Monitoring Automata for Software Defined Networks

Masterarbeit
Shrikanth Malavalli Divakar
Tag der Einreichung: 29. March 2017
1. Gutachten: Prof. Patrick Eugster Ph.D
2. Gutachten: Patrick Jahnke

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Distributed System Programming Group

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 29. March 2017 Shrikanth Malavalli Divakar

Contents

List of Figures

Listings

List of Tables

List of Algorithms

1 Introduction

1.1 Scope for Software Defined Networking and Network Monitoring

In the most simplistic terms, Software Defined Networking is the physical separation of the network control plane from the forwarding plane such that the control plane controls several devices [ONFb]. There has been an ever-increasing expectation on Software Defined Networking (SDN) as a novel approach towards potentially replacing conventional traffic engineering, network management and control methodologies[GYG13].

SDN provides solutions to networking needs of present day's data centre networks to cope with the dynamically changing, resource intensive computation and a heavy demand for data storage. [ONFb] highlights the key computing trends such as the changing traffic patterns due to an increasing IT consumer base and the rise in cloud services to cater the increasing data consumption with implication on the need for more bandwidth. All the aforementioned factors foster the network engineers to innovate new techniques such that present-day computer networks could be made qualitatively flexible and quantitatively scalable. The most important part of the SDN is the controller which is used in the orchestration of different network environments such as carrier, enterprise, campus and clouds. Thus the SDN controller is the central entity that is technically termed as the control plane of the network and abstracts the underlying network elements from the network application services and providers. The traffic forwarding elements such as the routers and switches that enable connectivity with focus on the fast processing of network packets at line-rate complete the SDN architecture, forming the forwarding plane.

Recent efforts in opening up the forwarding plane of network elements for programming through various standardized APIs such as OpenFlow [ONFa] has made it possible to control the traffic forwarding behaviour and also monitor the network devices in unprecedented ways. The OpenFlow protocol[ONFa], an innovation pertaining to software defined networking, enables to express the traffic forwarding rules as flow rules that could be commissioned on the switches through network tasks running on an SDN Controller. Several efforts have been made to make use of the SDN Controller's capability to communicate with the underlying networking devices via OpenFlow protocol for network monitoring purposes. Since SDN provides a global state view of the entire network, it throws open an opportunity to improve the performance of the network through network monitoring strategies that could potentially leverage the global snapshot of the network available with the SDN controller. Network monitoring can be broadly defined as a system that constantly monitors the network health[EJR+00]. A non-exhaustive list of monitoring tasks includes observing the operational state of the network devices, examining the traffic characteristics and notifying critical events. However, to carry out the aforementioned tasks while utilizing the SDN controller's comprehensive knowledge of the network, it remains a matter of concern as to how to effectively utilize the CPU capacity and limited memory available on the network devices for monitoring purposes. For instance, present day SDN switches contain fast-path memory that supports OpenFlow rule entries in the range 2000 to 20000 [NSBT16][SCF+12]. In a typical data-center network, a top of the rack switch demands 78000 flow rules to handle the traffic forwarding. It implies that it is practically not possible to have fine grained flow-rules installed in the network device memory for monitoring purposes. To handle large traffic aggregates, coarse-grained flow rules are preferred. There are many research works that propose monitoring solutions to be implemented on the SDN controller to coordinate effective utilization of the device memory. The measurements carried out on the SDN controller would be delayed by a few seconds to minutes depending on the network latency to get the dataset from

the devices [JYR11]. Also, the CPU capacity of the switch is undermined and the distributed nature of the network devices is not effectively leveraged in computing the measurement results. This serves as a motivation for the thesis work and as part of the thesis, some of the compelling works that make use of various network monitoring mechanisms through the SDN controller have been discussed taking into consideration their applicability in real word scenarios, the complexity of the system and performance metrics and finally a foundation framework is created on the switch as a first step towards building a system that could leverage the SDN controller's global state view of the network and at the same time effectively make use of distributed nature of the network devices and their computation capabilities. The work enables the future algorithms to be designed to run on the network devices directly to produce quick and useful measurement results so that they become self-resilient.

1.2 Contributions of the Thesis

The main focus of the thesis work is on creating a mechanism from the ground up to commission monitoring tasks called as agents on bare-metal switches by taking advantage of the global network state available with an SDN Controller. They are given the term agents as they perform statistics polling and provide the same to various algorithms that would be running on the device itself. Another important contribution is a novel approach introduced to utilize the limited memory space available on fast-path of the network devices effectively for monitoring without hampering the forwarding behaviour and sharing the memory with more important and critical tasks. The work shall leverage the OpenNSL[Opeb] APIs exposed by the hardware vendor to program the fast-path behaviour. The thesis work specifically utilizes memory programmability APIs to install fine-grained flow rules and runs automaton capable of reacting to dynamically changing resource utilization scenarios on the network devices while maintaining an account of statistics on a number of traffic flows. The proposed concept aims to poll flow and port-level statistics from the ASIC in real-time by attaining certain levels of polling accuracy in time-domain and thus enabling the possibility to use the collected statistics by specialized monitoring algorithms that would run on the network devices. As part of the work, the polling mechanism is measured for the load it would cause on the CPU and the accuracy of the timers that dispatch event handlers for polling is verified. Also, the framework will create the necessary messaging mechanism to move the monitoring tasks with the state information such as packets/bytes accounted for a flow from one network device that experiences resource shortage to another network device that could continue monitoring the flows. Alongside OpenFlow there exist other protocols such as OVSDB[PD13] and NETCONF that are used in configuring the parameters that are essential for the operation and maintenance of the network elements. In this work, the NETCONF protocol has been employed in provisioning tasks specific to monitoring the traffic flows.

1.3 Outline

The structure of the thesis is as follows. The Chapter State of the Art introduces and expands on the foundational concepts of Software-defined networking in general with emphasis on the prevalent network monitoring mechanisms that are proposed and in use by the SDN community. Further, the Chapter Technical Background starts by recalling the fundamentals of SDN architecture and introduces the reader to typical architecture followed in the design of commodity switches. It also throws light on various tools and technologies that aid in programming SDN-capable devices. A comprehensive description of the design of the Monitoring agent application is introduced to the readers in the Chapter System Design. The implementation details covering the tools and techniques used are present in the Chapter Implementation. The Evaluation Chapter looks at the use-case based validations and performance details of monitoring agents under varying polling intervals. Finally, the Conclusion and Future work Chapter summarizes the work undertaken in the context of the thesis and presents future research directions.

2 State of the Art

This section discusses some of the interesting research works in the field of network monitoring using software defined networking paradigms. Below literature works were chosen to understand thoroughly existing monitoring methodologies, problems that each of the work tries to address and how effectively they put the SDN paradigms to use in achieving monitoring tasks. These works can be classified into two categories.A set of monitoring mechanisms utilizing OpenFlow messages to fetch the required statistics from the devices and then run monitoring tasks on the SDN controller. These works do not require any special instrumentation beyond OpenFlow capability on the network devices to perform the designated task (FlowSense, OpenWatch). The other set of research works focus predominantly on utilizing the network device infrastructure efficiently while using the SDN controller to co-ordinate the task of statistics collection(iSTAMP, OpenSketch and On-line Measurement of Large Traffic Aggregates on Commodity Switches).

2.0.1 FlowSense

FlowSense[YLZ+13] follows a passive monitoring approach inspired by the reactive capability of OpenFlow based SDN controller. The SDN controller is designed to be reactive in the sense that it installs necessary flow rules on network devices whenever it receives an OpenFlow PacketIn message from a device indicating that there exist no forwarding rule on the device's information base that could be used to route/switch the packets through appropriate port to the destination. FlowSense is primarily designed to monitor link utilization in OpenFlow enabled SDN networks. FlowSense utilizes OpenFlow's PacketIn and FlowRemoved control messages for monitoring with an aim to achieve accurate and scalable monitoring technique with minimal or no network overhead. As per the OpenFlow protocol, every flow rule installed on the switch for traffic forwarding or monitoring is associated with timers such as hard time-out and idle time-out. Hard time-out is applicable to keep the flow rule active for a specific period of time in the routing table. Idle time-out is used to figure out if there is any traffic forwarding activity at all on the switch over a period of time. As and when a FlowRemoved message arrives at the SDN controller, FlowSense adds a check point and compares the time stamp of the last matching traffic to determine the active flow duration, if it was hard time-out, whole duration will be considered as active duration. At every check point it calculates the link utilization of a flow by calculating the ratio of the active duration to no of bytes transferred.The work is inherently disadvantageous due to its heavy reliability on the OpenFlow FlowRemoved message. There will be delay in calculating link usage status in case of long lasting flows, for instance video streaming traffic flows. Also, when there are aggregated flow rule entries installed on the device to optimally handle routing of many flows with same prefix match, FlowSense will have to wait till all the flows of that aggregated flow rule entry to expire to calculate the link usage. However, link utilization is desirable if calculated consistently such that traffic engineering techniques such as shaping and policing [Sha] could be employed based on the observed utilization levels.Another downside of the approach is that the method works only in a reactive flow rule installation environment and would not work if the flow rules for packet forwarding are installed pro-actively by the controller using the prior information available about the traffic routes. The positive aspect of FlowSense is that it does not poll the network device for any statistics and hence does not add to any additional control traffic between the controller and the underlying network.

2.0.2 OpenWatch

OpenWatch[Zha13] is another work that highlights adaptive flow statistics aggregation for anomaly detection. In this work an adaptive sampling method is chosen to overcome the burden of control

traffic, meant for monitoring, on network infrastructure. It uses linear prediction technique on previously collected samples to estimate the future measurement granularity in multiple dimensions such as time and flow-space domains. In time domain,flow statistics collection is controlled using a predefined, adjustable time interval. The switches report the statistics at the end of each time interval. The time interval is adjusted, if required, such that the switches could report statistics records more frequently for the OpenWatch to react/detect any anomaly. In the flow-space domain, it probes switches to collect statistics of entire flow space based on the predictions made. The coarse grained statistics are collected until there is a deviation ,beyond the threshold, from the historical data and when there is a deviation, fine-grained statistics are collected by modifying the rules to be more specific, removing any wild-card masks as required. If there is no deviation, application may zoom-out the rule to reduce the network overhead. OpenWatch emphasizes the fact that location of the switch, chosen for monitoring, along the path to a destination plays an important role in detection of some of the anomalies such as heavy hitters. The capacity of individual switches is taken into consideration before placing a rule. Network devices which cover high number of flow rules or with higher load will be used at the last. However, what is not considered in this work is the fact that the switches might run out of memory resource required to maintain flow entries and associated statistics as the memory resources are shared between multiple and more important tasks such as maintaining forwarding information base, access control lists based on MAC and IP addresses and so on. It is desirable to detect such scenarios wherein a network device runs out of fast-path memory required for higher priority tasks so that the flow entries meant for monitoring could be vacated to create space. As part of the thesis, it will be demonstrated through a proof of concept implementation that when a flow entry, designated for monitoring, is removed in order to create space for other tasks or when the traffic flow is routed across a different path due to traffic engineering reasons, monitoring of such a flow could be resumed on a different device present on the same path to the destination. In contrast to the proposed work, OpenWatch runs at the controller end and distributes only the flow rules amongst devices and expects for statistics to be reported to predict the future measurement granularity. Although network overhead is reduced by adapting the frequency of reports received, OpenWatch still adds to the control traffic on the communication channels between the controller and the underlying devices.

2.0.3 iSTAMP

iSTAMP[MWCS14] proposes an SDN based measurement paradigm for fine grained traffic flow measurements considering the constraints on availability of monitoring resources. This mechanism builds on compressed sensing and on-line learning to develop a theoretical foundation that provides guidance in building an adaptive flow measurement framework. It is thus called as an intelligent Traffic (de)Aggregation and Measurement Paradigm (iSTAMP) in which TCAM entries on the network devices are partitioned into two parts.

- Fetch measurements for aggregated flow-rules rather than fine grained flow-rules.
- Identify the most important and informative flow-rules and fetch statistics for such flow rules by installing more specific rules.

It also proposes an efficient Multi-arm Bandit based algorithm to adaptively track and measure the most rewarding flows with the highest impact on the monitoring application performance to provide accurate per-flow measurements. These flows are marked as important or rewarding from monitoring perspective. iSTAMP relies on OpenFlow APIs to install flow rules on the devices and fetch the resultant statistics. One important take away from this work is that the memory is divided carefully between monitoring and packet forwarding such that the routing/switching behaviour is not affected. iSTAMP application is evaluated for performance by using a typical monitoring task called hierarchical heavy hitters detection. The application is set to find out the aggregated traffic flow (an IP subnet prefix with

wild-cards representing a group of IP addresses) responsible for a higher utilization of bandwidth over an interval.

The thesis work derives inspiration from iSTAMP's idea of dividing the memory space, termed as TCAM, exclusively for fine-grained monitoring. However, the idea is further enhanced with the ability to maintain the memory space dynamically by keeping track of the TCAM space usage limits on the network devices. It allows the TCAM space used for monitoring to be dynamically varying and hence makes it possible to share the TCAM effectively for various other tasks such as Access control lists based on MAC/IP and also with OpenFlow like forwarding information base.

