DATA PLANE PROGRAMMING

MODULE 3 – IN NETWORK MONITORING, CACHING AND CONTROL (DVAD43)

HHK3.KAU.SE/DPP
AGENDA FOR WEBINAR

• Introduction to Course Module
• Introduction to Monitoring and Caching

Please note: The webinar will be recorded. We intend to make available the video for course participants. When enabling your camera and microphone, you agree that your video and audio will be recorded and made available electronically. In case you do not want your video/audio to be recorded, switch off your video and mic. You can interact with us via Slack, too.
• In Network Monitoring, Caching and Control
  – DVAD43: Module 3
Learning Goals:

- Give an account of basic principles and concepts of network monitoring, In-band Network Telemetry (INT), In-network caching, and control, → todays webinar
- describe techniques for network monitoring, INT, In-network caching, and control →2 more webinars
- explain how the INT framework can be programmed → 1 Exercise

Prerequisite: DVAD41!

https://www.kau.se/en/education/programmes-and-courses/courses/DVAD43
ASSIGNMENTS IN MODULE 3

• Everyone:
  – Active Participation (Webinar and Slack)
  – P4 Tutorials, basic exercises, read papers, watch videos
  – Quizz questions
  – Discussion posts

• For the Credit bearing course the following is required:
  – Submit assignment (graded)
  – Quizz and discussion posts
NEXT ONLINE MEETINGS FOR MODULE 3

• Online Schedule
  – Webinar 3: Course Introduction and Intro to In Network Monitoring, Caching and Control
    • Now
  – Webinar 2: In-Network Caching
    • Monday, 13th Mai, 17:00 – 19:00 Stockholm Time
    • https://kau-se.zoom.us/j/939364030
  – Webinar 3: In-Network Control
    • Monday, 27th Mai, 17:00 – 19:00 Stockholm Time
    • https://kau-se.zoom.us/j/480361456
NEXT STEPS

• Go to the course webpage https://hhk3.kau.se/dpp/
  – Make yourself familiar with the course homepage
  – Read syllabus
  – Start to read the papers, watch videos, etc.
  – Each module comes with a weekly plan for you.
  – Until the next webinar, see online schedule for week 1 and week 2

• Join Slack Channel
  – Several channels, see Webpage
  – Can also download Slack app to get notifications
• Questions? ➔ In Slack #monitoring-p4
• Wrapup and next steps
  – Go to menti.com and use the code 19 04 34
  – Mentimeter
IN-NETWORK MONITORING
Current Network Monitoring approaches

- (SDN)-switch maintains counters
  - Per port, flow, traffic aggregate (flow rule)
- Control Plane or Switch CPU polls counters periodically or is informed periodically
- Not fast enough
  - Network state changes rapidly
  - CPU stress too high for fine-granular monitoring
    - Sampling applied – Sflow/Netflow → loss of granularity
- Do not provide end-to-end monitoring
  - Need to correlate per switch state → difficult
• **In-band Network Telemetry**
  – Collect network state in the dataplane
    • Switch ID, Ingress, Egress port ID
    • Egress Link Utilization
    • Hop latency, Queue occupancy, Congestion Status
    • …
  – Re-use network packets that traverse the data plane to collect monitoring information
  – attach it in programmable way to packet headers
  – At line speed, in a programmable way
“Which path did my packet take?”

“I visited Switch 1 @780ns, Switch 9 @1.3μs, Switch 12 @2.4μs”

“Which rules did my packet follow?”

“In Switch 1, I followed rules 75 and 250. In Switch 9, I followed rules 3 and 80.”

<table>
<thead>
<tr>
<th>#</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>192.168.0/24</td>
</tr>
</tbody>
</table>
3. "How long did my packet queue at each switch?"

"Delay: 100ns, 200ns, 19740ns"

4. "Who did my packet share the queue with?"
3. "How long did my packet queue at each switch?"

4. "Who did my packet share the queue with?"

"Delay: 100ns, 200ns, 19740ns"
IN-BAND NETWORK TELEMETRY (INT) USING P4

Add: SwitchID, Arrival Time, Queue Delay, Matched Rules, ...

