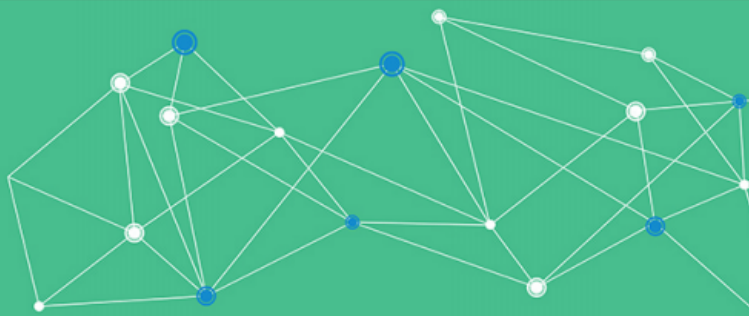
C.R.A.V.E.D



Final Report - Distributed Monitoring in SDN

Team: C.R.A.V.E.D.

Group members:

Clément Bertier, bertier@kth.se
Rebecca Portelli, rpor@kth.se
Alexandra Stagkopoulou, stagk@kth.se
Vikrant Nikam, nikam@kth.se
Efthymios Neroutsos, eftner@kth.se
Dongyang Zhang, dongyang@kth.se

Course: Communication System Design
Course number: IK 2200
Year: 2015-2016

Content

Instruction and Motivation

During the last decades the number of connected devices to the Internet, the applications and services that run on the network have increased rapidly. However, the underlying network has not been changed much. The routing is usually based on the destination IP prefix and the network management is stiff because it is restricted by the vendors' closed configuration. The need for more flexible, agile and vendor-neutral communication networks lead to the idea of decoupling the control and data plane and creating a network that is programmable by a logically centralized controller. This is the era of Software Defined Networking (SDN). The leading protocol of such a network is OpenFlow [1] developed by the Stanford University in 2008. The most characteristic benefits of SDN are the programmability of the centralized controller resulting in a dynamic packet forwarding based on several policies, the clear topology view of the controller and the reduced investment costs from the enterprises point of view [2]. SDN offers the opportunity to the network operators to avoid the low-level vendor-specific configurations and defines the required policies in a high level programmable environment.

Problem

Current SDN environments restrict their monitoring capabilities in gathering packet and byte counters from the dataplane as well as several QoS metrics. Network management in SDN environment needs to be enhanced to obtain the real network view by gathering additional monitoring parameters like bandwidth, RTT (Round Trip Time), dropped packets, link usage and expired TTL (Time To Live), TCP (Transmission Control Protocol) retransmissions along with hardware parameters like CPU/memory/cache usage, which will provide the state of each device in the network.

The goal of this project is to build a more verbose monitoring system for SDN-based networks. We consider a tree-based topology as an example network. The monitoring system will report the states or problems of the network and present them in a website for network administrators. In addition, we provide a wrapper as an interface to demonstrate the functionality of this monitoring system.

With this system, network administrators can have a real-time view of the network and monitor relevant metrics, including throughput, RTT between the controller and switches, retransmission packets etc. These metrics can provide a better understanding about the network. In addition, we also show alerts when there are abnormal values in the metrics. The alert is useful for administrators to react to network issues more timely. A database is kept in the background providing a history of the monitored parameters and allowing the operator to query and retrieve useful information.

Approach

In this section, all the modules of this project will be presented. The overall structure of our system is shown in Figure 1. Each box corresponds to a module that will be described in the following sections.