2.0.4 On-line Measurement of Large Traffic Aggregates on Commodity Switches

[JYR11]This is yet another work that focuses on utilizing the memory available on fast-path for the purpose of monitoring. This work also tries to solve the hierarchical heavy hitter detection problem. The framework utilizes simple match and count capability provided by the OpenFlow protocol to deploy monitoring rules on the network device. An important argument made as part of this work is that the monitoring activities when carried out on the devices themselves will be more beneficial than the tasks that are performed off-line by periodically polling the devices for various statistics. The framework restricts the role of the controller to only adapt the flow rules for monitoring periodically.
This work makes assumptions on number of TCAM entries available for monitoring purpose on data plane i.e. every switch is assumed to have a limit 'N' on the 'number of rules' to be used for traffic monitoring due to limited amount of TCAM availability. In the control plane, measurement framework relies only on features available in OpenFlow switches. To reduce the controller overhead, framework does not allow the switches to direct data packets to the controller; relegating the controller to installing only flow rules and reading counters. It is also assumed that controller only adapts rules at fixed intervals although controllers can read different counters at different times to spread the load over the interval. Evaluation results of this work revealed that 20 rules are sufficient to identify 88-94% of the 10%HHHes(i.e. IP prefixes utilizing 10% of the total bandwidth over a time interval 'T').
The main drawback of this approach is that the burden is on the controller to read the counters periodically to infer network health. The design suffers from the issue that the controller is a potential point of failure. It also requires, like other controller centric approaches, a lot of state information pertaining to the monitoring activities on each device to be maintained at th controller side. The polling related traffic between the switch and the controller could be reduced by using asynchronous notifications from the devices at pre-defined intervals to provide the status of the statistics to the SDN controller.

2.0.5 OpenSketch

OpenSketch's[YJM13] objective is to create a new set of APIs ,exclusively for measurement purposes, other than the commonly available functionalities of OpenFLow which are not specifically designed to help network monitoring goals. This work highlights the main challenge in designing new measurement API as to support a large spectrum of measurement activities while being efficient to use. The OpenSketch divides the data plane further into measurement data plane showcasing the fact that measurement activities could be performed by utilizing a part of the hardware infrastructure available on the network devices themselves. The control plane is used to commission the measurement data plane with necessary resources such as flow rules, allocation of memory and so on to perform monitoring. Disadvantages of NetFlow [RFC] based polling has been discussed as it leads to high CPU utilization, additional traffic towards a collector and requirement of a specialized system such as a collector that could analyse the thus obtained statistics data. OpenSketch emphasizes that there should be a dynamic measurement data collection framework that provides guarantees on the measurement accuracy. The OpenSketch relies on a customized FPGA data plane designed to achieve the tasks which are not generally available features

on large scale production ready bare metal switches. For instance, OpenSketch requires implementation of a few Hash functions in the data plane.It is an expensive idea to be adopted due to the high costs involved in redesigning the hardware. The OpenSketch also relies on the algorithms that run on the controller side to analyse the collected statistics.

The proposed concept in this thesis derives from the idea that providing guarantees on the accuracy of statistics polling is very important for time critical measurement tasks such as heavy hitter detection and link utilization. The thesis builds on this idea to design a monitoring framework that is configurable from the SDN controller to poll statistics at different time interval granularity on the device itself and only report the statistics if and only if expected by the controller which it can express while configuring th framework. Further, the work focuses on building a monitoring framework that is capable of detecting the fast-path memory utilization levels and make appropriate decisions such as offloading a few flow monitoring tasks to some other network device with the help of the SDN controller.

3 Technical Background

3.1 Fundamentals of Software Defined Networking

The Open Networking Foundation, a group behind the standardization of SDN and allied technologies defines Software Defined Networking as, "An emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of present day applications."[ONFb].

Figure 3.1 depicts the typical SDN architecture with its multiple layers distinguishing the Application, Control-plane and Data-plane functionalities. Network applications and services typically function in the Application layer providing various functionalities such as conventional Telecom central office network functions, application services related to wireless telecommunication such as Billing, Session management, Mobility management and so on. Applications make use of the underlying physical network for exchanging data in addition to imposing constraints on the network to achieve guarantees w.r.t time and desired quality of services. The applications at this layer are typically consumers of services provided by the control-plane. In general, Application layer entities are enabled to program the SDN Control-plane through a Northbound interface using RESTful APIs over HTTPS.

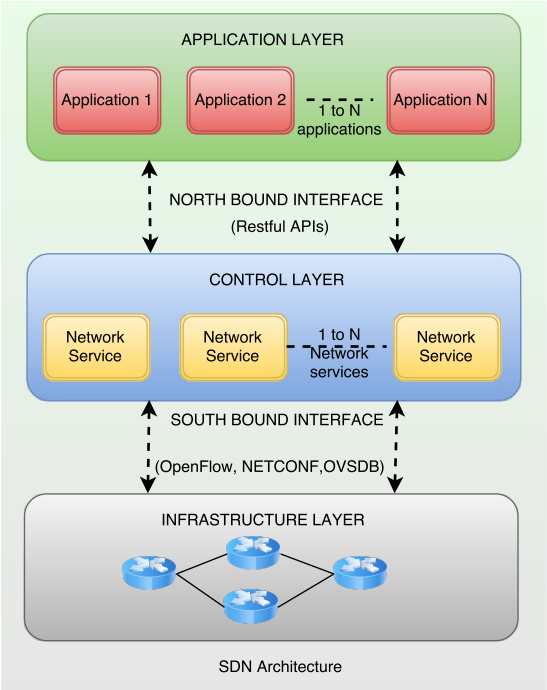


Figure 3.1: A simple SDN architecture

The SDN Control-plane is a logically centralized entity termed as the Controller that orchestrates the underlying network with desired parameters including but not limited to traffic engineering, Quality of Services and hence facilitates traffic management.The controller maintains a global network view visualized as a single logical switch to the Application layer programs and hence simplifies the programmability of the network.

The SDN Forwarding-plane is comprised of several entities such as virtualized network switches and

commodity hardware components.Open-vSwitch is one such instance of a virtualized network data-plane that could be programmed by the SDN Controller on similar lines as it could be done for a commodity hardware..More concepts related to commodity hardware components, their architecture and programmability has been discussed in the later section of the thesis.The SDN Control-plane programs the Forwarding-plane via South bound APIs, a term coined by the SDN standardization group. The OpenFlow protocol happens to be the foundational paradigm on the south bound interface that is in use extensively to program the Forwarding-plane. Besides OpenFlow, there exist other protocols such as NETCONF and OVSDB that could be put to use in commissioning the network devices before they could be used for traffic forwarding purpose.

3.2 Introduction to virtual switches and physical switches

3.2.1 Virtual software switches

A virtual switch is essentially a software application that enables communication in virtualized environments where multiple virtual machines could be coordinating and it could also be used to inter-connect physical nodes that do not require high bandwidth requirements.A virtual switch forwards the packets using the logic implemented completely as a software solution.Besides forwarding the packets, a virtual switch is also capable of performing deep packet inspection to carry out traffic shaping, policing and packet header manipulation.A virtual software switch is limited by its ability to switch the packets at line rate.Hence, it is mostly usable in switching the VM to VM traffic that rarely needs to hit the physical wire speed.However, there have been improvements made;for instance, to accelerate the packet processing on software switches running on Intel platforms, there have been technological changes proposed as part of Intel's DPDK[INT] that works on the principle of copying the packets that arrive on a Network interface card directly onto user space to be consumed by the application waiting for the packets.Performance metrics of such an approach has been detailed out and discussed in the work [ERWC14]

3.2.2 Introduction to physical switches and commodity switches

A physical switch is a network appliance specially designed to perform traffic switching/routing tasks.All the routers and switches in traditional networking world fall under this category.Their functionality is embedded into an Application Specific Integrated Circuit which is a hardware component designed to provide network functions.While the ASICs carry-out packet forwarding activities at wire speed, the essential tasks such as their configuration, maintaining the protocol state information and forwarding information base that is required for traffic management and so on are offloaded to a specialized application running on a conventional CPU.Such an application is coined the term Control-plane.Traditionally, the control plane of a network appliance resided on the same chassis that housed the ASIC.

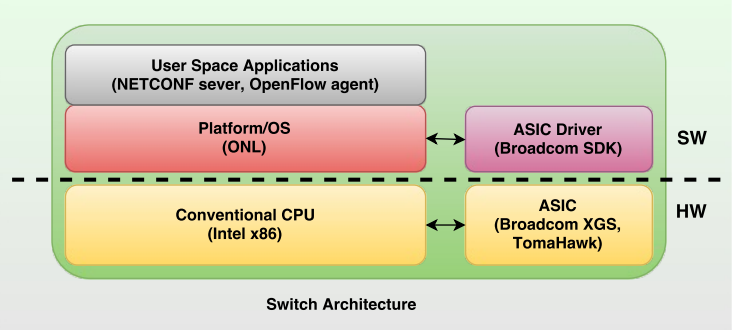


Figure 3.2: Physical switch architecture

In Software Defined networking, as discussed earlier, control plane is decoupled from the network appliance and resides on a separate entity called the SDN Controller.However, there also exist a conventional CPU on such appliances which is predominantly used to relay the information between the controller and the ASIC.For instance, ASIC requests for routing information from the SDN Controller via the in-house CPU.Like-wise, SDN Controller commissions the ASIC with the help of the conventional CPU present with the ASIC.A commodity switch is essentially a physical switch that is composed of off-the-shelf components provided by different vendors and can be assembled into a desirable configuration.Figure 3.2 gives pictorial representation of a commodity hardware that is built from off-the-shelf components.

3.3 NETCONF Protocol

This section provides an overview of the NETCONF[EBBS11] protocol which happens to be the foundation stone of the thesis work as it builds upon the idea of automated configuration of network devices.Information pertaining to protocol messages and modelling of data exchanged between the NETCONF peers is discussed.The thesis builds on the NETCONF protocol to automatically commission the monitoring agents and hence it is critical to understand the protocol as such.

NETCONF is defined in IETF standard RFC6241.It is primarily a network management protocol that makes use of XML encoding in Remote Procedure Call based messages to exchange configuration data and device information.

The architecture components of NETCONF based system are as listed below:

- NETCONF Server.
- NETCONF Clients or Managers and
- Data models.

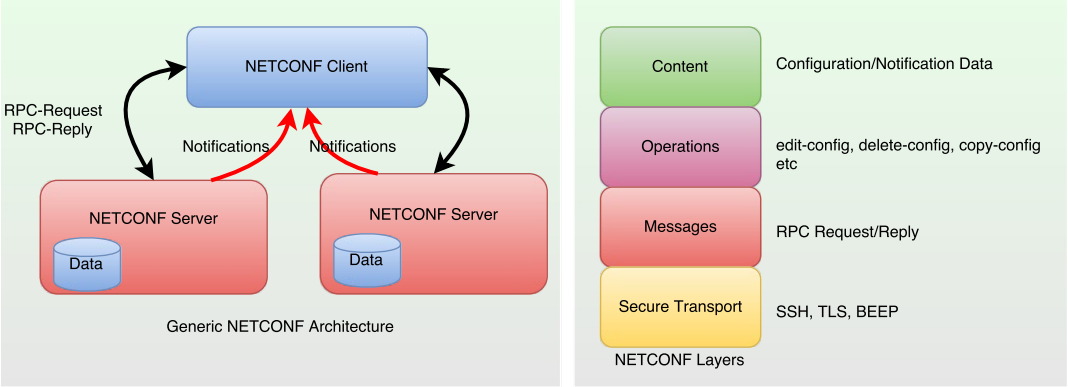


Figure 3.3: NETCONF protcol layers and components

NETCONF Server typically runs on the target device that should be configured.NETCONF Manger/-Client module runs as a centralized entity. This module commissions the required devices via NETCONF Server component located on respective devices.Lastly, NETCONF Data models provide the necessary data definitions for configuration and management information to be exchanged between NETCONF Servers and Clients.

As per the NETCONF standards, 4 conceptual layers of the protocol are as depicted in the Figure 3.3. Transport layer provides the necessary communication mechanism between NETCONF Server and Client components. All the NETCONF implementations adhere to the standard by supporting SSH as a default transport mechanism. The Message layer provides the RPC based XML encoded messaging

paradigm for data communication. Following are the RPC message frame elements used by the protocol semantics.

- RPC:<rpc> element is used to form RPC requests to be sent to a NETCONF peer.
- RPC-Reply:<rpc-reply> element is used to send replies that contain some data to be returned to the requester.
- RPC-Error:<rpc-error> element is used to indicate failures in RPC request operations.
- RPC-OK:<ok> element is returned as a reply to RPC requests if there is no data to be returned to the requester.

The RPC request messages are processed in the same order as they are received at the device. The Operations layer defines the set of operations that could be invoked on the target device through RPC messages. NETCONF base protocol provides the operations mentioned in the table 3.1 for data manipulation on the device.

