Performance Debugging in SDN Controllers - A Case Study

Sania Zara, Muhammad Umar Hameed, Adnan Iqbal, Noman Javed, Muhammad Saqib Ilyas
Namal College, Mianwali
Email: {sania.zara, umar.hameed, adnan.iqbal, noman.javed, saqib.ilyas}@namal.edu.pk

Abstract—OpenDaylight (ODL), a popular controller for Software Defined Networks (SDNs), has a known problem of gradual decrease in flow setup rate. After eight hours of continuous stress testing, flow setup rate of the controller degrades by 80%. We employ a step by step investigation based on behavioral testing of emulated switches and controller. We suggested a fix for the problem and modified version of ODL controller shows consistent flow setup rate during long-running tests.

I. INTRODUCTION
OpenDaylight (ODL) is a popular SDN controller. It is an open source, community driven project. ODL has an extensible and flexible architecture which supports several southbound plugins, including OpenFlow, and has thus seen support from several vendors.

Stress testing of SDN controllers is common to assess the production readiness. These tests typically consist of a long sequence of flow establishment requests generated by virtual switches. The rate at which the controller processes these requests is an important performance parameter. Stress testing of ODL using a tool known as CBench has revealed that the flow establishment rate degrades gradually during long running tests [1]. This bug was first reported in the release code-named Helium and has been well known in the ODL community for more than a year.

This paper presents our efforts to determine the root cause of this performance bug and a proposed solution. In this study, we started with all code being suspect and used a process of elimination to progressively mark portions of code clean. We only used standard network troubleshooting and software performance debugging techniques. First, through network packet traces, we determined whether the performance degradation was due to ODL or the virtual switches in the CBench test environment. Packet traces revealed a consistent request issuance rate from CBench, marking it as clean. On the other hand, a gradual increase was seen in ODL’s flow establishment service time, indicating that ODL is responsible for the decreasing flow establishment rate.

Having determined that ODL was responsible for the performance degradation, we tested for packet loss within the controller. The controller was also tested for memory leaks and thread-related issues. Next, we performed CPU profiling on ODL, through which we were able to locate the specific module within ODL that is responsible for the processing delay. Specifically, we found that during a long running test, the number of calls to a particular function increased gradually. This function is called once for every switch in the network while serving a flow establishment request. Even though CBench periodically created a new set of virtual switches and disconnected previously used switches, ODL was unnecessarily maintaining state for all the switches. As the test progressed, the size of the switch state collection in ODL grew, resulting in a gradual increase in the time required for flow establishment. We applied a fix to the code in which we removed the disconnected switches from the collection within ODL. Subsequent testing showed that the modified ODL’s performance in long running tests was consistent.

II. OPENDAYLIGHT
OpenDaylight (ODL) is a Java-based open-source project that is developed under a collaboration of more than 50 corporate entities, including Linux Foundation [2]. Since its inception in 2013, use of ODL is spreading rapidly and several organizations including AT&T, Comcast and Orange have adopted ODL. ODL’s architecture follows a service oriented approach, which is briefly described below.

A. Architecture
ODL has a layered architecture as shown in Fig. 1. The service requests by network applications are communicated to the Service Abstraction Layer (SAL) using the northbound APIs. The northbound plugins in the SAL intercept these requests. OpenDaylight uses RESTCONF [3] as its northbound protocol.

B. Performance Issues
Two key performance metrics for an SDN controller are latency and throughput. Latency is defined as the interval
between the issuance of a PacketIn message for establishment of a network flow and the corresponding response by the SDN controller. Throughput is defined as the number of PacketIn messages handled by an SDN controller per second. OpenFlow-based SDN controllers are generally benchmarked using CBench [4]. We performed stress testing of OpenDaylight (Lithium release) running on a Dell PowerEdge R220 server with 8 GB of RAM using WCbench - a CBench wrapper for ODL. We experimented with different number of switches and runs\(^1\). Our testing revealed that ODL performance degrades gradually with time. Fig. 2 shows results for three representative experiments. A degradation of 80% in the flow setup rate was observed in a continuous testing session lasting 8 hours. An investigation into OpenDaylight bug database revealed that this problem is already reported by ODL community as Bug 1395. This bug was first reported in July 2014 and is still unresolved.