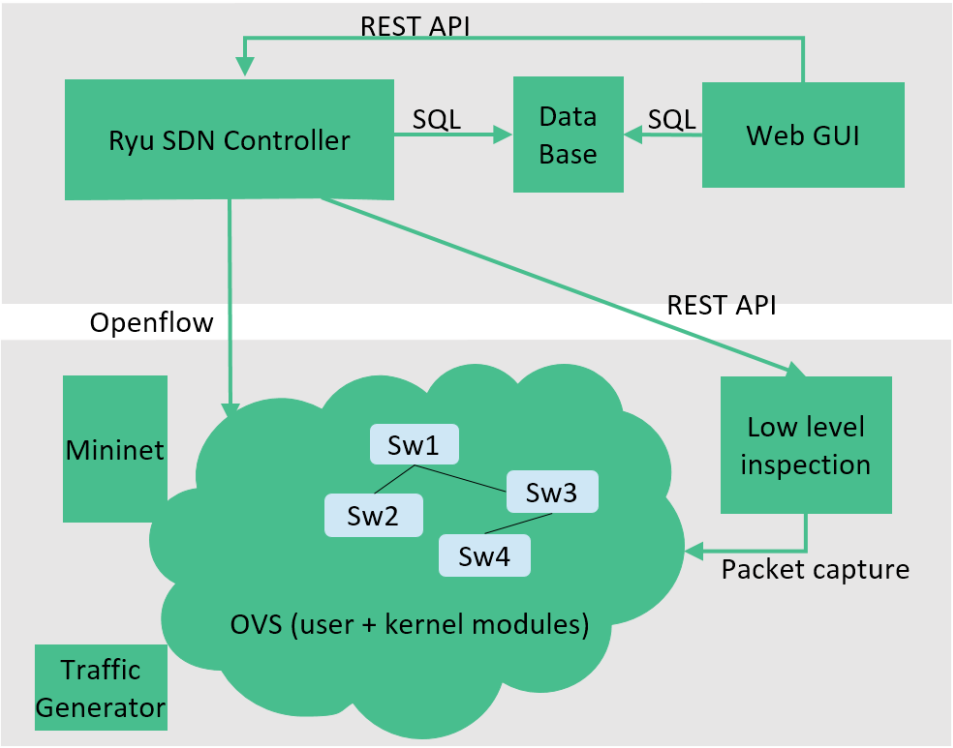


Figure 1: Overall architecture

Topology

The topology is a tree-like consisting of fifteen switches and sixteen hosts representing a data center. Eight hosts are used as end users while the remaining eight hosts are used as servers where different services are implemented. These services are Hypertext Transfer Protocol (HTTP), Domain Name System (DNS), File Transfer Protocol (FTP) and Voice over IP (VoIP). In general the topology is implemented so that various types of traffic can be supported such as TCP and UDP. More details about each specific type of traffic and how it is generated will be given in Traffic Generation section.

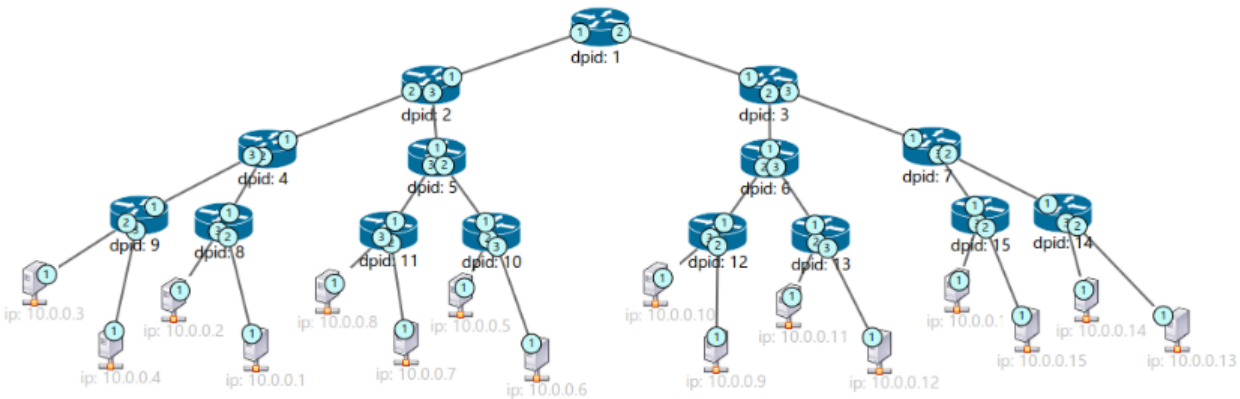


Figure 2: Topology

For the implementation of the topology, we used mininet [13], a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run in Linux containers and the switches support the OpenFlow protocol, which is used for the communication between the switches and the controller. Mininet can use either Open vSwitch [14], where the default kernel-assisted implements the fast path in the kernel-space and the slow path in user-space or ofsoftswitch [15] as the dataplane technology. For the purposes of this project,

Mininet and OVS v.2.4.90 process are running in user-space mode in a cost performance in order to more effectively stress the system and collect valuable information regarding the system CPU utilization and memory consumption.

Controller

For the implementation of the controller we use the Ryu SDN controller[3]. Ryu provides software components with well defined SDN APIs that make it easier to communicate with the switches and gather the required statistics. Ryu also supports the OpenFlow v.1.3 protocol which was used to manage the switches, control the traffic flow and routing in the network and exchange vital information with the switches in order to effectively monitor the network and influence the way it works.

The controller is the only responsible element in the network to make routing decisions. If the packet is a layer 2 packet (e.g. ARP packet) then the routing is based on the source and destination MAC address, while for IP packets (e.g. ICMP, TCP packets) the routing is done according to the IP source and destination address.

The controller is not only responsible for making routing decisions but is also responsible for collecting some statistics that are useful for the network monitoring. This is done by exchanging OpenFlow messages with the switches. With these messages, the controller is able to request specific information from each switch that can be used to evaluate the current state of the network.

First of all the controller periodically requests all the flows installed on the switches as well as port statistics for each port of each switch. Flow statistics include information about the type of the packets (TCP/UDP/ICMP/ARP), source and destination addresses and ports, the number of packets that match the specific flow and finally the actions taken when a packet matches a specific flow (e.g. decrement TTL by one). Port statistics are related to the number of packets and bytes received and sent from each port as well as error-packets received.

Moreover, the controller collects the number of dropped packets per switch, when the packet is dropped due to a flow instruction (e.g. when using a firewall that drops packets). In order to accomplish this, the controller periodically checks for flows with empty instruction set. According to OpenFlow, an empty instruction set indicates that the switch should drop the packet. When such a flow is identified, the controller records the number of packets that matched this flow as well as the identifier of the switch that forwards this flow. The total number of recorded dropped packets per switch is forwarded to the database. Finally, the controller resets the counter of the flow sending an OpenFlow request.

Another statistic measured from the controller is the RTT for the communication between the controller and each switch. In order to get the RTT values, OpenFlow echo request and reply messages were used. As a result, the RTT is the time between the echo request issued by the controller and the echo reply received by the controller.

All the above information and statistics which are gathered periodically from the controller are then pushed to the database in order to be finally displayed on the GUI.