Table 3.1: NETCONF 1.1 base protocol operations

Protocol Operation	Description
edit-config	Used to apply configuration settings to a Netconf managed device.
get-config	Used to retrieve the configuration settings from the data-store.
get	Used to get the operational status of the device not defined as configuration data in the data model.
copy-config	Used to copy the contents of one data store to another.
delete-config	Used to drop the data store that maintains the configuration settings on the device.

The thesis work banks upon the "edit-config" operation of the NETCONF protocol. The operation is specified as an xml element tag embedded within the RPC request message. <edit-config> element can have one of the values for the attribute "operation" mentioned in the table 3.2. Besides, the operation accepts several parameters such as "target","default-operation","test-option","error-option" and "config". All the parameters are embedded as xml child elements of the root <edit-config> element. The "target" parameter specifies the type of the data-store(for e.x: running or candidate) to be affected. The discussion pertaining to types of data-store is deferred until next paragraph. The "default-operation" parameter is similar to the "operation" attribute discussed in the table3.2 and carries the value "merge" if not explicitly specified. All the other parameters are optional and out of the scope of the thesis work as they are not used to model the configuration data for the monitoring application.

Table 3.2: edit-config operation and its attributes

Create operation's attribute	Description
Create	Create the configuration in the data-store
Merge	Merge the contents to the existing configuration in the data-store
Remove	Remove the configuration from the data-store
Replace	Completely replace the configuration indicated by the message.
Delete	Delete the configuration if exists, otherwise return error.

The content layer is not standardised by the NETCONF. Nevertheless, YANG[Bjo10] data modelling is a widely used technique for defining the content of NETCONF data and protocol operations.

The NETCONF protocol provides persistence of configuration data through its data-store mechanism. The data-store is essentially a place to store and access device configuration data. There are many ways to implement the data-store. A file-based data-store saves the configuration data in an XML file. It is also possible to use a sophisticated database implementation to store the configuration data besides the possibility to use the flash memory to maintain the data. The NETCONF protocol basically defines three different types of configuration data-stores.

- Running data-store: The running configuration data-store holds the complete configuration currently active on the network device[EBBS11].
- Candidate data-store: This data-store is used to hold configuration data that can be manipulated without impacting the device's current configuration[EBBS11].
- Startup data-store: The startup configuration is loaded by the device when it boots from this data-store if it exists. The content to this data-store is copied from the running data-store through an explicit <copy-config> operation.

.In this work, only the running-data store is used for commissioning the monitoring agents as the focus is on getting the monitoring agents running on the device in real-time. If the candidate data-store is used, additional time will be required to commit the contents to the running-data store which would delay the deployment of monitoring agents on the device.

The NETCONF protocol also defines the ability to send asynchronous notifications from the NETCONF Server components to the Clients that subscribe for notifications. There is a dedicated IETF standard that describes NETCONF notification[TC08] mechanism.

A NETCONF Manager/Client can receive notifications of interest by creating subscription. As per the protocol, the NETCONF server replies to the subscription request to indicate if the subscription is successfully registered. The <create-subscription> element is embedded in an RPC message to carry out a subscription operation. The Server will continue to send asynchronous notifications to the Client upon successful subscription registration. The event notifications will be sent to the Client as long as the session exists between the Server and the Client. Each notification will be sent as XML data enclosed within the <NOTIFICATION> XML root element. Only in the case of notifications, RPC XML element is not used and <NOTIFICATION> happens to be the root of the XML tree sent to the Client.

There can be several event streams created and used by the NETCONF Server. The information pertaining to the event streams can be obtained by querying the Server through <get> operation. From the perspective of the thesis work, a simple subscription to the default stream is sufficient as the monitoring application does not create any specific event stream to deliver monitoring notifications.

3.4 YANG Data model

YANG is a data modelling language that is used to model the contents of the NETCONF protocol messages exchanged. The data is basically arranged as a hierarchy of information elements which would ultimately be interpreted as an XML tree in an NETCONF RPC message. Each element is interpreted as node within the data model and a node may contain a value and one or more child nodes. YANG model also provides the ability to express constraints to be imposed on the managed data. For the purpose of defining the data model, YANG provides certain built-in data types. It is also possible to construct user-defined data types from the built-in types.

```
1  module monitoring-model {
2    namespace "http://monitoring-automata.net/sdn-mon-automata";
3    prefix "mon";
4    description
5    "Data model for Monitoring agents on bare metal switches.";
6
7    revision 2016-10-09 {
8    description
9    "Initial revision.";
10   }
11   /* Data definitions using various other constructs of the YANG language
       follow... */
12 }
```

Listing 3.1: Module definition using YANG

In the following paragraphs, various other YANG language constructs that have been used in modelling the network monitoring application are introduced.

For any new model to be defined, YANG data modelling begins with the module, sub-module and import constructs. A "module" construct provides an overview of the data model to be defined. It is the base unit of construction in YANG[Bjo10]. It provides a brief description of the model itself and the revision history to keep track of the changes. Finally it includes the data definitions that describe the data model in detail. The listing 3.1 has an excerpt from the module definition of the monitoring application designed and developed as part of the thesis work. It begins with the keyword "module" and encompasses a brief description about the module which is followed by the revision history. It also has a "namespace" definition that is used as an implicit prefix to all the XML elements of the "monitoring-model" when used in RPC messages.

```
grouping state-table {
  description
  "XFSM table for monitoring is represented as a table containing four
   columns and unlimited state transitions(Rows)";
  list state-table-row-entries{
    key "state";
    leaf state{
      type state-index;
      description
      "Holds the state entry for this column in the state table.";
    }
    /*other related leaf nodes follow here....*/
  }
}
container state-machine {
  leaf TotalStates{
    type uint32;
  }
  uses state-table;
  description
  "state-table is a group of related nodes";
}
```

Listing 3.2: YANG Data Definition statements

.
There are basically 4 different types of nodes for modelling the data.

- Leaf nodes: They are simple nodes as defined in listing 3.2 that act as a place holder for a basic data type such as an integer or a string. In the specified listing, a leaf node called "TotalStates" is defined which is a place holder for an unsigned integer type.

- Leaf-List Nodes: A list of leaf nodes where each node holds one value of a particular base type such as an integer or a string. This type of data definition construct is not used in the current work.

- Container Nodes: All the relevant leaf nodes can be grouped together to form container nodes. A container node is mainly composed of child nodes (that are leaf nodes) and assumes no explicit value. An excerpt of container node definition from the monitoring application is presented as part of the listing 3.2.The container "state-machine" has a leaf node called "TotalStates" defined. It also invokes a reusable grouping node which will be described later.

- List Nodes: A list is a sequence of structure or record entries. Each entry is comprised of several leaf nodes and one or more key leaf nodes used to uniquely identify the record entry within the list. The listing 3.2 has a list named "state-table-row-entries" with a leaf node called "state" designated as the "key" node that uniquely identifies each record within the list.

3.4.1 Grouping -A reusable data-modelling construct in YANG

There is yet another concept that provides more expressibility while defining the data model called the "grouping" construct. All the related nodes(container, lists, leaf and leaf-list) can be grouped together under the "grouping" construct making the thus formed group of nodes reusable. However, the construct

itself is not a data definition statement like other node definitions and hence does not explicitly include any data node in the resultant hierarchical data tree. A group thus formed using this construct is similar to a structure or any other user-defined type that is comprised of many basic types in a conventional programming language. The "uses" statement is used to invoke the "grouping" definition by mentioning the name of the group as an argument in the statement. For instance, "state-table" in the listing 3.2 is a grouping construct used to model the monitoring agent. The group consists of nodes that define the state table entries.

3.4.2 Custom RPC definition in YANG

As described in earlier section, all the NETCONF protocol messages are merely RPC messages. The NETCONF standard also allows to define custom RPC operations other than the ones defined as part of the base protocol. In order to support custom RPC functions, YANG has an "rpc" construct. The construct provides the semantics for custom RPC request-reply messages.

```
rpc mon_status{
  input{
    leaf mon-id{
      type uint32;
      mandatory true;
    }
    leaf device-id{
      type uint32;
    }
  }//end of input
  output {
    leaf mon-id{
      type uint32;
      mandatory true;
    }
/*Other leaf node definitions as required to return the data to the client
   */
  } //end of output
}//end of RPC
```

Listing 3.3: YANG Custom RPC operation definition

As depicted in the listing 3.3, a custom RPC operation is associated with a name that indicates the method to be invoked by the NETCONF peer when the RPC message is received. The "input" node present within the rpc node specifies any arguments supplied as arguments to the RPC call. The "output" node represents the data expected to be returned by the NETCONF peer in the RPC reply message. In a generic case, both the "input" and the "output" nodes are optional.If "output" node is ommitted, rpc-reply OK is returned to the NETCONF peer. The listing is an excerpt from the custom rpc operation defined to fetch the latest flow entry statistics maintained by the monitoring agent.

3.5 LIBNETCONF library

There are many implementations of NETCONF protocol such as "ncclient" and "netconf4android" that implement only Netconf client functionalities. Many other implementations of Netconf do not adhere to the latest RFC6241[Kre13]. Other than Libnetconf there is another open source implementation by YUMA works[Yum] that provides both the server and client side implementations. However, Libnetconf

is chosen for its simplicity and is readily available as a Netconf Server with the OFCONFIG[OO] implementation. The following section throws light on the internals of Libnetconf library which is critical to comprehend the enhancements made to the server.

As stated earlier, the Libnetconf[Kre13] library is an open source implementation of the Netconf protocol based on the RFC6241. It enables the developers to add Netconf functionality to any heterogeneous network.The library provides the necessary APIs required to manipulate the data-store, to carry out operations on the device based on the content of the data-store and also to handle custom RPC operations.

The Netconf protcol mandates SSH as a default transport protocol. The Libnetconf based Server does not implement any transport protocol by itself and relies on an external SSHD or TLS application to be interfaced with the Libnetconf based Server.The library reads the messages from the transport protocol application such as SSHD/TLS and writes back the response to its output interface file so that the SSHD/TLS application could read the message and forward the obtained Netconf message to the peer which happens to be a Netconf Client.

3.5.1 NETCONF Server architectures supported by LIBNETCONF

The LIBNETCONF library provides 3 different types of NETCONF Server architecture implementation.

- Single-level Architecture: In this type of implementation, Netconf server is implemented as a single process. SSH daemon starts the Netconf server process as its subsystem every-time there is a new Netconf Session connection establishment request. There is a downside to this approach as multiple instances of device manager (Netconf server) is required per connection. The device manager has to resolve the race conditions that arise while accessing the configuration data-store.

- Integrated Architecture: The design incorporates a single application that has the transport protocol embedded into it making the Netconf server bulkier. The advantage of this approach is that the Netconf Server will have full control over the Client sessions. Further, it also avoids inter-process communication that would be required in case of the Multi-level architecture.

- Multi-level Architecture: This type of implementation has a single instance of Netconf Server running as a system daemon. There will be one agent created per Netconf session to maintain the session which implies that it requires additional inter-process communication mechanism between the agents and the Netconf Server. Libnetconf library provides the necessary mechanism to serialise and de-serialise the Netconf messages and the Netconf Server's responsibility is to only take appropriate actions on the device and the configuration data-store as per the contents of the request messages.

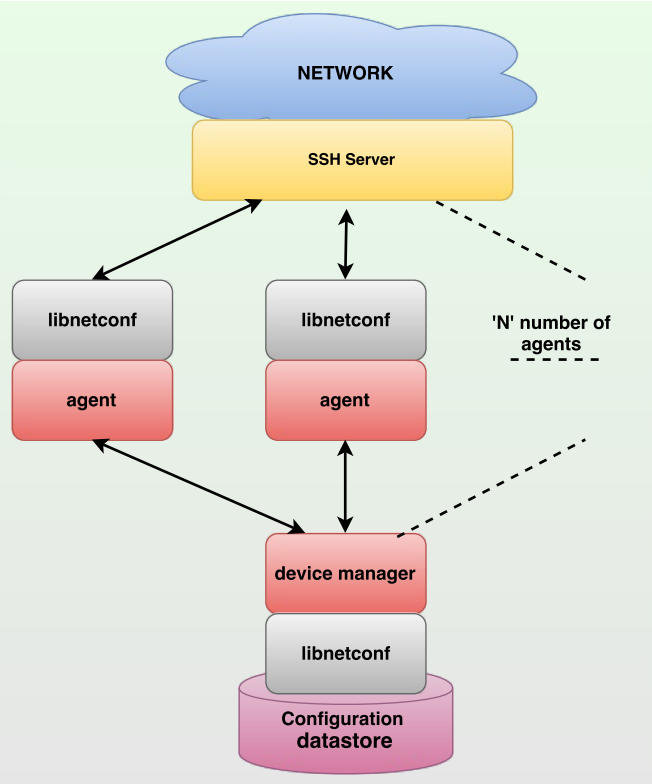


Figure 3.4: NETCONF Server Multi-level Architecture

source:https://rawgit.com/CESNET/libnetconf/master/doc/doxygen/html/da/db3/server.html#server-arch

The Figure 3.4 depicts such an architecture. The architecture fits well with the idea that an SDN capable switch could be managed by multiple SDN Controller instances for redundancy [BBMB16].