Original Packet

Log, Analyze Replay
Visualize
• Challenge is in **Where and How** to add what Telemetry items for network wide customized monitoring

**Problem:**
• When to create what probes
• When to inform Data Collection
• Tradeoff between granularity, workload and overhead/latency

/* INT: add switch id */
action int_set_header_0() {
    add_header(int_switch_id_header);
    modify_field(int_switch_id_header.switch_id,
                 global_config_metadata.switch_id);
}

/* INT: add ingress timestamp */
action int_set_header_1() {
    add_header(int_ingress_tstamp_header);
    modify_field(int_ingress_tstamp_header.ingress_tstamp,
                 i2e_metadata.ingress_tstamp);
}

/* INT: add egress timestamp */
action int_set_header_2() {
    add_header(int_egress_tstamp_header);
    modify_field(int_egress_tstamp_header.egress_tstamp,
                 eg_intr_md_from_parser_aux.egress_global_tstamp);
}
**Goal**

- Enable programmable observability and closed loop control based on analytics
  - Fault prediction, resource utilization prediction, anomaly detection, SLA violation detection
  - Activate smart probes and observe traffic for a target subscriber for a given time and notify events
INT OVERHEAD USING P4

1. Packets timestamped just before transmit
2. INT Headers pushed according to controller configuration
3. Packets timestamped when received on port
4. TX and RX timestamp difference calculated offline

SDN Controller

Installing INT rules

OSNT
OPEN SOURCE NETWORK TESTER

Packet
Packet with INT header

P4-NetFPGA running INT protocol written in P4
INT OVERHEAD USING P4

• INT Overhead benchmark results
  – NetFPGA: scales linearly with INT items to push
  – Edge more overhead due to more complex P4 code

<table>
<thead>
<tr>
<th>Header fields</th>
<th>Core (µs)</th>
<th>Edge (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.22</td>
<td>5.45</td>
</tr>
<tr>
<td>2</td>
<td>5.73</td>
<td>5.96</td>
</tr>
<tr>
<td>3</td>
<td>6.05</td>
<td>6.29</td>
</tr>
<tr>
<td>4</td>
<td>6.41</td>
<td>6.64</td>
</tr>
<tr>
<td>5</td>
<td>6.73</td>
<td>6.96</td>
</tr>
<tr>
<td>6</td>
<td>7.06</td>
<td>7.29</td>
</tr>
<tr>
<td>7</td>
<td>7.38</td>
<td>7.61</td>
</tr>
<tr>
<td>8</td>
<td>7.71</td>
<td>7.94</td>
</tr>
<tr>
<td>9</td>
<td>8.10</td>
<td>8.32</td>
</tr>
<tr>
<td>10</td>
<td>8.44</td>
<td>8.66</td>
</tr>
<tr>
<td>11</td>
<td>8.79</td>
<td>9.01</td>
</tr>
<tr>
<td>12</td>
<td>9.13</td>
<td>9.35</td>
</tr>
<tr>
<td>13</td>
<td>9.48</td>
<td>9.70</td>
</tr>
<tr>
<td>14</td>
<td>9.83</td>
<td>10.04</td>
</tr>
<tr>
<td>15</td>
<td>10.17</td>
<td>10.39</td>
</tr>
<tr>
<td>16</td>
<td>10.52</td>
<td>10.73</td>
</tr>
</tbody>
</table>
INT ASSIGNMENT MRI

Multi-Hop Route Inspection (MRI)

- send a low rate traffic from h1 to h2
- a high rate iperf traffic from h11 to h22.
- Reduce the BW of link s1-s2 to 512kbps in topology.json.
- The link s1-s2 is common between the flows and is a bottleneck
- Insert the telemetry items such as Queue Length using P4
- Capture packets at h2, and observe high queue size for that link.
MRI STRUCTS

```c
struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
    ipv4_option_t ipv4_option;
    mri_t mri;
    switch_t[MAX_HOPS] swtraces;
};

header mri_t {
    bit<16> count;
};

header switch_t {
    switchID_t swid;
    qdepth_t qdepth;
};
```
control MyEgress(inout headers hdr,
inout metadata meta, 
inout standard_metadata_t standard_metadata) {
    action add_swtrace(switchID_t swid) {
        hdr.mri.count = hdr.mri.count + 1;
        hdr.swtraces.push_front(1);
        // According to the P4_16 spec, pushed elements are invalid, so we need
        // to call setValid(). Older bmv2 versions would mark the new header(s)
        // valid automatically (P4_14 behavior), but starting with version 1.11,
        // bmv2 conforms with the P4_16 spec.
        hdr.swtraces[0].setValid();
        hdr.swtraces[0].swid = swid;
        hdr.swtraces[0].qdepth = (qdepth_t)standard_metadata.deq_qdepth;