On top of what mentioned above, the controller is also gathering statistics that are collected by the low level inspection module, which will be described in a detailed way later in this report. For example, when the low level inspection module records a retransmission, it has to send this information to the controller. This communication is done by using REST API [16] (Representational State Transfer), a software architectural style of the web based on HTTP

protocol . The controller is not able to request information from the low level module as it is not needed, however it should always be able to accept information sent to it. More specifically, the statistics that are sent from the low level inspection module include the number of TCP retransmissions per flow, the number of out of order packets per flow and the number of packets dropped because of the TTL expiration. Once again, all the information gathered from the low level inspection module is pushed to the database.

Apart from gathering statistics and providing routing decisions, the controller is also used to provoke some errors in the network so that we can test that the monitoring tool developed for the SDN network works as expected. For example, the controller causes TCP retransmissions on a specific TCP flow. This is achieved in the following way: when the controller receives a packet from a target flow, it sends the packet back to a particular output port of that switch with instructions regarding how to forward it, but without installing any flow on the switch. As a result, all packets for this flow, arriving at the switch, will be sent to the controller, then back to the switch and they will finally be forwarded. The delay caused by this operation in the packet delivery is large enough to provoke TCP retransmissions which are recorded by the low level inspection module.

CPU and Memory Statistics

CPU and Memory statistics was one of the most important metrics to be collected per switch. It turned out to be very challenging, because in order to collect individual CPU & memory statistics per switch it is necessary that each switch is managed by individual ovs-vswitchd process. However, there is only a ovs-vswitchd process running in the background which manages all the OVS switches [11]. As a result, we could not collect separate CPU & Memory metrics for each switch. In order to solve this problem, we analysed and tried various approaches as mentioned below which were not successful:

1. Collecting CPU statistics of switches using cpuacct subsystem, however all switches run under same root process so we cannot distinctly figure out individual statistics.
2. Multithreading of ovs-switch using ovs commands. It did not help because multi-threading support is for software data paths to use for handling new flows.
3. Fetching CPU statistics of ovs-vswitchd process from Open vSwitch Database ovsvswitchdb. CPU & memory statistics such as CPU (number of CPU processors available), memory (RAM allocated, RAM in use etc.), load average (system load average over time intervals) can be retrieved from ovs-vswitchdb. However, these values does not give correct statistics as ovs-vswitchd process is running in userspace and only the first packet of a new flow goes to user-space, while the following packets use the OVS kernel module.
4. Using Docker containers to collect CPU statistics. However, Docker containers only represent hosts in our scenario. OVS switches are still running on host machine under a single ovsvswitchd process. Hence this approach could not be considered as correct alternative for CPU statistics collection from switches.
5. We also considered option of using Intel DPDK [4] to run individual ovs-vswitchd process for each switch and operate Open vSwitch entirely in userspace. However considering the timeline and complexity of using Intel DPDK, we discarded this approach.

After considering all the above approaches, we decided to monitor the ovs-vswitchd process and to run OVS switch userspace mode instead of running it in kernel-space mode in order to get as a whole statistics of this process. It is possible to launch Mininet with the Open vSwitch in user-space mode by passing “datapath=user” switch parameter.

In order to collect information about the CPU utilization and memory consumption various profiling tools were analyzed such as sysprof, valgrind, systemtap, pidstat, htop, oprofile etc. We opted for Pidstat [5] tool since it can be used to monitor individual processes and provides CPU & memory statistics for that process using a single command. By monitoring “ovs-vswitchd” process, the overall memory and CPU utilization of OVS switch is obtained. Following CPU and memory statistics is collected using pidstat tool after 15 seconds interval periodically,

- %CPU: Total percentage of CPU time consumed by the task.
- %usr: Percentage of CPU consumed by the task while executing at the user-space (application)
- %system: Percentage of CPU used by the task while executing at the kernel-space
- %Memory: Total percentage of memory used by the task.
- minflt/s: Total number of minor faults the task has made per second, those which have not required loading a memory page from disk.
- majflt/s: Total number of major faults the task has made per second, those which have required loading a memory page from disk.

CPU/memory monitoring is performed from controller and CPU/memory statistics data are pushed to database after every data collection interval.

Traffic Generation

In order to be able to inject traffic into the network, a traffic generation module was implemented. It supports TCP, UDP and ICMP protocols. Specifically, the services which produce TCP traffic are HTTP and FTP (Web and FTP servers). For the generation of UDP traffic, the DNS and SIP (VoIP) protocols are used. Apart from the above, the iperf tool was used to inject both TCP and UDP traffic in the network and Ping command to generate ICMP traffic.

Traffic generation module is responsible for randomly selecting two hosts and a traffic type and injecting one of the traffic patterns above (randomly). It consists of two important classes namely Sender and Receiver. These classes randomly select one of the following hosts as sender (end user) and receiver (server).

End User Hosts	Servers
h1,h2,h3,h4,h5,h6,h7,h8	t1,t2,t3,t4,u1,u2,u3,u4

Table 1. End user host and server list

Traffic generator also selects one of the following traffic types and invokes the traffic generation by passing additional parameters if necessary.