3.5.2 LIBNETCONF TransAPI module

The library exposes a set of APIs called TransAPI to connect the data-model to be used for device management with a custom add-on module that will be eventually running alongside the NETCONF Server. The add-on module is responsible for applying requested changes to the data-store as well as to the device(triggering the change in operational state of the device).

The TransAPI module works on an idea that every configuration data-model expressed in YANG will have pre-defined sensitive paths in its corresponding XML format. A sensitive path can be looked at as an XPATH expression that points to a particular XML element tag present in the XML hierarchy. When the NETCONF Server receives a request indicating changes to the settings represented by such an XML element, it will invoke the appropriate call-back function in the TransAPI add-on module to take necessary actions such as applying the requested changes to the device.

```
1 int callback_path_into_configuration_xml(void **data, XMLDIFF_OP op,
    xmlNodePtr old_node, xmlNodePtr new_node, struct nc_err **error)
```

Listing 3.4: A Typical TransAPI Callback Function signature

The listing 3.4 depicts an example TransAPI callback function provided with the LIBNETCONF developer guide[LIB]. In all cases, the suffix "path_into_configuration_xml" will be replaced by the absolute

path to the XML element that indicates the change in the configuration settings.The number and type of the arguments to the callback functions remain same irrespective of the XML element in the data-store the callback function is associated with.

3.5.3 RPC handling with LIBNETCONF

The library has a set of APIs to handle custom RPC requests.As discussed in the earlier section on YANG model, it is possible to define custom RPC operations that carry NETCONF compliant messages.The LIBNETCONF library has a way to associate the custom RPC operation with a function in the TransAPI module.It refers to the YANG definition and generates skeleton function corresponding to each RPC definition in the data-model and makes it available for custom implementation with the TransAPI module.To be able to send simple rpc replies with no data content, LIBNETCONF provides rpc_reply_ok() function and to include data content in the replies, it provides rpc_reply_data_ns(data,ns)(where 'data' is the XML content and 'ns' specified the name-space prefix for XML content).

All the other skeleton functions generated for a newly defined TransAPI module and how they are leveraged to fill meaningful code to carry out operations such as instantiating monitoring tasks on the device is explained in the implementation section of the thesis work.

3.6 A short notes on FAST-PATH memory on the Switch

The fundamental operation of a router/switch is to route/switch the traffic along the appropriate path through a particular port to the destination.To achieve this functionality, network devices rely on Ternary Content Addressable Memory as it provides Forwarding Information Base lookups within a single clock cycle[Jia13].In addition to FIB look up, TCAM also aids in pattern matching[ZCZ+15] and multi-field packet classification[Jia13].

A TCAM can hold three states 0, 1 and 'X' i.e. a don't care state.It generally stores ACL, QoS, IP, MAC and other information pertaining to packet processing.Most of the network devices contain multiple TCAM tables to carry out packet processing in both ingress and egress directions[CT].

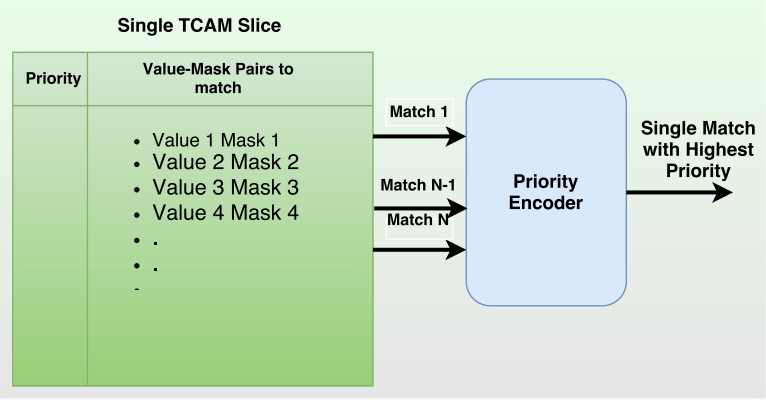


Figure 3.5: A pictorial representation of a TCAM slice

A slice is the smallest unit of TCAM allocation and such a slice generally contains varying number of rule entries depending on the platform.A TCAM's capacity is measured in terms of its width and depth.The width indicates the number of bits a TCAM entry can match[Jia13] and the depth indicates the total entries supported by a TCAM table[Jia13]. Various fields can be extracted from a received packet and used as input to the TCAM table lookup.The obtained fields are matched against value and

mask pairs present in the TCAM tables to get the matching result.The figure 3.5 presents an overview of the match-action functionality performed by a TCAM circuitry.For instance, consider the case that a field to be matched has the value 10101 and multiple rule entries such as 10XX1,10X01,10101 amongst many other rules exist in the TCAM slice.All the above mentioned rule entries would indicate TCAM entry hit as the bits marked 'X' match both a '0' or a '1'.In such a case, priority encoder will choose the entry with highest priority to be the match result obtained for the search field.It is worth noting that TCAM slices themselves are assigned certain priority within a system.For example, OpenNSL allows to create TCAM slices as "feature groups" with a user chosen priority or a default system given priority.A network device generally contains more than one TCAM slice(for instance IP based TCAM slice, MAC based TCAM slice, VLAN based or a TCAM slice with a match criteria that encompasses a combination of many fields and so on) with different fields chosen for matching criteria.It implies that, due to parallel search characteristics implemented within and across TCAM slices,a single query to match any received packet may match against multiple entries (each entry may have an associated action to be taken in case of a match) in each of the TCAM slices.In such a case, if at all there is any conflict in the consolidated actions from all the matched TCAM entries, highest priority entry from the highest priority TCAM slice will be considered and the action associated against the chosen entry will be performed.

3.6.1 Table Type Pattern based TCAM lookup in OpenFlow switches

This section introduces the concept of Table Type Pattern used in OpenFlow switches to effectively use the TCAM space while installing OpenFlow flow rule entries.This section is important to be able to get an understanding of the design decisions made while implementing the monitoring application.

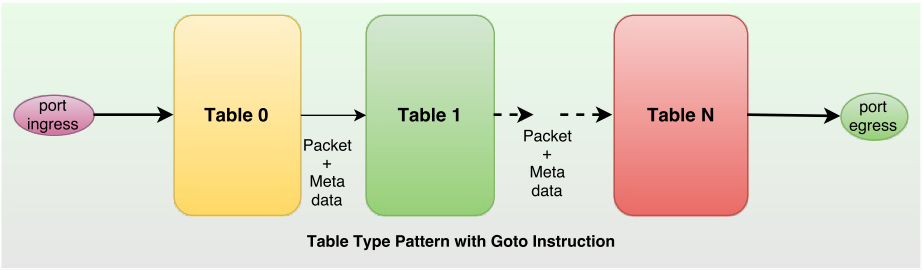


Figure 3.6: A pictorial representation of a TCAM pipeline approach

A Table Type Pattern (TTP) is an abstract switch model that describes specific forwarding behaviours that an SDN controller can program via the OpenFlow-Switch protocol[OT]. A TTP basically provides specification on a set of flow tables and describes the valid flow entries for an OpenFlow Logical Switch.The OpenFlow 1.0 version supported only single TCAM flow table programmability[MAB+08].However, later versions of OpenFlow introduced packet processing through a series of flow tables called TTP that match on packet headers. TTP allows access to other TCAM tables that were conventionally used as VLAN, MAC and IP address matching TCAM tables.The OpenFlow protocol provides "Go to" table instruction to perform lookup operations in such a table pattern.

The figure 3.6 depicts a TTP in action.A packet that enters the switch traverses through the TCAM tables.Each TCAM table is configured to match a packet header field or a combination of fields from the packet header and if there is a flow entry that matches the selected field(s), then the action associated with the entry is performed. Generally, when there is a series of tables, "Go to" table instruction is used to forward the packet to the next table.It will also add table meta-data information that could be used by the next TCAM table for processing the packet.If the matching flow-entry in a TCAM table does not direct the packet for further processing by another table, the processing ends in the current table and actions associated with the matched flow entry is executed.If a packet does not match any

entries present in a TCAM table, it will be considered as a table-miss.In such a case, a default flow entry may specify the behaviours such as forward the packet to the controller or to drop the packet and so on.

OpenFlow supports both the Hybrid pipeline as well as the OpenFlow only pipeline. The Hybrid model based packet processing is a combination of the legacy Ethernet based switching and the OpenFlow pipeline based processing.The standard leaves it open for implementations to decide the classification mechanism(either to use OpenFLow pipeline or legacy pipeline) used to process the packets in a hybrid model.The OpenNSL library that is used as part of the thesis work enables hybrid model by classifying the packets based on the port type a packet is received on.More details are discussed pertaining to this topic in a dedicated section on OpenNSL.

The chosen bare-metal switch for the thesis work has an ASIC from the vendor that supports OpenFlow pipeline based TCAM lookups as well as the conventional L2/L3 switching.(However, due to the reasons that will be discussed in the System Design section of the work,conventional pipeline is chosen.The design decision is also driven by the fact that the ASIC driver library in the hybrid model allows only VLAN and MAC based packet header match criterion if OpenFlow pipeline is used.The monitoring application necessitates that the packets are matched against the source and destination IP header fields also).Nevertheless, the concept behind the monitoring application is generically applicable to all the bare-metal switches that allow TCAM flow-rule programmability with the ability to match the packet header fields to identify flows based on source-destination IP address pairs.

3.7 OpenNSL

The OpenNSL library is an open source network switching library made available by Broadcom to program their StrataXGS[Str] architecture based ASICs that are in use on various bare-metal switches that fit well with SDN programming principles.It provides a rich set of APIs to design network applications that perform several switching and routing tasks.Some of the most notable features opened up for programming the ASIC are as listed below[Opeb]:

- APIs to manage L2 forwarding database, L3 feature based routing and VLAN management.
- Port level and Switch level management functions.
- Enables network monitoring activities such as link monitoring and statistics collection.
- Packet transmit and receive from the user space applications.
- Buffer statistics Tracking- Used to monitor the device level and port level packet buffer usage to dynamically fine-tune the buffer allocation.
- A group of APIs called as field processor APIs to manage memory(TCAM) and to harness the memory for packet processing on the ASIC.
- A set of APIs to manage IPv4/IPv6 tunnelling and link aggregation(trunking) features.
- It also provides Kernel Network Configuration APIs that can be used to send and receive packets through the native virtual network interfaces of the operating system which in turn would invoke OpenNSL packet transmit and receive functions to interact with the ASIC.
- Renders traffic engineering capabilites to the network applications via various QoS and policing APIs.

The OpenNSL visualizes the network as a collection of devices that are inter-connected and that contain several ports.The OpenNSL enabled functions are applied to each device specifically by the network applications to configure the device as well as to program the fast-path.[Opea].

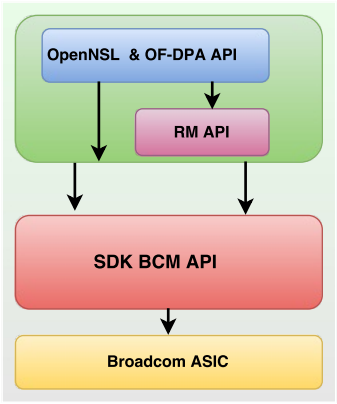


Figure 3.7: The OpenNSL framework

The figure 3.7 shows various modules of the OpenNSL framework. The SDK BCM API module is the Broadcom Software Development Kit which is essentially the driver software that interacts with the underlying ASIC. It is not Open to the network software development community for any modifications and it typically runs as a kernel module within the operating system. The OpenNSL module itself runs on top of the SDK module just described and exposes a set of APIs to achieve aforementioned features. The OpenNSL module encompasses a subset of APIs called OpenFlow Data-plane Abstraction that could be used to program OpenFlow packet processing pipeline. The OF-DPA APIs are made available as a standalone distribution also that comes without the other features of the OpenNSL library listed above. These set of libraries are compliant with the OpenFlow 1.3.4[OD] specification. When the OF-DPA API module collocated(as depicted in the figure) with the OpenNSL module is used, it enables hybrid model described by OpenFlow and lets the programmers to take control of flow-rule installation into INGRESS PORT,VLAN,MAC,BRIDGING and ACL policy TCAM tables. However, only VLAN and BRIDGING tables allow flow installation with the latest library distribution in hybrid mode.

```
typedef struct ofdpaFlowEntry_s
{
  OFDPA_FLOW_TABLE_ID_t       tableId;
  uint32_t                    priority;
  union
  {
    ofdpaVlanFlowEntry_t      vlanFlowEntry;
    ofdpaBridgingFlowEntry_t    bridgingFlowEntry;
  } flowData;
  uint32_t    hard_time;
  uint32_t    idle_time;
  uint64_t    cookie;
}ofdpaFlowEntry_t;
```

Listing 3.5: OF-DPA enabled flow tables with OpenNSL library

The listing 3.5 shows the parameters required to add flows into the OpenFlow based flow tables available with OpenNSL. The VLAN flow entry is used to install the flow rules into the VLAN match table while the bridging table matches the destination MAC addresses in the received packets. The OF-DPA only distribution[OD] provides a wide range of TCAM table management capability i.e ability to match all the packet header fields specified by the OpenFlow 1.3.4.

