows:`
		   `(pkts, bytes) = getCount(fID, tRange)`
		`if bytes > hh_size:`
		      `Alarm(flowID, HH_FOUND, [])`

Isolation [6]	An example isolation policy looks like hosts in one group (say, A) should not talk with hosts in another group (say, B). A group can be described by a set of host IP addresses (*e.g.*, IP prefix).	**Endhost:**
		`grpA = A's IP prefix`
		`grpB = B's IP prefix`
		`tRange = (t1, t2)`
		`flows = getFlows(<*,*>, tRange)`
		`for fID in flows:`
		   `if fID.sIP in grpA and fID.dIP in grpB:`
		      `Alarm(flowID, ISOLATION_FAIL, [])`

Identifying poor connections	Identify connections with a large number of timeouts. The `getLossyFlows` query maintains a per-flow timeout counter. The counter is incremented if the inter-packet arrival time is greater than 300 msec. An alarm is raised when the timeout counter is greater than threshold times number of packets seen so far by a flow.	**Controller:**
		`install(hIDs, Query = getLossyFlows(Threshold), 0)`
		Endhost:
		`flowIDs = getLossyFlows(Threshold)`
		`for fID in flowIDs:`
		   `Alarm(fID, TOO_MANY_TIMEOUTS, [])`

Table 4: Debugging applications.

Applications	Description	Pseudo code
TCP out of order packet delivery [8]	Count the packets of a flow arrive in out of order. A controller can install a query at endhosts that perform checks on TCP packet sequence number.	**Endhost:** For each flow maintain two variables: expected sequence number and counter. For each packet: 1. If the tcp sequence number is not same as expected sequence number then increment the counter. 2. Add tcp sequence number and payload length, and assign to expected sequence number.
TCP non-monotonic [8]	Count the packets have the sequence number lower than maximum sequence number seen so far.	**Endhost:** Maintain two per-flow variables: maximum sequence number and counter. For each packet: 1. Increment counter if the tcp sequence number is lower than maximum sequence number. 2. Otherwise, update maximum sequence number with the tcp sequence number.
Flowlet size histogram [8]	Compute per-flow flowlet size distribution.	**Endhost:** For each flow maintain a list for flowlet sizes, counter, and timestamp. For each packet: 1. If the inter-packet arrival time is greater than threshold `Th`, append the counter value to the list, reset the counter, and update timestamp. 2. Otherwise, increment the counter, and update timestamp. 3. Finally compute per-flow flowlet size distribution from the list.
Packet counts [8]	Count per srcIP packets that traverse a particular switch.	**Controller:** 1. Get list of hostIDs from the switch. 2. On each hostID execute a query that filters flows that have switch in its path, then groups the filtered flows by srcIP, and returns the byte counts. 3. Aggregate responses from each hostID.
Packet drops on servers [13]	Localize packet drop sources (network vs. server).	**Controller:** Search for problematic flows on source and destination endhost TIBs. If found, then it is most likely a network problem.
Loop freedom [6]	Detect forwarding loops.	**Controller:** We leverage the fact that commodity SDN switches recognize only two VLAN tags in hardware, and processing more than two tags in a packet header involves switch CPU, consequently switch forwards packet to the controller and detect loops (more details in the PathDump paper [11]).
Overlay loop detection [13]	Loops between SLB and physical IP.	Since, an SLB could be an endhost connected to the network, packets in a loop may not reach the controller. Such events cannot be able to detect and debug by PathDump.
Protocol bugs [13]	Bugs in the implementation of network protocols (e.g., PFC, RDMA)	**Endhost:** Captures control packets (PFC, CNP, NACK) which can be further analyzed to understand health of an RDMA flow.
Netshark [6]	Network-wide path-aware packet logger. View properties of a packet at each hop in its path.	PathDump offers partial support with a view on sub-set of properties. For example, it can tell packet properties at a switch like input and output port, but not packet header values and matched flow table version.

Table 5: Debugging applications.

Applications	Description	Pseudo code
Incorrect packet modification [6]	Localize switch that modifies packet incorrectly.	Debugging this problem requires packet header information at each switch hop. Since, PathDump monitors packets at endhosts it may not pin-point the culprit switch. But, if packets going to different destinations are impacted, PathDump helps to localize to a small part of the network (steps to follow are similar to the silent random packet drop detection).
Traffic matrix [9]	Get traffic volume between all switch pairs in a network.	**Controller:** 1. Retrieve list of hostIDs from a pair of switches, and know the hostIDs present in both lists. 2. On each host TIB present in both lists, execute byte count query that first filters flows traverse both switches, computes sum of flow sizes, and returns the sum. 3. Finally aggregate all query responses.
Newly opened TCP connections [12]	Monitor number of new connections established at a server in a specific time period.	**Endhost:** For each packet, increment counter if the syn bit in tcp flags is set. At the end of a window, raise an alert if the counter exceeds threshold `Th`
TCP Incomplete flows [12]	Connections for which FIN packet is not yet seen.	**Endhost:** The agent upon receiving FIN packet, it evicts the corresponding flow from the flow table. So, at any instance, number of flows in the flow table tells the number of incomplete flows. Count flows at regular intervals, and raise an alert if the count exceeds threshold `Th`.
SYN flood attacks	This attack can be detected when the number of incomplete TCP handshakes are unusually high. In an incomplete TCP handshake, a server receives SYN and sends SYNACK packet, but no ACK, essential to establish a connection, from the source is received.	**Endhost:** Debugging this problem requires to maintain TCP state (that is, tracking status of a TCP connection). In the current design, the agent does not keep track of TCP state, so it might require design changes for stateful support.
Slowloris attack [1]	The byte transfer rate of many TCP connections is unusually lower than normal.	**Endhost:** At regular intervals (say, every 1 sec) compute number of distinct flows (connections) and total number of bytes received. Next, compute the average number of flows per byte (simple division). If the average exceeds `Th` raise an alarm.
Zorro attacks [10]	Unauthorized users login to vulnerable Internet-connected devices (IoT), gain shell access, and issue a sequence of shell commands, one of which contains the keyword “zorro"‘.	**Endhost:** The agent does not collect packet payloads. But, if necessary the agent could be extended to run regular expressions on packet payloads, and detect attacks that need string search in payloads.
DNS tunnel detection [5]	No TCP connection(s) after DNS query.	**Endhost:** DNS tunnel detection requires to read and store information in DNS headers (*e.g.*, data in DNS query response). Currently, the agent does not parse headers beyond layer 3. In the future, it can be enhanced to parse higher layer headers and detect problems in those layers.

Table 6: Debugging applications.

References

[1] Slow dos on the rise. `https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html`.

[2] Solving the mystery of link imbalance. `http://tinyurl.com/m9vv4zj`.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[4] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM*, 2015.

[5] A. Gupta, R. Harrison, A. Pawar, R. Birkner, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven network telemetry. *CoRR*, abs/1705.01049, 2017.

[6] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX NSDI*, 2014.

[7] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *ACM SIGCOMM*, 2016.

[8] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, 2017.

[9] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling Path Queries. In *USENIX NSDI*, 2016.

[10] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. Iotpot: Analysing the rise of iot compromises. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[11] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with pathdump. In *USENIX OSDI*, 2016.

[12] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. Quantitative network monitoring with netqre. In *ACM SIGCOMM*, 2017.

[13] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM*, 2015.