Traffic Type	Additional parameters
Ping	Count
HTTP	-
FTP	File (size)
VOIP	Call rate, Number of calls
DNS	-
UDP(iperf)	Bandwidth, time
TCP (iperf)	-

Table 2. Traffic types

Once traffic generator selects two hosts and invokes traffic generation, Sender class generates request depending upon the type of traffic and Receiver class enables the chosen service so that request sent from the sender can be accepted. In order to simulate a realistic traffic scenario, sender, receiver and traffic type is randomly chosen and there is an infinite loop which issues the requests so that traffic in the network run continuously.

Low Level Inspection Module

The purpose of the module is to collect TCP retransmissions, out-of-order and expired TTL packet counters, which requires the module to inspect the packets, extract information from different header field, and perform relevant calculations.

The overall structure of the module is illustrated in Figure 3.

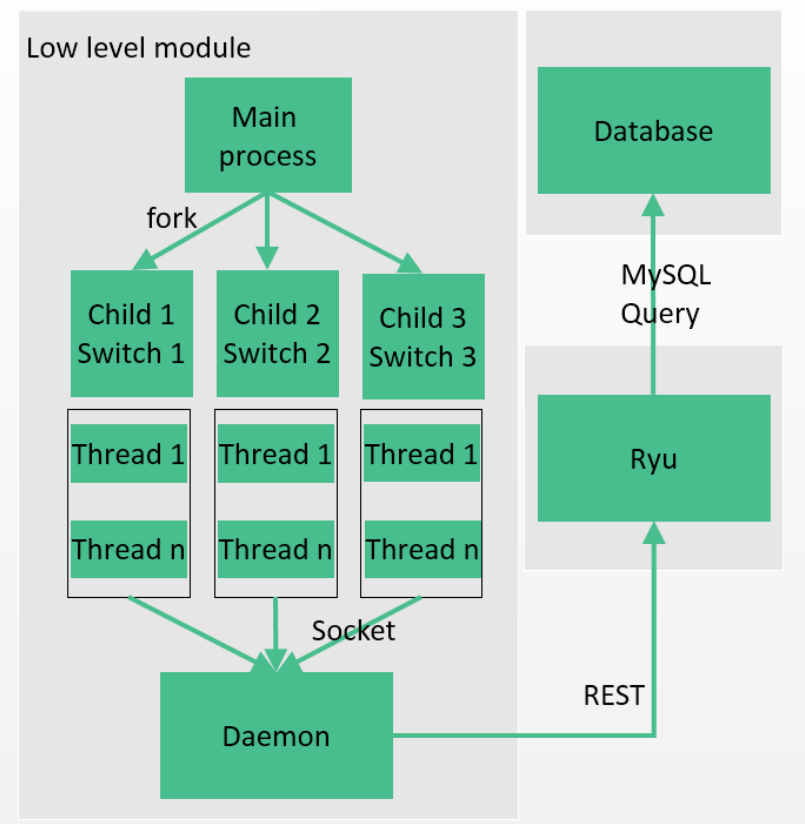


Figure 3. Low level inspection module

Library - libpcap

To effectively inspect packets, using available libraries is a good start instead of starting from scratch and pure socket programming. After comparison, we choose libpcap [6] because of the following reasons. First, libpcap is popular and powerful, supporting different platforms and a wide range of capabilities. Second, both tcpdump [6] and Wireshark [12] is based on libpcap to capture data. Since we are familiar with tcpdump and wireshark, using libpcap is a convenient startpoint. In addition, there are a lot of documentations and references for libpcap, which is very useful for beginners.

Capturing on multiple interfaces simultaneously

To monitor the whole network, we need to listen on all the network interfaces of the switches to capture packets. In our network there are 15 switches, adding up to 44 interfaces to be

monitored at the same time. Naturally we have one process for each switch to monitor its own interfaces, that is 15 processes in total. By isolating the processes, our intention is to be realistic and also it can be helpful to see how the load of each individual switch is by checking the load of its inspection process. To lighten the load, we have threads inside the processes, one thread corresponding to each interface.

In the implementation, we fork one process for each switch, and use C++ multi-threaded function inside each process.

Retransmission/out-of-order/expired TTL packet counters

Retransmission and out-of-order TCP packets are end-to-end concepts, which means we need to collect them at the endpoints of a connection.

A C++ structure is constructed to save information of all the flows in one switch, in both directions. Saved information including Flow ID (consisting of source/destination IP address and port), Acknowledge number, Sequence number, timestamp etc. Thus, by comparing the information we can decide whether retransmission or out-of-order packet occurs. The detection of packets with expired TTL values is more straightforward by looking into the header information of the TTL value.

In all, each switch maintains information of all the flows that go through it and calculates related counters. Mutex has to be used to prevent conflicts when different threads are accessing a common data structure.

Sending gathered data to controller via daemon

Sending the gathered data to database is the final step. After gathering the data from a switch's interfaces, the switch is supposed to send the updated values to Ryu controller, which will relay the values to database.

The communication between controller and low level inspection module is performed via REST API provided by Ryu controller. For the sake of convenience, we use library libcurl to perform the HTTP POST function.

One of the most challenging part is the strategy of sending the data. Depending on the traffic in the network, the counts may be updated frequently, maybe 1000 times per second, or less than 10 times per second. It is not realistic to update the value whenever there is update in the counts, in which case a heavy traffic load may lead to large number of updates to the controller and inject more load to the network.