Lastly, RM module depicted in the figure 3.7 exposes a set of resource management APIs. This is used to intelligently divide the switch resources into several logical switches. The module mediates the access to hardware resources which enables network virtualization[CB10].

3.7.1 FAST-PATH memory programmability through OpenNSL

This section throws light on the APIs available to program the TCAM tables on Broadcom ASIC. As mentioned earlier, the OpenNSL library has Field Processor APIs that could be put to use as matching criterion while processing the packet. It allows to classify packets based on layer 2 to layer 7 packet header information as well as user-defined offsets in the payload. The APIs also allow to associate actions with the TCAM flow rule entries which would be performed when the packet header satisfies the match criteria defined for the flow rule. The TCAM table entries are termed as Field Processor entries by OpenNSL. The TCAM table itself is coined the term Field processor group which would contain all the entries with same match criteria. The group of attributes used to define the match criteria is termed as a QSET or a Qualifier set. A QSET may have one or more attributes (packet header fields or the meta-data such as ingress port information) in it as per the requirement.

```
1 opennsl_field_qset_t fp_mon_stats_qset;
2 opennsl_field_group_t fp_mon_stats_grps;
3 OPENNSL_FIELD_QSET_INIT(fp_mon_stats_qset);
4 OPENNSL_FIELD_QSET_ADD (fp_mon_stats_qset, opennslFieldQualifyStageIngress)
5 OPENNSL_FIELD_QSET_ADD (fp_mon_stats_qset, opennslFieldQualifySrcIp);
6 OPENNSL_FIELD_QSET_ADD (fp_mon_stats_qset, opennslFieldQualifyDstIp);
7 rc = opennsl_field_group_create(unit, fp_mon_stats_qset,0,&
    fp_mon_stats_grps);
```

Listing 3.6: TCAM table creation with Qualifier attributes

.
Generally a TCAM table could be created in various modes as mentioned below:

- Single width mode: Allocates one TCAM slice and rules in this mode consume one TCAM entry from the allocated slice.[Cis]
- Double width mode: It uses two TCAM slices and the rules consume one entry from each slice to fulfil the match criteria.This mode is used to expand the search key width of a TCAM table.[Net]
- Auto mode(default mode supported by OpenNSL):This is an auto expandable mode and the number of entries available for a TCAM table is generally not fixed. A slice can continue to grow if there is enough space to accommodate all the qualifiers used with the group.Generally the available space should be a multiple of 512 as per the existing industrial implementations.[Cis]

The OpenNSL library reserves a minimum size of 256 flow entries per TCAM slice if the TCAM table mode is not explicitly specified at the time of creation.Such a TCAM slice is auto expandable based on the memory availability on the system.

The listing 3.6 shows how a TCAM table, so called Feature Group, could be created. It requires that a Qualifier Set be created and initialized as shown in the lines numbered 1 and 3 respectively. The Qualifier set is used as the selector or the match criteria against the header fields of the received packets.In the listing, A Qualifier set to match the received packets against their Source and Destination IP addresses is created. A feature group could be created in one of the several stages of the packet

processing pipeline supported by the OpenNSL library.Generally, in any packet processing application, a processing pipeline is created by connecting the set of input ports with the set of output ports through a set of lookup tables [ID]. Likewise, a TCAM based table pipe-line could be created in the ASIC by harnessing the programmability provided by the OpenNSL APIs.The table 3.3 provides a snapshot of the various pipeline stages supported by OpenNSL . With the code mentioned in the listing 3.6, the TCAM table matching and the associated action is performed at the very early Ingress stage of the packet processing pipeline as indicated in the line number 4.

Table 3.3: TCAM table pipeline stage support in OpenNSL

Field Group Stage	Description
opennslFieldStageFirst	The earliest stage in the device a packet field is matched.
opennslFieldStageIngressEarly	Ingress stage is divided in to two stages. This is the ealiest Ingress stage.
opennslFieldStageIngressLate	The latest point of packet processing on the Ingress stage.
opennslFieldStageDefault	Default stage defined for the device.
opennslFieldStageLast	TCAM that is matched at the last stage of the pipeline.
opennslFieldStageIngress	To match packets at the Ingress and when there is only such stage in the pipeline. (Ingress Early and late stages are not used in that case).
opennslFieldStageEgress	To match the packets exiting the device through a designated egress port.
opennslFieldStageExternal	External Field Stage.
opennslFieldStageHash	hashing based TCAM table.
opennslFieldStageIngressExactMatch	Exact match TCAM table.

3.7.2 Associating actions and statistics with flow-rule entries

There are several options that could be associated as set of actions to a flow-rule entry within a feature group. It also allows to maintain statistics per flow-rule entry. The statistics object values linked to the flow-rule entry are modified every time the flow-rule entry is matched. The statistics could be collected over QoS parameters [TRT], number of packets and bytes received that match the criterion defined over L2 to L7 protocol fields as well as user defined offsets in the packet payload.

Table 3.4: Objects association with a Flow-rule entry in a TCAM table

Object	Description	APIs
Action. (opennsl_field_action_t)	Used to attach actions to flow-rule entries	opennsl_field_action_add opennsl_field_action_delete
Statistics. (opennsl_field_entry_t)	Used to attach/detach statistics objects to flow-rule entries	opennsl_field_entry_stat_attach opennsl_field_entry_stat_detach
Policing. (opennsl_policer_t)	Used to perform metering by attaching policing policies to flow-rules.	opennsl_field_entry_policer_attach opennsl_field_entry_policer_detach

The table 3.4 provides a non exhaustive (but most useful) list of objects that could be associated with a flow-rule entry.Firstly, action object is used to perform typical actions such as drop the packet, redirect the packet on a different Class of Service Queue [NBBB98][BBC06],mirror the packet through other ports,forward the packet to CPU for further processing and so on upon a flow-rule match.Secondly,statistics object association is done with the flow-rule entries using the APIs mentioned in the table.To rate limit the traffic, policing policies could be applied to a flow matching the rule using the policer attach and detach APIs.

The library lets the collection of statistics either by specifying an explicit "opennslFieldActionStat" action or by associating the statistics object described in the table 3.4 with the desired statistics parameters discussed earlier.

```
1 rc = opennsl_field_entry_create(unit, fp_mon_stats_grps, &entry);
2 rc = opennsl_field_stat_create(unit,fp_mon_stats_grps,2,lu_stat_ifp, &
     stat_id);
3 rc = opennsl_field_entry_stat_attach(unit, entry, stat_id);
4 rc = opennsl_field_entry_install(unit, entry);
```

Listing 3.7: Statistics entity creation and association with a flow-rule entry

The listing 3.7 depicts the series of steps involved in creation and association of a statistics object with a flow-rule entry. In the step 1, a flow-rule entry is created in the group specified by "fp_mon_stats_grps". The "entry" parameter is a return value from the API which uniquely identifies the flow-rule entry within the field-processor group/TCAM table. As a second step, a statistics object is created by specifying the number of statistics to account for (i.e. '2' in the listing just mentioned). The API returns a statistics ID that uniquely identifies the object.It should be noted that same object ID could be associated with more than one entry.In the third step the stad_id is attached to the flow-rule entry and as the last step flow-rule entry is installed into the field-processor group/TCAM table.

3.7.3 Value-Mask calculation for installation of flow-rule entries

The flow-rule installation necessitates that the qualifier set in accordance to the qualifiers defined for the field-processor group be specified during flow-entry installation. Each parameter that could be used as a qualifier has a corresponding API to include that parameter in the set. For instance, to include source and destination IP addresses as qualifiers, the APIs "opennsl_field_qualify_SrcIp" and "opennsl_field_qualify_DstIp" are used respectively. All the qualifying APIs expect the value-mask pair of a qualifier to be specified as an argument to the API. The "value" in the value-mask pair indicates to the actual value to be matched in the received packet and the mask is used to figure out the exact number of bits to be matched in the given "value".

To match the IP address field of a packet, the wild card mask is calculated as in the listing 3.8. Rule of thumb: To exactly match a bit in the specified value, the corresponding bit in the mask should be set to '1' and to ignore the bits in the value, the corresponding bits should be set to '0'. The listing 3.8 has details on the calculation of mask to match an IP header field in the received packet and the flow-rule entry. As it can be seen, the exact match case requires that all the bits are '1' in the mask whereas in the case 2, it requires that only the bits that should be matched in an IP address should have their corresponding bits set to '1' in the mask. Therefore, the mask used in the case 2 would match all the IP addresses in 50.0.0.0/8 subnet. The case 3 presents example mask calculation to match a range of IP addresses. It can be observed that only the last octet has its 0th and 1st bit changing in all the mentioned IP addresses. So, to match the given IP range, all the octets should be matched except the last octet. In the last octet, 0th and 1st bits are ignored and hence set to '0' in the mask which makes

the effective value of the octet as 'FC'.

```
/*Case 1:Exact Match*/
IP address to match:50.0.0.3 corresponds to 0x32000003
Mask:255.255.255.255  corresponds to 0xFFFFFFFF

/*Case 2:Match all the IP addresses with the subnet mask 55.0.0.0/8*/
Longest prefix match:50.0.0.0 corresponds to 0xFF000000
MASK:255.0.0.0 corresponds to 0xFF000000

/*Case 3:Match all IP addresses in the range 192.168.199.36 -
   192.168.199.39*/
IP address 192.168.199.36 -> 11000000.10101000.11000111.00100100
IP address 192.168.199.37 -> 11000000.10101000.11000111.00100101
IP address 192.168.199.38 -> 11000000.10101000.11000111.00100110
IP address 192.168.199.39 -> 11000000.10101000.11000111.00100111

/*Mask to match the range of IP addresses is calculated as below*/
/*IP Range => last octet has '0' set in its 0th and 1st bit position.*/
Range: 11000000.10101000.11000111.00100100
Mask : 11111111.11111111.11111111.11111100 => 255.255.255.12 => 0xFFFFFFFC
```

Listing 3.8: IP Address mask Calculation

3.7.4 TCAM table operations

The TCAM table allows a set of operations to be performed over the flow-rule entries other than the association of actions and statistics objects with the entries.

- Flow-entry install: The operation adds a new flow-rule entry to the TCAM table provided the qualifiers used in the flow-rule entry satisfies the qualifier set defined for a field-processor group. Before performing this operation an entry object should be created using the API opennsl_field_entry_create which returns an opaque handle that uniquely identifies the flow-entry in a field-processor group.

- Flow-entry remove: The operation destroys the object created for an entry and removes the flow-entry rule from the hardware tables.

- Flow-entry re-install: The operation is used to re-install a flow-rule entry in the hardware tables. It is used to modify the match criterion that would still satisfy the Qualifier set defined for a field-processor group.

- Flow-entry set-priority: The operation sets the priority of a flow-rule entry. The priority is unique across all the available hardware tables in the system. This feature is useful when one flow-entry should be preferred over the other for a particular flow when there are multiple flow-rule entries with similar matching characteristics.

- Flow-entry get-priority: The operation is useful to get the priority of a flow entry. Before setting the priority, a get operation could be used to check the priority, if at all already assigned.

- Flow-entry get-statistics: This operation is used to fetch the statistics attached with the flow-rule entry.

- Flow-entry get-all-entries: The operation fetches all the entries present in a field-processor group into an array.

- Check Field-processor group status: The operation is useful to check the current status of the hardware table. It provides information on the number of entries currently installed and the number of statistics objects associated. It would also include system wide resource availability.

3.7.5 Other Monitoring capabilities available with OpenNSL

Table 3.5: Other OpenNSL APIs for Monitoring

Monitoring Parameter	Description
Buffer Statistics	Provides APIs to monitor and track the packet buffer utilization and also to tune the allocation.
Class of Service Queue statistics	Exports APIs to keep an account of the packet reception characteristics per class of service queue associated with any port
Link Monitoring	APIs are available to receive notifications about the operational state of the links.
Packet Receive and Trasnmit	APIs to send and receive packets from an user-space application

The buffer statistics APIs could be used to track the packet buffer utilization at port levels as well as system level over a period of time. These statistics turn out to be useful in buffer tuning and various issues discussed in the work on Data-Centre TCP [AGM+10].

The packet receive and transmit APIs let an user space application to send and receive packets which could be utilized to implement latency calculation mechanism on the switch itself unlike the works.

3.8 ZeroMQ and CZMQ based message queue

The ZeroMQ library is an in-memory message queue implementation that could be used for communication between loosely coupled processes/components. The CZMQ [Akg13] library is a high-level wrapper library built around ZeroMQ for software components implemented in 'C'. The library provides asynchronous communication through sockets. Amongst many other features available with CZMQ, the event reactor loop implementation provides event dispatch mechanism such that the events such as message arrival on sockets and timer expiry could be handle gracefully without the need for interrupts.

4 System Design

In this chapter, system design and their alternatives that were considered before the final implementation are discussed.The section provides insight into the penultimate system design incorporated while building the monitoring application.