III. A SYSTEMATIC APPROACH FOR PROBLEM LOCALIZATION

In this section, we follow a systematic problem localization process for locating the source of the performance problem discussed in the previous section. Before forming the initial hypotheses, one must understand the complete path a PacketIn takes after originating from a CBench switch. This path is shown in Fig. 4. Incoming packets from virtual switches instantiated in CBench are forwarded to OpenDaylight. These messages are decoded in an OpenFlowPlugin library named OpenFlowJava. Decoded data is then translated in an OpenFlow-Plugin and passed to a Model Driven Service Abstraction Layer (MD-SAL) for the registration of routing information. Data is then passed to northbound network applications - a learning switch application in case of CBench testing. A response to the PacketIn request is generated, which is passed down through the OpenFlowPlugin to OpenFlowJava. OpenFlowJava translates this response into a FlowMod OpenFlow message and passes it to the switch. CBench then records the total number of responses it receives from each switch and reports them in the form of responses/second.

Based on the above description, we make following three hypotheses for the degrading performance of ODL:

\(^1\)A run is a number of test iterations which typically last 100 seconds.

- Decrease in PacketIn rate.
- Increase in packet loss rate.
- Increase in processing time within ODL.

To test the above mentioned three speculations we devise different tests based on packet analysis and code profiling techniques using a stable release of Lithium. In all of the tests, controller and benchmarking tools were run on the same machine. The following subsections present the individual testing techniques in detail and their outcomes.

A. Packet Analysis using Wireshark

Methodology: In this test we computed two quantities from captured packet traces: a) end-to-end flow setup time for each PacketIn b) CBench latency in sending new PacketIn after receiving relevant FlowMod.

Result: Fig. 3 shows the average end-to-end flow setup time and CBench latency on 1\(^{st}\), 100\(^{th}\), 200\(^{th}\) and 300\(^{th}\) run. We observe that CBench latency is consistently small - always under 4 ms as compared to the flow setup time. The flow setup time increases linearly and goes beyond 100ms.

Lessons Learned: Since the packet processing delay for CBench is not increasing with time, and the switches immediately send the next PacketIn after reception of a FlowMod, CBench is absolved of any role in the deterioration of flow setup rate. Thus, ODL must be responsible for the decreasing performance.

B. Packet Count Analysis

Methodology: In order to get a packet count at different points of Fig. 4, we made use of statistics counters which are available in two modules of the controller: OpenFlowJava (OpenFlow protocol library) and OpenFlowPlugin. Counters in these modules give information about how many packets (PacketIns/FlowMods) were received by each module and how many were sent upstream (US) - to other controller modules or downstream (DS) - to the switches. Total packet counts at the entry and exit points shown in Fig. 4 were collected using JConsole [5] after 300 runs. This was done via the counters

<table>
<thead>
<tr>
<th>Table I: Total Packet Drop</th>
</tr>
</thead>
<tbody>
<tr>
<td>US_RECEIVED IN_OF_JAVA</td>
</tr>
<tr>
<td>1,463,543,131</td>
</tr>
</tbody>
</table>

Fig. 3: Results of Packet Analysis using Wireshark
US_RECEIVED_IN_OFJAVA and DS_FLOW_MODS_SENT.

**Result:** The results in Table I show a loss of roughly 64K packets after 8 hours of testing which is around 0.004% of the total incoming packets.

**Lessons Learned:** A small percentage of packet loss during a 300 runs test eliminated the possibility of excessive packet drop as a cause of performance deterioration.

### C. Memory Analysis

**Methodology:** In order to monitor the memory utilization and growth patterns of different objects in ODL, we used a well-known Java profiling tool named JProfiler.

**Result:** Fig. 5 illustrates the correlation between memory usage and flow setup rate (responses/second) decay. It reveals that over 300 runs, responses/second have deteriorated significantly (from over 60K to below 10K for 64 switches), but memory usage does not change significantly. These results were later verified using JConsole and YourKit and memory usage pattern was found to be the same.

**Lessons Learned:** We deduced that performance deterioration of ODL is not due to memory related issues.

### D. Thread Analysis

**Methodology:** Total number of threads and their states were recorded using JProfiler.

**Result:** Fig. 6 shows the total number of threads in different states at various points during a long running test session. The total number of threads remains fairly uniform throughout the test. There was no evidence of an abnormal increase in number of waiting threads with the passage of time. Moreover, no deadlocks were detected during the course of testing.