The solution we chose is to implement an intermediate process, which is running continuously in the background. The inspection program sends the updated value to this daemon process whenever there are updates through a Unix Domain Socket. The advantage of using the Unix Domain Socket is that the write operation using send family function is atomic, making the sending process fast and the write operations don't conflict with each other. Then, daemon process will send updates to the Ryu controller in batches periodically, saving bandwidth and time.

Database & Cache

Database structure

The purpose of the database was a simple aggregation of all the metrics to be later processed into meaningful data. As it was unnecessary to bother with complex DBMS relationships between tables, most of the data is just stored as raw data (meaning as they come) in a table regarding its origin. However we split the database into two parts: the first one was a hard-coded version of the topology of the network, to be able to refer to it each time there was a need for a reference to interfaces or switches for instance; the second part consisted of the dynamic storage of the data.

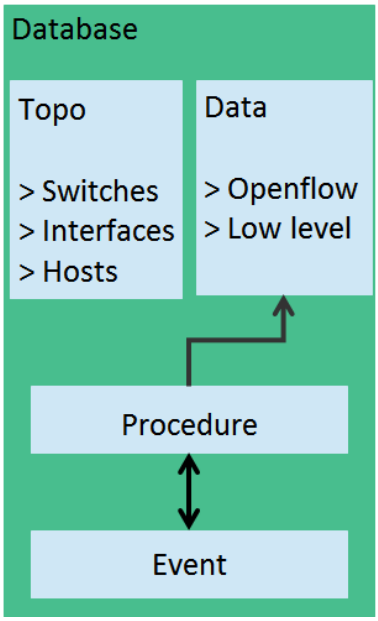


Figure 4. Structure of database

The dynamic storage of the data was divided in two parts: openflow related metrics (flows, ports statistics) and the low-level module acquired metrics. All of the dynamic data was inserted from the controller to the database even for low-level metrics that were initially passed to the controller through the REST API and then, if properly communicated, pushed into the database.

The procedures aim to clean the database, meaning keep the data up-to-date to be relevant and archive the old data if needed. The purpose of the archives (usually namee craved.XXX_backup) was to provide long-term information of the network without overloading the frequently queried tables, i.e. the ones the cache uses. This is done by packing data (calculating the average and variance of the selected values) over a particular range of time and inserting this entry in a backup table. These tables can be queried by the administrator. The events are a redundant way of calling the procedures when needed, typically after 1 or 10 minutes.

Cache

We decided to implement a cache strategy at a very early version of the project as we were afraid that triggering multiple queries from each GUI user would overwhelm the CPU. This process has been implemented through the use of a single script (controller_craved/querydb.pl). The script executes a series of SQL queries and packs them into JSON format, finally redirecting its output towards a file directly accessible from the web-server. With the JSON formatting, the user can immediately start processing the retrieved data through the use of javascript, making it efficient.

The cache was built to be extendable, such that one can add a new metric within a single line of code.

Web GUI

The subsequent figure illustrates how the pages in the web GUI interact with the controller, the database and the cache in order to retrieve the necessary data to visualize the network's topology, chart statistics and events visualization as well as results returned through database querying.

The main tools used to develop a user friendly web GUI were:

1. HTML 5: used to utilize new features over the traditional HTML, specifically for user interactivity with the GUI
2. CSS 3: used for the ease of the provided styling features.
3. JavaScript libraries: used for displaying connected nodes to represent the topology, to generate and plot charts, to run processes in parallel and to calculate statistics.

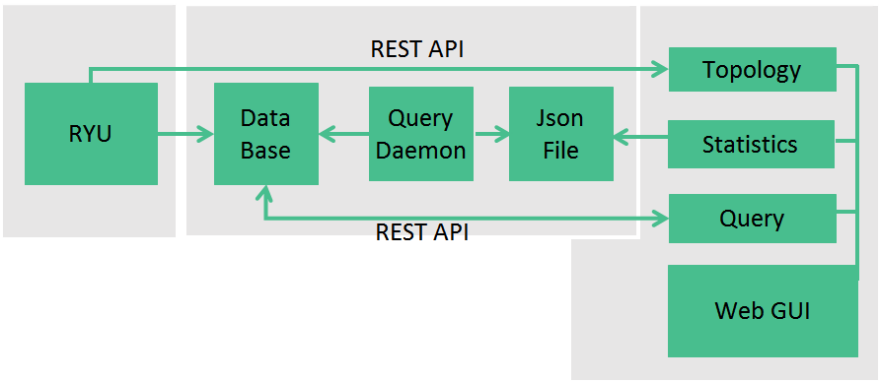


Figure 5. Interaction of Web GUI with other modules

Topology

The Topology page enables a network administrator to visualize the current network topology. This was accomplished by extending the already provided topology viewer by the Ryu framework. As displayed in the above Figure 5, REST API is used to dynamically get the necessary information related to the nodes present in the network topology and storing them in the relevant JSON object. The labelled-force graph from D3.js [7] library is used in order to display the nodes and easily interact with them.

Statistics - Charts

The Statistics page offers several detailed charts to illustrate network measurements. These charts are:

1. Total received bytes / packets per switch
2. Received bytes / packets at a time per port for a specific switch
3. Total number of expired TTLs per switch
4. Total number of expired TTLs per flow
5. Total number of dropped packets per switch
6. Total number of out of order packets per flow
7. Number of TCP retransmissions per flow
8. Average RTTs between controller and each switch including the respective maximum and minimum 98% confidence level using the student t-distribution.