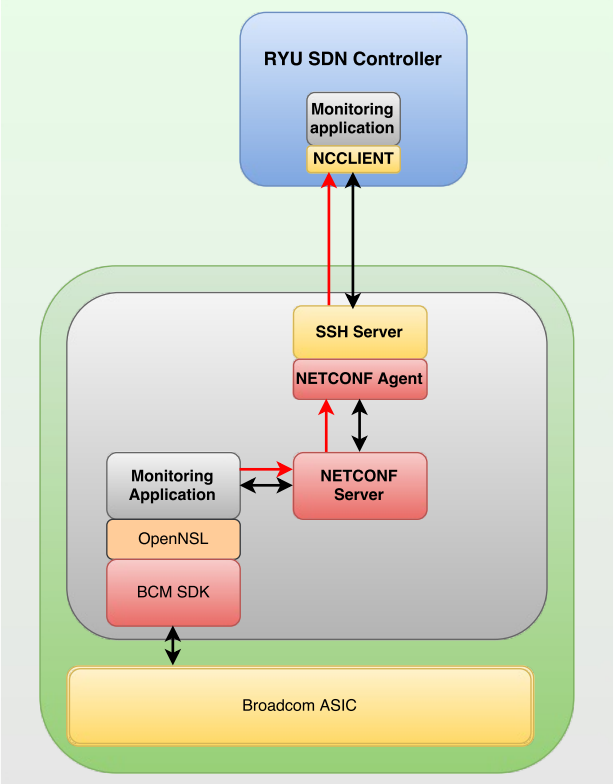


Figure 4.1: System Design

The figure 4.1 shows the complete system design incorporated for the development of Monitoring application on the bare-metal switches. The upper half of the figure depicts the RYU SDN Controller. The monitoring application is built-in to the SDN controller and makes use of the ncclient library [BTS09] (RYU's default choice to support Netconf) supplied python bindings for establishing Netconf communication. The lower half of the diagram is a representation of a switch running multiple processes on its control plane processor. The SSH server instantiates a Netconf agent to manage the Netconf session with the controller. The agent works in tandem with the SSH server as a subsystem as discussed in the section 3.5.1 on multi-level server architecture. The agent could communicate with the Netconf server either through unix domain sockets or using D-BUS messages as per the chosen option at the time of compilation of the Netconf server program (OF-CONFIG program discussed later). The Monitoring application runs as a separate process context on the CPU and leverages the OpenNSL APIs discussed in the section 3.7 to orchestrate the monitoring activities. The communication between the Netconf server and the monitoring application is enabled through ZeroMQ [Pie13] messages. The red arrow depicts the asynchronous notifications generated by the monitoring application and double-headed black arrow indicates the exchange of monitoring-agent configuration messages between the processes. More details on design choices and functionalities of each of the mentioned module follow in the next sections.

4.1 Southboud APIs: Netconf as the choice of protocol

The OpenFlow protocol is fundamentally designed as a paradigm to program the flow tables required for packet switching. Although, it is possible to add new protocol semantics to OpenFlow through its experimental extension headers, the protocol is not originally meant for network Monitoring. For carrying out network configuration management, Netconf has been widely accepted and adopted as a standard across the SDN controller implementations. The Netconf protocol itself is an alternative [dPFSEG15] to conventional SNMP [CFSD90] based Network monitoring and configuration.Besides, Netconf maintains a data-store with all the configuration settings pertaining to a monitoring agent on the device itself. This helps in mitigating the risk of SDN controller becoming the single point of failure. Secondly, Netconf,as discussed in the section 3.3 supports XML based message encoding. It implies that the expressibility that is required to configure the monitoring agents is available by default with the protocol.

4.2 SDN Controller: RYU as the choice

There are numerous implementations of SDN controller such as OpenDayLight, ONOS, RYU and so on with different levels of complexity and features built-in to their design [SEKC16]. The OpenFlow protocol is the de-facto communication paradigm supported by all the controller implementations along with the Netconf protocol. OpenDayLight, FloodLight and ONOS implement the latest Netconf standard [EBBS11]. The RYU controller leverages the Netconf client library implementation in python called "ncclient". To design the monitoring application, various Netconf operations have been harnessed. The RYU controller is simplistic in its design and follows a component based architecture. Due to its simplistic nature that would result in faster development cycle, the RYU controller has been chosen for implementation of monitoring application.

4.3 Multi-threaded vs Single process approach for the monitoring Application

The core requirement of the monitoring application being polling the port and flow-level statistics at the intervals defined by the controller, a multi-threaded design was initially considered such that each thread was expected to poll statistics for an indicated flow and provided the values to monitoring algorithm running within the thread. However, this design suffers from the scalability issue. If the number of monitoring threads exceed the total number of cores available on the CPU, it would result in computational overhead due to frequent context switches. Hence, only one thread that polls the statistics on behalf of all the monitoring-agents is preferred. It should be noted that the OpenNSL library internally spawns many threads to interact with the ASIC. Additionally, the ZeroMQ[Pie13] sockets library used to handle communication with the Netconf server uses background threads to perform asynchronous IO on the socket.

4.4 Netconf server architecture selection

The section 3.5.1 has already thrown light on various different Netconf server architecture styles possible with the Libnetconf library. The multi-level architecture is of great importance as it makes way to establish connections from different controllers to the same switch device. Firstly, it enables network virtualization concept as each controller can have a dedicated connection to the switch and share the hardware resources through OpenNSL APIs. However, network virtualization is not in the scope of the thesis work. Secondly, it enables redundancy i.e. allows hot or cold standby controller that would take over the network responsibility in the event of failure of the currently active controller. For the same reason, the monitoring application design is built around the said architecture. In the event of failure of an SDN Controller, the redundant controller will be able to fetch all the monitoring related configuration settings from

the device by establishing a new Netconf session and then executing the <get-config> Netconf operation.

4.5 OpenNSL Vs OF-DPA APIs

As already discussed, the broadcom switches are programmable using the vendor supplied APIs that come in two different types of distribution.

- OpenNSL library that supports Hybrid mode: The hybrid mode is already discussed in the section 3.6.1.
- OF-DPA distribution: The distribution only supports the OpenFlow pipeline programming.

The OpenNSL library allows to fetch Class of Service Queue statistics that would reflect the traffic load at queue level of each port and also across the control plane interface to the CPU. The traffic load across the interface could be used as an indicator of the CPU load level. Also, the OpenNSL library allows for various traffic engineering features such as redirecting the packets on a certain class of service queue which is a desirable feature to monitor flows that require bandwidth guarantees. The capability to direct the packets on a certain class of service Queue for ingress packet processing is not available with OpenFlow and it allows only the egress packets to be sent on a certain queue to support quality of service. The source-destination IP pairs based matching is critical for flow identification. Hence, the OF-DPA APIs available as part of the OpenNSL library could not be used as IP address based matching of packets with the contents of a TCAM table is not supported. Lastly and most importantly, to reduce the controller overhead, the mechanism similar to OpenFlow reactive packet processing that allows the switches to direct data packets to the controller (to get forwarding information) is not considered, relegating the controller to installing rules for monitoring purposes only and then to reading counters or to receiving event notifications that would be generated by the monitoring application. For packet forwarding, the forwarding information base (L2 and L3 address table) programmability available with the OpenNSL library is utilized. The devices are programmed to forward traffic from a subnet of IP addresses or to a specific destination host by default.

4.6 Monitoring messages and monitoring statistics

The table 4.1 presents monitoring messages designed for carrying out monitoring tasks. The first three messages are modelled as configuration settings in the data-store maintained by the Netconf server application. The remaining messages are designed as custom RPC operations.

Table 4.1: Monitoring messages

Monitoring messages	Description
MON_HELLO	To start a monitoring agent
MON_STOP	To stop a monitoring agent
MON_PARAM_CHANGE	To change the statistics polling interval of an agent
CONFIGURE_DEVICE_ID	To assign unique identifier to the devices.
GET_FIELD_PROCESSOR_GROUP_STATUS	Used to fetch the TCAM table status
SET_FP_ENTRY_THRESHOLD	Used to set a threshold on the TCAM memory
PORT_STATISTICS	Used to fetch the PORT statistics.
PORT_STATISTICS_CLEAR	Used to clear the PORT statistics.
MONITORING_STATUS	Used to get the ïňĆow statistics
CONFIGURE_TCAM_TIMER	Used to configure memory utilization check interval
MODIFY_EVENT_THRESHOLDS	To modify the flow statistics thresholds to detect events
ENABLE_DISABLE_ACTION	To dynamically enable or disable an action.

Table 4.2: Asynchronous notification messages

Notification Message	Description
MON_SWITCH	Notification to indicate that the configured TCAM threshold exceeded.
MON_EVENT_NOTIFICATION	Indicates if any port/flow events occurred for which the threshold was defined in the monitoring request.
MON_LINK_UTILIZATION	Notifies the statistics required to calculate the link utilization. (Number of unicast + non-unicast packets) and the number of packets received for a flow.

It is also possible to specify thresholds against these parameters when the monitoring agent is commissioned. It would be useful in detecting the change in traffic pattern based on certain predefined threshold. The monitoring application would notify the controller through an asynchronous notification enclosed in MON_EVENT_NOTIFICAITON message when the threshold exceeds. Along with flow statistics, port level statistics such as number of unicast packets and bytes received, dropped and transmitted, number of non-unicast packets that are received, dropped and transmitted could be obtained. On similar lines the data for broadcast packets and multi-cast packets could also be retrieved. The monitoring data model (present with the code repository) defined using YANG has clear and detailed explanation on the

statistics that could be retrieved.

The following table 4.3 provides the flow statistics parameters that could be retrieved from the ASIC through the monitoring application.

Table 4.3: Flow Statistics

Flow statistics parameters	Description
opennslFieldStatPackets	Number of packets accounted for the flow
opennslFieldStatBytes	Number of bytes accounted for the flow

The statistics object to fetch these counters is created when the monitoring agent is first commissioned on the switch through MON_HELLO message. More details follow in the implementation section which discusses on how a MON_HELLO message is handled in the application. Besides specifying the statistics required, the protocol semantics for monitoring also enables the ability to express the actions to be carried out such as notifying the controller of a certain event, sending notification after a time window and notifying the link utilization of a particular flow. The appendix to the thesis provides the sample XML encodings that encompass all the features of a monitoring agent.

5 Implementation

5.1 Implementation details of the Netconf server

The Open Networking Foundation suggested switch management protocol i.e. OpenFlow Configuration protocol is built around Netconf Server that follows the design details highlighted in the section 4.4. With a few modifications such as to completely disable the OF-CONFIG feature which is to configure and manage Open-Vswitch (hence it is not required by the presented work), the thesis work leverages the Libnetconf based Netconf server embedded within the OF-CONFIG implementation and enhances the server with an add-on module called TransAPI discussed in the section 3.5.2 and thereby manages and configures the network monitoring application designed to run on the bare-metal switch. The source-code that newly added/modified as part of the implementation is structured as below:

- server.c: The source file contains the Netconf server implementation and initializes the data-store required for the monitoring application.
- monitoring-model-transapi.c: The source file contains the TransAPI add-on module implementation that handles the monitoring application messages.

To begin with, as part of the existing implementation, the server initializes the Libnetconf library by calling the function nc_init() and initiates the communication with the agents. As a next step, the monitoring TransAPI module called "monitoring-model" is linked statically to the Netconf server through the ncds_new_transapi_static() API.

```
1  struct transapi mon_model_transapi = {
2    .version = 6,
3    .init = mon_model_transapi_init,
4    .close = mon_model_transapi_close,
5    .get_state = mon_model_get_state_data,
6    .clbks_order = TRANSAPI_CLBCKS_ORDER_DEFAULT,
7    .data_clbks = &mon_model_clbks,
8    .rpc_clbks = &mon_model_rpc_clbks,
9    .ns_mapping = mon_model_namespace_mapping,
10   .config_modified = &mon_model_config_modified,
11   .erropt = &mon_model_erropt,
12   .file_clbks = NULL,
13 };
```

Listing 5.1: Overall structure providing content of Monitoring module to the Libnetconf

The structure in the listing 5.1 is passed as an argument to the said function to provide the details on the call-back functions defined in the "monitoring-model" module. The Libnetconf library invokes the functions in the structure when it receives any Netconf message defined for monitoring data-model.

5.1.1 Monitoring-model TransAPI initialization and close

The Libnetconf library ensures that the initialization function "mon_model_transapi_init" is invoked before any other callbacks are called within the new module. This leaves scope to initialize the required data-structures to handle monitoring messages. A new thread (zmq_client) is created within this initialization function that listens on a socket to receive asynchronous notifications pertaining to monitoring

tasks from the monitoring application. A ZMQ_REQ socket is also created to send and receive RPC request-reply messages to the monitoring application on behalf of the SDN controller.

The "mon_model_transapi_close" does the reverse of the initialization function. It is invoked when the Netconf server is terminated. The function destroys the thread that was created for receiving notifications and also closes the RPC request-reply socket. When this function returns, the Netconf server will be terminated.