        hdr.ipv4.ihl = hdr.ipv4.ihl + 2;
        hdr.ipv4_option.optionLength = hdr.ipv4_option.optionLength + 8;
        hdr.ipv4.totalLen = hdr.ipv4.totalLen + 8;
    }
}
MRI STATES

```haskell
state parse_mri {
    packet.extract(hdr.mri);
    meta.parser_metadata.remaining = hdr.mri.count;
    transition select(meta.parser_metadata.remaining) {
        0 : accept;
        default: parse_swtrace;
    }
}
```

```haskell
state parse_swtrace {
    packet.extract(hdr.swtraces.next);
    meta.parser_metadata.remaining = meta.parser_metadata.remaining - 1;
    transition select(meta.parser_metadata.remaining) {
        0 : accept;
        default: parse_swtrace;
    }
}
```

**SWTRACE**

REMAINING = 2 - 1 = 1
REMAINING = 1 - 1 = 0
control MyEgress(inout headers hdr, 
    inout metadata meta, 
    inout standard_metadata_t standard_metadata) {

    action add_swtrace(switchID_t swid) {
        hdr.mri.count = hdr.mri.count + 1;
        hdr.swtraces.push_front(1);
        // According to the P4_16 spec, pushed elements are invalid, so we need
        // to call setValid(). Older bmv2 versions would mark the new header(s)
        // valid automatically (P4_14 behavior), but starting with version 1.11,
        // bmv2 conforms with the P4_16 spec.
        hdr.swtraces[0].setValid();

        hdr.swtraces[0].swid = swid;
        hdr.swtraces[0].qdepth = (qdepth_t)standard_metadata.deq_qdepth;

        hdr.ipv4.ihl = hdr.ipv4.ihl + 2;
        hdr.ipv4_option.optionLength = hdr.ipv4_option.optionLength + 8;
        hdr.ipv4.totallen = hdr.ipv4.totallen + 8;
    }
}

typedef bit<32> switchID_t;

typedef bit<32> qdepth_t;

MRI EGRESS
INT ASSIGNMENT - RESULTS

```
\options

###[ MRI ]###
| copy_flag = 0L
| optclass = control
| option = 31L
| length = 20
| count = 2

\swtraces

###[ SwitchTrace ]###
| swid = 2
| qdepth = 0

###[ SwitchTrace ]###
| swid = 1
| qdepth = 17
```
MRI TIMESTAMPs

```plaintext
hdr.ioam_payload[0].tx_nsec =
     (bit<32>)meta.intrinsic_metadata.current_global_tstamp;
 hdr.gpe_ioam.len = hdr.gpe_ioam.len + 4;

hdr.ipv4.totalLen = hdr.ipv4.totalLen + 32;
hdr.udp.len = hdr.udp.len + 32;

[0] switch_id | bos : 0x0 | switch_id : 0xcafe |
[0] ingress_ts | bos : 0x0 | value : 0x7661ee71 |
[0] egress_ts  | bos : 0x0 | value : 0x7661fe71 |
[1] switch_id  | bos : 0x0 | switch_id : 0xcafe |
[1] ingress_ts | bos : 0x0 | value : 0x7661ca93 |
[1] egress_ts  | bos : 0x0 | value : 0x7661da93 |
[2] switch_id  | bos : 0x0 | switch_id : 0xcafe |
[2] ingress_ts | bos : 0x0 | value : 0x7661a076 |
[2] egress_ts  | bos : 0x1 | value : 0x0x7661ba93 |
```
update & merge

to update your local repository to the newest commit, execute

```
  git pull
```

in your working directory to *fetch* and *merge* remote changes.

to merge another branch into your active branch (e.g. master), use

```
  git merge <branch>
```

in both cases git tries to auto-merge changes. Unfortunately, this is not always possible and results in *conflicts*. You are responsible to merge those *conflicts* manually by editing the files shown by git. After changing, you need to mark them as merged with

```
  git add <filename>
```

before merging changes, you can also preview them by using

```
  git diff <source_branch> <target_branch>
```
• In-Network Caching
  – Caching of Key-value stores in the data plane
  – NetCache

• In-Network Control
  – Triggering control actions for e.g. Cyber-physical systems from within the data plane
  – FastReact