9. The percentage of CPU usage for user-space, kernel-space and the total percentage as well as the percentage of the memory used.

These charts are generated dynamically and updated every 10 seconds by getting data from the cache, and refining it into JSON objects for chart plotting purposes. The plots are developed with C3.js [8] which is a library that provides easy to use and customizable charts. This library was opted to be used instead of other JavaScript libraries because it generates D3-based charts without having to use the complex D3.js library. However, it still permits integration with D3.js. In this case, D3.js is used to partially generate the visualization for the RTT chart where the D3.js confidence intervals plotting are combined to the C3.js bar chart which show the respective average RTT value.

Lastly, the generated charts have customized tooltips to provide a network administrator with more data related to a particular statistic when hovering on a data point in a chart.

Statistics - Alerts

In order to notify a network administrator regarding important events that occur in the network, the GUI provides alert notifications for:

1. RTTs - An alert is generated when a switch's last value present in the cache is lower than the minimum confidence interval or higher than the maximum confidence interval for that particular switch.
2. Expired TTL, Retransmission, Out of order packets - When a new flow or an existing flow with a higher statistic value than before is detected in the cache, an alert is generated.
3. Dropped packets - Similar to the previous condition, the alerts are generated when a new switch or an already existing switch with a higher dropped packet count than before is detected in the cache.

These alerts operate by first storing the number of RTT values per switch in nested JSON arrays for the statistics need to be monitored from the start of opening the website. The library async.js [9] is used to run this storing phase in parallel for 15 times (one for every switch and per port for the received bytes and packets) every 10 seconds until 1 minute since the website was opened.

The next phase starts checking for abnormalities and displays an alert for every abnormality encountered. The algorithm calculates the 98% minimum and maximum confidence intervals with a student t-distribution (using jstat.js [10] library) for the RTT statistics. Again this runs for 15 threads, one per switch. Additionally, the calculations are stored in other nested JSON arrays representing the switch identifier, the relevant statistic and the port number if needed.

Subsequently the algorithm proceeds to compare the last RTT value that arrived in the cache, with the respective minimum and maximum confidence intervals. If this value is greater than the maximum or less than the minimum confidence interval, an alert is displayed. As for the packets dropped, out of order packets, expired TTL and retransmission statistics, whenever the value in the cache is new or increases an alert is shown.

The visualization for the alerts was decided to take the form of changing the background for the respective statistics' menu item to red in the Statistics page including the background for the Statistics header menu item. In addition to this, an alert box is generated at the bottom of the chart on the Statistics page in order to give a more detailed description about the alert that occurred. The information in an alert box displays a collective amount of switch identifiers or flows

that have abnormalities for that particular statistic along with the respective statistic chart's name to view.

In the end, a First In First Out (FIFO) swap is made in order to keep the stored values for the RTT up to date. The whole process of recalculating the RTTs' confidence intervals, comparison and FIFO swapping is repeated again every 10 seconds for each thread. As for the rest of the statistics that are monitored for alerts, they keep all the details for the abnormal values that were encountered for future comparison and decision making until the web site is closed. Lastly this process is also repeated every 10 seconds for monitoring purposes.

Database Query

The Database Query page allows a network administrator to insert a raw MySQL query to retrieve data in a table format from the database. Otherwise, he/she is given the choice to choose from a set of menu items with pre installed MySQL queries. The page treats the input query with AJAX and refreshes the result area in the page with the requested query. The query is sent through the use of REST API to the controller. In case that an error with the input query is encountered the administrator is notified about the error and is asked to re-try.

Results

With the monitoring system, the network administrator can view the network easily through web GUI. Here are some examples of the interface.

Figure 6 shows the network topology view. The structure of the network is clearly shown, with necessary information such as dpid of each switch, host IP address etc.