5.1.2 Data call-backs and RPC call-backs of monitoring data-model

The data-structure "struct transapi" in the listing 5.1 holds two member variables data_clbks and rpc_clbks that in-turn point to other data-structures mon_model_clbks and mon_model_rpc_clbks respectively. The functions in mon_model_clbks are invoked by the Libnetconf when some data-model operation indicating monitoring start, stop or polling interval change is requested by the SDN controller. The functions in mon_model_rpc_clbks are invoked when custom RPC operations are requested. The table 4.1 lists various messages in use by the monitoring application. First three messages affect the data-store represented by "datastore.xml" file and the rest of the messages listed trigger custom RPC operations indicated by the mon_model_rpc_clbks.

5.1.3 Handling MON_HELLO,MON_STOP and MON_PARAM_CHANGE messages

A call-back function similar to the one discussed in the section 3.5.2 is used to handle the monitoring messages. The Netconf server invokes the "callback_mon_monitoring_model_mon_monitoring_agent" function when it receives a Netconf message with XML content containing a hierarchy of elements encompassed by the <monitoring-model> element tag. For instance, a typical message to start monitoring i.e. MON_HELLO is depicted in the listing 8.1 under the section 8. As it could be seen in the listing, the <monitoring-model> element tag has a child element called <monitoring-agent> which contains all the configuration elements specific to a monitoring agent. To differentiate between the types of monitoring messages, the "XMLDIFF_OP op" parameter of the call-back function is used as below.

```
1  if( op & XMLDIFF_MOD )
2  {
3    msg_type = MON_PARAM_CHANGE; //Interpret this as a MON_PARAM_CHANGE
       message
4  }
5  else if ( op & XMLDIFF_ADD)
6  {
7    msg_type = MON_HELLO; //Interpret this as a MON_HELLO message
8  }
9  else if ( op & XMLDIFF_REM)
10 {
11   msg_type = MON_STOP; //Interpret this as a MON_STOP message
12 }
```

Listing 5.2: Detection of operations on the data-store

The "operation" attribute of the child element <monitoring-agent> specifies the type of action to be taken on the data-store. Accordingly, the Libnetconf library calls the above mentioned call-back function with parameters that reflect the changes done to the data-store. The "op" parameter is essentially a bitmap that indicates the kind of operation performed on the monitoring-model's data-store as per the request message. So, a bit-wise "AND" operation reveals the bit that is set in the bitmap. According to

the bit set, type of the monitoring message is deduced and the same is indicated in the message sent to the monitoring application process over RPC request socket. It should be noted that all the operations are synchronous which implies that the client requests are provided with time guarantees regarding the execution of a request.

5.1.4 Custom RPC requests to TransAPI module

The custom RPC operations introduced for monitoring tasks are as listed in the table 4.1. Each operation has a one-to-one mapping to a call-back function in the TransAPI module. The custom RPC operations are invoked when the RPC message's content matches with any of the RPC operations defined in the data-model for the module. The messages are essentially relayed onto the monitoring application awaiting its reply. Upon receiving reply, the same will be forwarded transparently to the SDN controller. The XML format of each of the custom RPC operation is as listed in the section 8, listing 8.7 through 8.5. All the XML elements are mandatory and Libnetconf library would fail an RPC request when a certain message has its content missing. The listing 5.3 provides a quick reference of RPC messages to call-back functions mappings.The "name" member variable corresponds to RPC message and the "func" member variable corresponds to the call-back function that would be invoked when the RPC message is received.

```
struct transapi_rpc_callbacks mon_model_rpc_clbks = {
  .callbacks_count = 10,
  .callbacks = {
  {.name="set-fp-entry-threshold", .func=rpc_set_fp_entry_threshold},
  {.name="set-field-priority", .func=rpc_set_field_priority},
  {.name="get-field-priority", .func=rpc_get_field_priority},
  {.name="configure-device-id", .func=rpc_configure_device_id},
  {.name="get_field_processor_group_status", .func=
   rpc_get_field_processor_group_status},
  {.name="get_bst_statistics", .func=rpc_get_bst_statistics},
  {.name="mon_stop", .func=rpc_mon_stop},
  {.name="mon_status", .func=rpc_mon_status},
  {.name="port_statistics_clear", .func=rpc_port_statistics_clear},
  {.name="port_statistics", .func=rpc_port_statistics}
  }
};
```

Listing 5.3: RPC messages to Call-back functions mapping

5.2 Implementation details of the Monitoring application

The monitoring application is implemented as a daemon process as discussed in the section 4.3. At the core of the monitoring application, the implementation follows the event-driven reactor pattern[Sch95] where an event is either the reception of the monitoring request messages from the SDN controller via the Netconf server or the expiration of timers associated with the monitoring agents. The main function creates necessary sockets to receive request messages from the Netconf server as well as to send notifications to the controller. It also initializes the OpenNSL driver and sets up the port configuration and installs default routes to handle traffic routing. A sophisticated, thread-safe logging library called "zlog" [Sim] is initialized in the main function. Finally, the application initializes and starts an event based reactor loop provided by the CZMQ [PH] library that invokes appropriate event handlers upon detecting events.

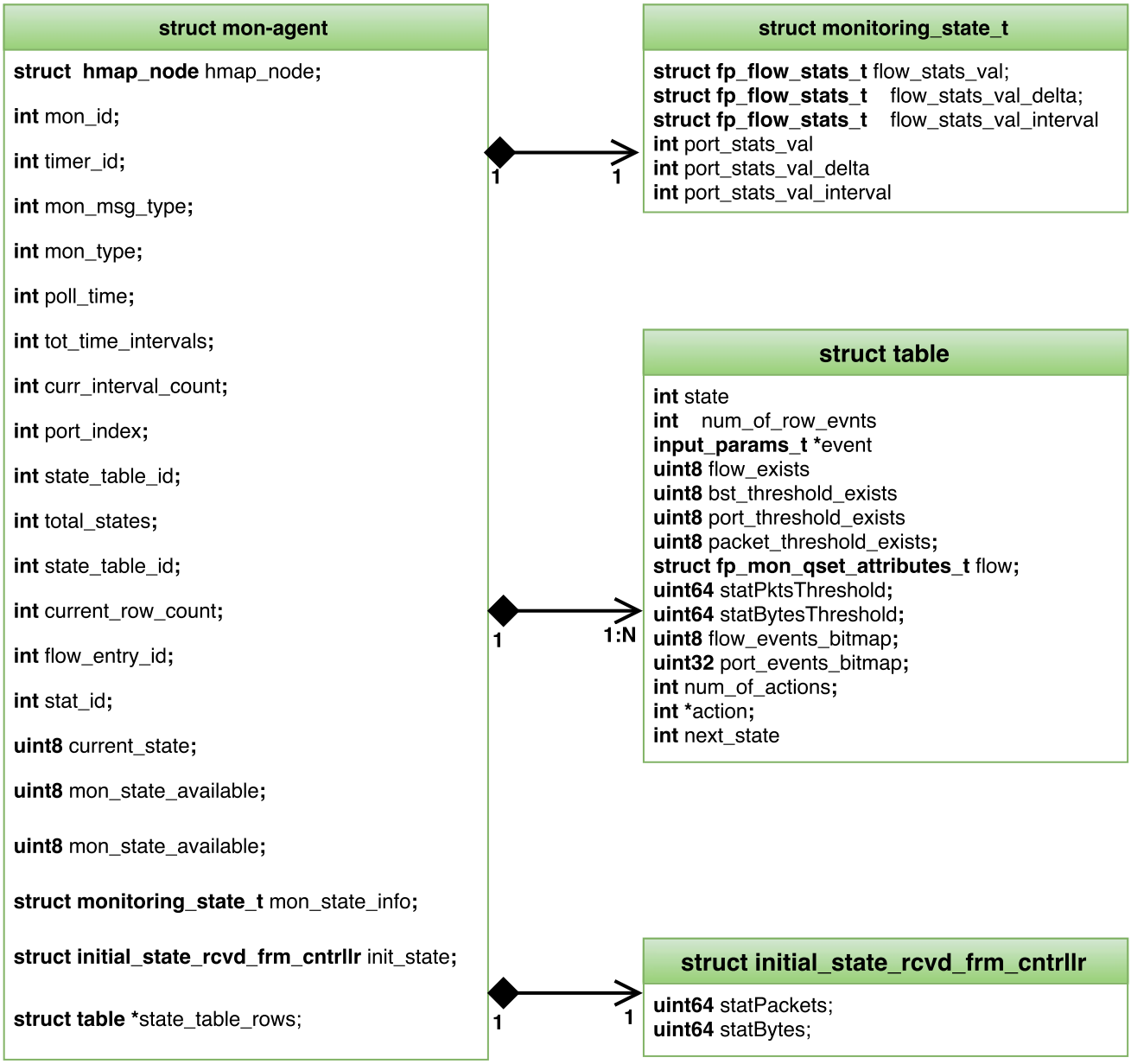


Figure 5.1: Monitoring Agent Class Diagram

The figure 5.1 presents the class diagram for the monitoring agent. A monitoring agent is essentially a hash-map node that maintains the statistics information pertaining to a flow or a port or both. The statistical information that each monitoring agent maintains is termed the state information pertaining to a monitoring task. Each agent is implemented to behave according to the configuration parameters received from the SDN controller. The SDN controller expresses the monitoring requirements for an agent as so called state table which would contain 5 tuples, S={s,i,a,f,n} in one or more rows. The state table is inspired from the work on OpenState[BBCC14]. However, there is not much similarity between the two works as OpenState is designed to control the flow behaviour. The 's' indicates the current state of the state-machine and uniquely identifies a flow-rule monitoring request (useful when there exist multiple flow rules in the state-table description), 'i' indicates various events that should be detected by the agent at the end of each polling interval. The set variable 'a' points to the set of actions an agent is expected to perform at the end of each/after 'N' polling interval and finally 'n' refers to the next state. A multi-row configuration table is feasible because of this implementation where each row would point to a different flow-rule monitoring requirement. Although, only one row would be sufficient

to hold the monitoring requirements for a single flow identified by source-destination IP pair, a multiple row state table is made feasible (as a place holder) such that the SDN controller could push the flow monitoring requirements for different flows as part of a single monitoring agent. For instance, if the SDN controller knows a set of IP addresses it should monitor for link utilization (but one after the other), then a multi-row monitoring agent could be used. It also enables to realize the idea similar to zoom-in and zoom-out capabilities in the Openwatch work discussed in the literature section. The class diagram shows the compositional relationship of a monitoring agent with the contained objects. The "table" structure would contain the configuration parameters obtained from the controller for a single flow. If an agent is configured to monitor multiple flows one after the other, where each flow should be monitored only for a defined interval of time before moving onto the next flow, then the "mon-agent" structure would typically point to multiple "table" objects corresponding to the number of flows to be monitored. However, in the current work, an agent is assigned with the task of monitoring only one flow which implies that several monitoring agents would exist in the hash-map. The "monitoring_state_t" data-structure would maintain the flow statistics obtained at the beginning and end of an interval which is used to calculate the effective counter values during that interval. When the monitoring agent is moved from a device to some other device, in order to satisfy TCAM memory threshold requirements, the obtained monitoring state information pertaining to the flow would be saved in the structure "initial_state_rcvd_from_cntrllr" on the new device. The state information thus obtained would be used to initialize planned monitoring algorithm to predict the future flow behaviour amongst others. however, such an algorithm is not considered in the scope of the current work. The current work only facilitates the transfer of monitoring state information from one device to another via the controller. The SDN controller could specify a list of parameters and their thresholds in the monitoring requests such that an agent could check if any of the specified parameter exceeded the threshold and report the same to the controller.

5.2.1 A summary of event-loop implementation using CZMQ

Algorithm 1 Event-driven loop enabled through CZMQ

```
1: procedure zloop_start                                              ▷ Reactor pattern
2:     while flag ≠ false do                          ▷ exit event loop if an error/interrupt occurs
3:         Traverse the timer list
4:         for all timer in timer_list do
5:             if timer_now ≥timer_when then
6:                 if timer_id =tcam_timer_id then
7:                     flag ← s_fp_group_status_check_timer_event();        ▷ TCAM-status event
8:                 else
9:                     flag ← s_timer_event();                          ▷ Invoke Timer event handler
10:                end if
11:            end if
12:        end for
13:        if poller has message from NetConf Socket then
14:            flag ← s_netconf_socket_event();                         ▷ Invoke Netconf event handler
15:        end if
16:    end while
17: end procedure
```

The pseudo-code 1 summarizes the event reactor loop provided by the CZMQ library from the monitoring application perspective. The zloop_start in real does not differentiate between the timer IDs. It would just check if any timer expired and then invoke the corresponding event handler registered at the time of

timer creation. However, for understanding how the monitoring application is designed and to make it easy to comprehend which event handler is invoked when,timer ID based differentiation is depicted in the pseudo-code. The event loop runs through the timer list that the library maintains, into and from which it adds/deletes timers upon creation and deletion respectively. While it traverses the list, it checks if any of the timers have expired and triggers the corresponding event handler registered. In the monitoring application case, the handler s_timer_event is a generic reactor handler for all the monitoring agents. The agents would fetch relevant statistics from the ASIC as specified in the monitoring request as part of the logic that goes into the handler function. Likewise, s_fp_group_status_check_timer_event function handler would be dispatched if the TCAM timer expires. Inside this handler, the system checks the TCAM utilization levels against the configured thresholds and triggers notifications to the SDN controller if the threshold exceeds.