**Lessons Learned:** The possibility of an increased delay within ODL due to typical threads related issues is eliminated.

### E. CPU Profiling

**Methodology:** JProfiler was used for CPU profiling of the ODL code, with the aim of identifying Hot Spot functions.

**Result:** We noticed that time consumption of the poll() method of QueueKeeper class increases gradually with time. This method is called by a Harvester thread and is responsible for polling a set of queues. The Harvester thread takes PacketIns from these queues and forwards them for further processing. Table II shows the time consumed by this method averaged over 10 runs after every 100 runs. The same results were produced by instrumenting code with timers.

**Lessons Learned:** Delay in ODL processing time is due to an increase in the CPU consumption of the poll() method. This finding leads to the following possibilities:

- Method execution takes longer with the passage of time.
- Method invocations increase with time.
- Both of the above.

### F. Call Graph Analysis

**Methodology:** Call graph functionality of JProfiler was used to record the invocation counts and time spent within the problematic function. These measurements were recorded after every hundred runs.

**Result:** Table III shows the invocation count during different

\[ \text{TABLE III: Time spent in poll() method} \]

<table>
<thead>
<tr>
<th>Run Number</th>
<th>1-10</th>
<th>90-100</th>
<th>190-200</th>
<th>290-300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Consumed (sec)</td>
<td>92.3</td>
<td>587</td>
<td>634</td>
<td>713</td>
</tr>
</tbody>
</table>

---

2Hot Spot functions in JProfiler are functions that consume the most CPU time.
JProfiler after every of queues. We monitored the growth of the collection using the poll() method is called for every element in this collection a thread responsible for harvesting queues in a collection. Each number of calls to a poll() method. This method gets called in the reason behind degrading performance is an increasing size of collection warranted a deeper look into the behavior of ODL.

IV. ROOT CAUSE ANALYSIS AND RESOLUTION OF THE PERFORMANCE ISSUE

Systematic investigation process of Section III revealed that the reason behind degrading performance is an increasing number of calls to a poll() method. This method gets called in a thread responsible for harvesting queues in a collection. Each queue is maintained to hold parsed data coming from a switch. The poll() method is called for every element in this collection of queues. We monitored the growth of the collection using JProfiler after every 100 runs and found that the collection grows as a product of run number and number of switches.

The relationship between the number of switches and the size of collection remained constant and equal to the number of switches. Moreover, the CPU time consumption and invocation count of poll() method also remained constant with every run. The controller’s responses/second also remained uniform as shown in Fig. 7.

These results verified our hypothesis that the problem of performance decay is due to an increase in the size of the aforementioned collection of queues. Increasing size of this collection results in an increase in the number of calls to the poll() method and hence an increase in the overall CPU time consumed by ODL.

To fix this issue, we propose that whenever a switch is disconnected from ODL, its queues should be removed from the queue collection. This would ensure a fixed collection size and also a fixed number of invocations of poll() method throughout the long running test.

The modified version of ODL was tested using a new set of switches after every run. Fig. 8 shows the responses/second for 300 runs using 16, 32 and 64 switches. It can be seen that now the responses/second remain uniform for long duration tests even while using variable set of switches.

V. CONCLUSION

This paper has presented a systematic approach to localize performance issues in OpenDaylight. This approach solely relies on commonly available tools. Application of this approach to diagnose latency issues in OpenDaylight revealed inappropriate use of queues as the definite cause of problem. Improvement in queue implementation code resulted in elimination of the aforementioned performance issue.

REFERENCES


<table>
<thead>
<tr>
<th>Run Number</th>
<th>1-10</th>
<th>90-100</th>
<th>190-200</th>
<th>290-300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation Counts (million)</td>
<td>94</td>
<td>563</td>
<td>675</td>
<td>713</td>
</tr>
<tr>
<td>Time Consumed per Invocation (microseconds)</td>
<td>0.98</td>
<td>1.04</td>
<td>0.94</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Table III: Invocation count and execution time per invocation for poll() method.

Fig. 7: Performance with Fixed Switches

Fig. 8: Performance after Bug-fix