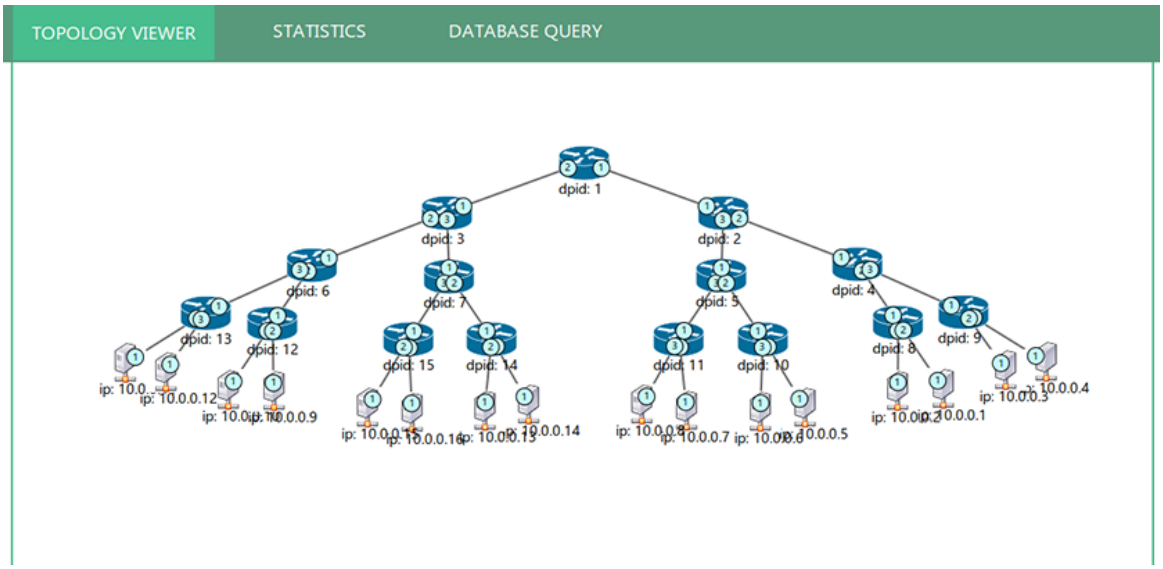


Figure 6. Topology overview

Figure 7 shows an example of statics that are collected from the system. The displayed metrics is Received Bytes in each interfaces of each switches. More detailed information can be shown when the mouse is hovering on each bar.

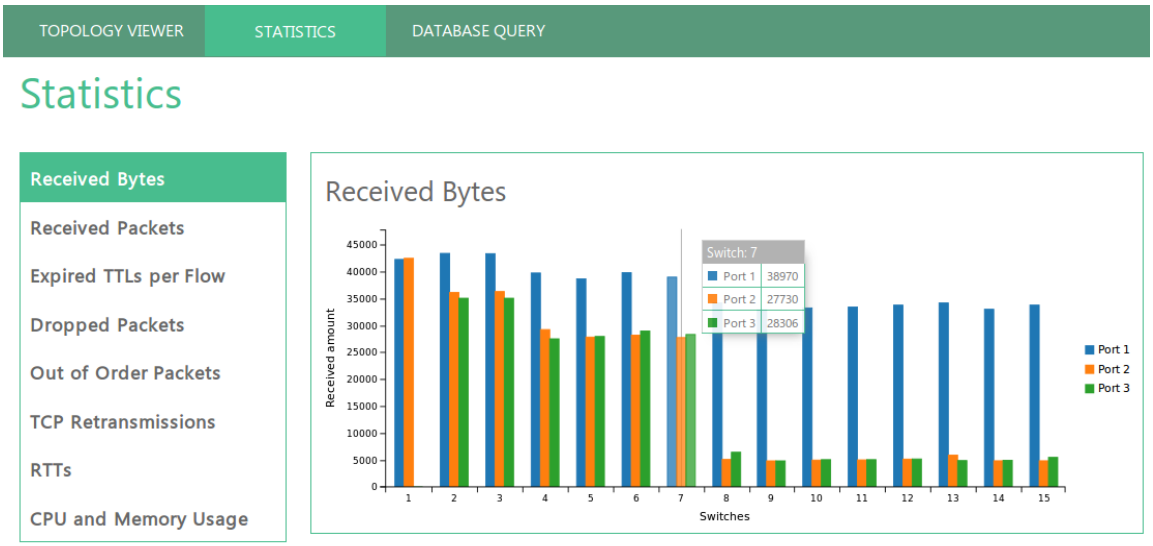


Figure 7. Statistics

Figure 8 shows the interface when there is an alert. In this case the alert is triggered by the Out of Order Packets, which is in red. Detailed information are displayed collectively in one alert box under the chart so that network administrators can take a look at the problem.

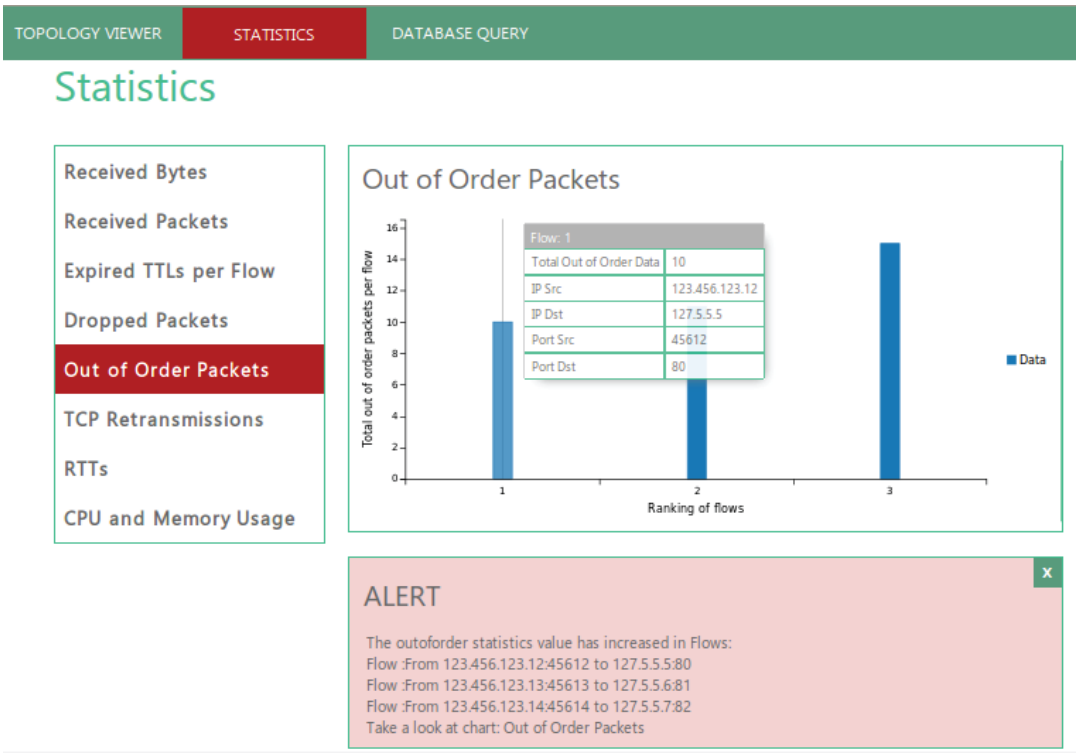


Figure 8. Statistics with error

Figure 9 shows the interface of database query. MySQL queries are performed if the network administrators are interested in getting information from database directly.

