Example 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 the switch API and returns the hostIDs seen in an 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.
`getHighE2ELatencyFlows(delta)`	Return the `flowIDs` whose inter-arrival packet time is greater than the delta.
`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.
`getAbnormalFlows()`	Return the `flowIDs` whose whose `protocol = TCP`, and the packets arrive out of order.
`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

Application	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 priority flows and the low priority flow.
		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 at an upstream switch are delayed by a high priority flow. In the absence of the high priority flow, there would be no contention at the upstream switch.
	The analyze function in the pseudo code checks for traffic cascades, first by looking 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 [4]	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 a switch at the same time. Such an event might fill up switch buffer, causing high job finish time for the 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 [1]	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.

Application	Description	Pseudo code

Number of active connections	Find the number of active flows converge on the same switch interface in a short time period. This information allows to detect incast [2] traffic at an egress port. Moreover, the sum of flow bytes in a short period might suggests the queue utilization.	**Controller:**
		`linkID=(S1, S2), tRange=(t1, t2)`
		`flowIDs = [], bytes = 0, queue_limit = 100KB;`
		`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`
		`#Check if many flows are present in a long queue`
		`if len(flowIDs) > 25 and bytes > queue_limit:`
		`  return (flowIDs, bytes)`

Silent random packet drop detection [6]	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 = getPathInfo(flowID, tRange)`
		`  Alarm(fID, POOR_PERF, pathInfo)`
		Controller:
		`Runs localization algorithm such as MAX-COVERAGE on the received paths that potentially include faulty links.`

Path conformance [3, 5]	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`

Isolation [3]	Hosts in one group (A) should not talk with hosts in another group (B). A group can be described by a set of host IP addresses (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, [])`

Table 3: Debugging applications.

Application	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.	**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, [])`

High end-to-end latency	Capture flows that observe high inter-packet arrival times. The `getHighE2ELatencyFlows` query has a per-flow variable that captures arrival time of last seen packet. Upon arrival of a new packet, if the difference between arrival time of current and last seen packet is greater than delta, the query raises an alarm.	**Controller:**
		`install(hIDs,Query=getHighE2ELatencyFlows(delta), 0)`
		Endhost:
		`flowIDs = getHighE2ELatencyFlows(delta)`
		`for fID in flowIDs:`
		`  Alarm(fID, HIGH_E2E_LATENCY, [])`

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, [])`

TCP out of order packets delivery	Report flows receiving packets out of order. The `getAbnormalFlows` query has per-flow variable that stores the sequence number of last seen packet. The query raises an alarm, if the sequence number of newly arrived packet is less than the sequence number of last seen packet.	**Controller:**
		`install(hIDs, Query = getAbnormalFlows(), 0)`
		Endhost:
		`flowIDs = getAbnormalFlows()`
		`for fID in flowIDs:`
		`  Alarm(fID, OUT_OF_ORDER, [])`

Table 4: Debugging applications.

References

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

[2] 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.

[3] 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.

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

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

[6] 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.