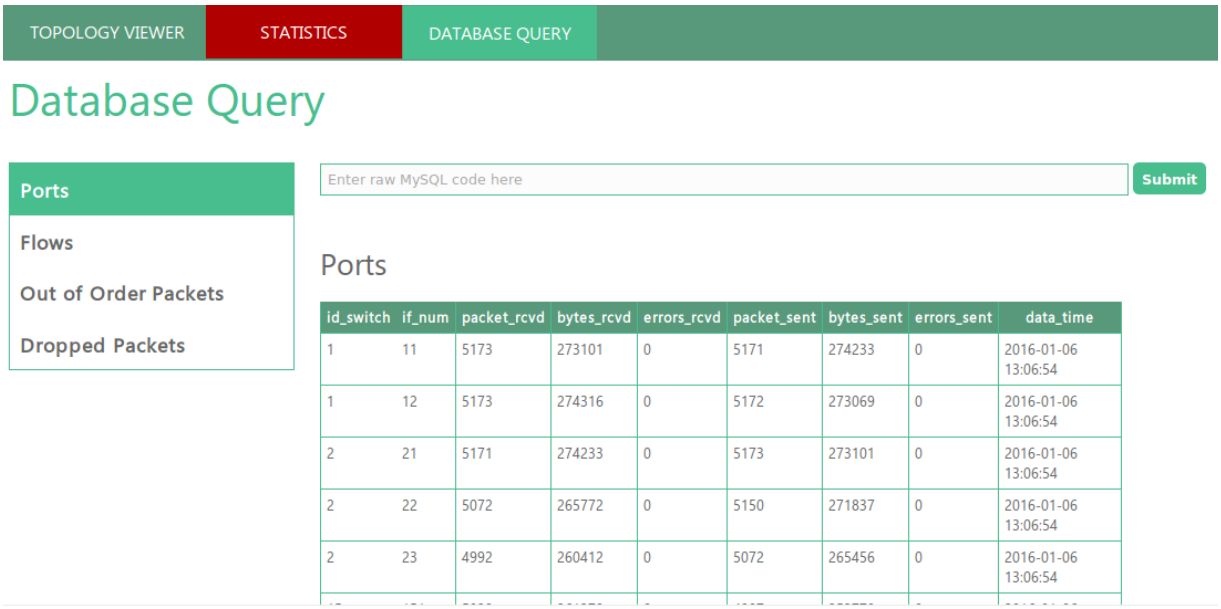


Figure 9. Database query

A demonstration video can be found via the following link. In the video we introduces the interfaces and some operations. (Link to video: https://www.youtube.com/watch?v=e9E6QPDt9hY)

Further work

With the monitoring system we have built, network administrators can already achieve better understanding about the network with more detailed metrics. However, there are some options left to be improved or investigated to enable better monitoring.

In order to gather statistics about CPU for each switch, one potential solution is to use DPDK. DPDK is a library for packet processing. Currently the whole ovswitch is running as one process, which means it can not represent the CPU value of individual switches. However, the CPU utilization per switch is important because it shows the load of that switch, which could imply issues if it's abnormally high.

For the inspection module, it will be more useful if it can be improved to be lighter so that it can be more scalable. Instead of having multi-threads assigned per switch, it is possible to run a single-threaded process per switch. The problem with this idea is that by running only one process, the orders in which packets are processed in various interfaces may be different from the order in which they are arriving. Since the order is critical when determining retransmission and out-of-order, it is important to prove if the packets are handled in correct order.

References

[1] Openflow protocol: http://archive.openflow.org/
[2] Software-Defined Networking: Discover How to Save Money and Generate New Revenue http://www.cisco.com/c/en/us/solutions/collateral/service-provider/open-network-environment-service-providers/white-paper-c11-732672.html
[3] Ryu SDN controller: https://osrg.github.io/ryu/
[4] DPDK http://dpdk.org/
[5] Pidstat manual: http://sebastien.godard.pagesperso-orange.fr/man_pidstat.html
[6] TCPDUPM: http://www.tcpdump.org/
[7] D3: https://github.com/mbostock/d3/wiki
[8] C3: http://c3js.org/
[9] Async: https://github.com/caolan/async
[10] Jstat: https://github.com/jstat/jstat
[11] Ovs-vswitchd: http://manpages.ubuntu.com/manpages/natty/man8/ovs-vswitchd.8.html
[12] Wireshark: https://www.wireshark.org/
[13] Mininet: http://mininet.org/
[14] Open vSwitch: http://openvswitch.org/
[15] ofsoftswitch: http://cpqd.github.io/ofsoftswitch13/
[16] REST API: http://www.restapitutorial.com/