5.2.2 Handling monitoring messages

Algorithm 2 NetConf Socket Event

```
1: procedure s_netconf_socket_event                ▷ handles messages from the SDN Controller
   msg ← Read message from netconf socket
2:     if (msg is not monitoring_messages) then
3:         handle Custom RPC messages to set device Id, TCAM threshold, fetch
4:         port statistics and monitoring status (flow statistics) here.
5:     end if
6:     if (msg = MON_HELLO) then
7:         mon_agent ← extract config from the msg
8:         hashmap ← insert(mon_agent);
9:         ret ← start_flow_stats_state_machine();          ▷ Install flow-rule into TCAM
10:        Start timer for this agent if no agent with similar poll requirement exist already
11:        timer_id ← zloop_timer();
12:    else if (msg = MON_STOP) then
13:        mon_id ← extract mon_id from the msg
14:        mon_agent ← monitoring_agent_mapping_find(mon_id);
15:        timer_id ← mon_agent.timer_id;
16:        mon_fp_stats_feature_entry_destroy(mon_agent -> entry_id);   ▷ Remove flow-rule
17:        hashmap ← monitoring_agent_mapping_destroy(mon_agent);
18:        zloop_timer_end(timer_id);                   ▷ Delete the timer for this agent
19:    else if (msg = MON_PARAM_CHANGE) then
20:        Change the poll-time for this monitoring agent
21:    end if
22: end procedure
```

The pseudocode 2 summarizes the events dispatched by the reactor loop when it detects messages on the socket dedicated for communication with the Netconf server. Firstly, the implementation checks for type of the message received. If the message is meant to configure the device identification number, TCAM threshold limits, to fetch the port statistics/to clear the port statistics or to fetch the flow statistics then it is handled in a function called check_msg_type. It sends the response to the SDN controller after performing the tasks requested and the handler returns to the reactor-loop. If the message is to start/stop the monitoring agent or to change the poll interval time, it is handled as shown in the if-else-if block of the pseudocode. The flow-rules are installed/un-installed into/from the TCAM tables through the APIs discussed in the section 3.7.1. The pseudocode also shows the insertion and deletion of monitoring agent objects from the hash-map depending on the type of the message. A sophisticated

implementation of hash-map is borrowed from the OVSLIB library of the Open-Vswitch distribution and the library is linked dynamically to the monitoring application.

5.2.3 Handling poll timer expiry events to fetch statistics

The poll timer is configured from the SDN controller for each agent. When the timer expires, "s_timer_event" handler fetches the port and flow statistics. There exist two different implementations of the event handler. In the first implementation, the monitoring agents with similar polling requirement are grouped together under a single timer and their flow-rules are polled from the TCAM to fetch the statistics. In the second implementation, each monitoring agent gets its event handler. The grouping is done when the MON_HELLO is received. The SDN controller is expected to configure the ingress port number on which a flow that should be monitored arrives. The statistics collected are used to calculate the effective value of the counters during that interval.

$$Counter\_val\_interval = (Counter\_val\_at\_delta - Counter\_val\_at\_start\_of\_interval). \tag{5.1}$$

The calculated interval counter is used to check if any parameters defined by the controller exceeded the threshold. This should be useful in identifying events on the switch such as transmission errors, packet drops, change in traffic pattern such as an increase in the number of packets received or number of bytes received within a flow and so on. The event handler function also checks for the actions to be taken and if the action is explicitly specified, it notifies the collected port and flow statistics to the controller. An agent could be assigned the three types of monitoring tasks:

- Flow Statistics: This type of task only monitors a flow identified by source-destination IP address pairs. The task is configured using the symbolic constant "FP" in the monitoring message.
- Port Statistics: This type of task pegs only the port level statistics.The task is configured using the symbolic constant "STATS" in the monitoring message.
- Port and Flow level Statistics: This type of task maintains statistics for both i.e. a given flow as well as the port on which the flow arrives. The task could be configured using "FP_STATS" symbolic constant in the monitoring message.

5.3 Monitoring application on the RYU Controller

As discussed in the section 4.2, a monitoring application is implemented on the controller side that takes advantage of the global network state available with the controller. The controller application essentially provides interfaces for other controller applications that are aware of network state to initiate and control monitoring activities. The application uses "ncclient" library. A set of new Netconf compatible custom RPC operations have been added to support the semantics of the network monitoring that the presented work tries to achieve. The APIs such as configuring the TCAM timer interval, modifying event thresholds and dynamically enabling or disabling actions are available only through netopeer client. However, the server side implementation supports all the APIs discussed in the system design section. The RYU SDN controller essentially wraps up the standard Netconf <edit-config> operation in a set of APIs to start, stop, for agent movement and to fetch details on flow monitoring. The <edit-config> element tag's attribute called "operation" is used to differentiate between the monitoring tasks such as start, stop and parameter change. A detailed discussion on various values the "operation" attribute could hold and the semantics of <edit-config> operation is already discussed in the section 3.3. Besides exposing various APIs based on Netconf protocol, The application spawns a thread that listens for asynchronous notifications from the switches. The thread also facilitates the movement of

monitoring tasks from one switch to another to maintain TCAM memory utilization levels on the switch.

The class diagram in the figure 5.2 provides an overview of the state information pertaining to monitoring maintained at the controller side and also shows the methods exposed to integrate the monitoring application with other SDN controller applications. The "MonCapableSwitch" represents the connection to a specific switch in the network and the class provides necessary wrappers around the Netconf operations. The table 4.1 already describes the custom RPC operations supported for monitoring purposes. The class "MonitorApplication" is the main class that creates and destroys the monitoring agents. It maintains a dictionary that maps the devices to their SSH sessions with the controller. It also maintains a list of Monitoring agent objects represented by the "MonAgent" class where each "MonAgent" object in turn maintains the state information such as the monitoring parameters configured on the switch. The other way to obtain the monitoring configuration parameters already running on the switches is discussed in the section 4.4. As shown in the class diagram, the "MonAgent" class provides the necessary interfaces to retrieve the monitoring parameters from the dictionary that it maintains. The "monitor_thread" member references the thread created by the monitoring application that facilitates the movement of monitoring agents. The new custom RPC operations required by the monitoring application are implemented as device operations in their respective classes that inherited from the super class "RPC" provided by the "ncclient" python library.

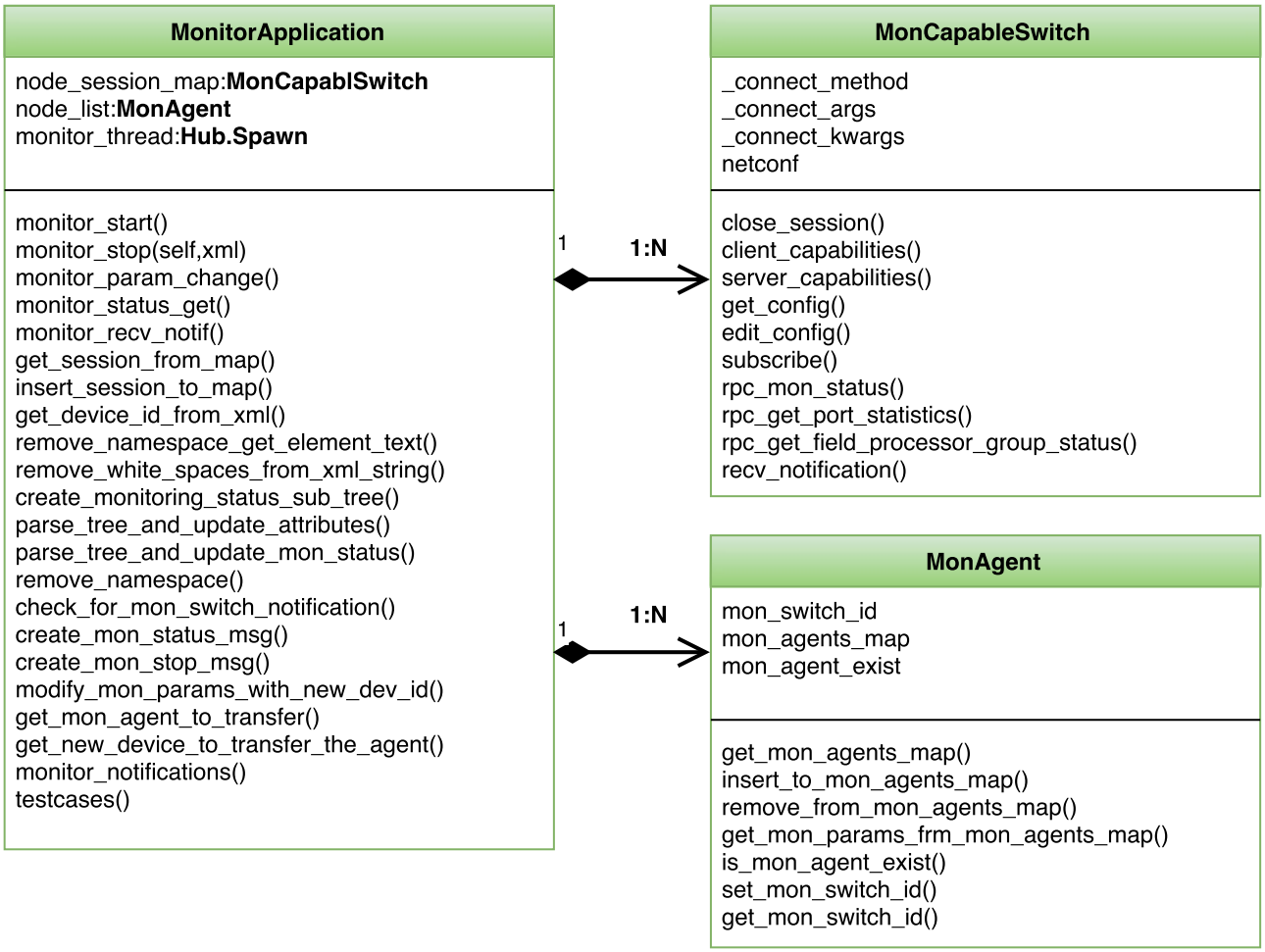


Figure 5.2: RYU Application Class Diagram

5.3.1 Asynchronous notification handling and agent movement using RYU controller application

The pseudo-code 3 presents the functionality of the monitoring thread. As shown, the thread will run continuously to listen to the notifications from the switches. If a MON_ SWITCH notification is received, the thread extracts the TCAM status from the notification and figures out the number of flow rules by which the threshold on the switch exceeds. At this point, it is supposed to decide the flow-rules that it should transfer onto some other switch on the path to the destination to continue monitoring. The decision making is left open and is not in the scope of the present work. Hence, as a work around, the thread gets a pre-decided (from APIs created to interface with the decision making algorithm) flow and switch to which the monitoring task will be transferred. The monitoring status obtained from the switch will be embedded in the new monitoring request that should be sent to the newly selected device. The status will contain the statistics on number of packets and number of bytes until the point the MON_SWITCH notification is received from the switch.

Algorithm 3 Notification reception and Agent movement thread

```
1: while True do                                          ▷ exit event loop if an error/interrupt occurs
2:     Traverse the timer list
3:     for all sessions in session_map do
4:         notif ← sessions.receive_notification()        ▷ Check if there is any notification for a session
5:         if (notif = MON_SWITCH) then
6:             retrieve TCAM status parameters
7:             dev_id ← notif                             ▷ Get the Device from which notification is received
8:             tcam_count ← notif                         ▷ Get the current TCAM usage limits of the monitoring table
9:             min_free_entry_dev ← notif                 ▷ Get the TCAM threshold defined on the device
10:            rules_to_transfer ← (tcam_count - min_free_entry_dev)
11:            tcam_trnsfd_count ← 0
12:            while (tcam_trnsfd_count < rules_to_transfer) do
13:                ret ← get_mon_agent_to_transfer()      ▷ Interface to external algorithm
14:                mon_status ← session.mon_status(dev_id)
15:                StatPackets ← mon_status               ▷ Feth the statistics from the reply
16:                StatBytes ← mon_status                 ▷ Feth the statistics from the reply
17:                new_device ← get_new_device_to_transfer_the_agent ▷ Interface to external algo
18:                prepare new MON_HELLO to send to new_device
19:                sessions.mon_start(new_device)         ▷ Send the monitoring parameters to new device
20:                tcam_trnsfd_count ← ((tcam_trnsfd_count + 1)
21:            end while
22:        end if
23:    end for
24: end while
```

6 Evaluation

The chapter provides insight into the experimental set-up that is used in the validation of the monitoring application and discusses the results in detail. To validate the presented work, following use-cases are defined.

- Configure the TCAM threshold on the devices under test and start monitoring the flows on one of the devices. Initiate monitoring for a number of flows until the threshold is reached. The monitoring application should detect the threshold exceeded event and report the same to the controller so that the controller could move the monitoring tasks to some other switch on the path to the destination.
- Configure a number of monitoring agents on one of the switches to fetch the flow and port-level statistics. Vary the polling intervals and observe if the agents are able to get the required statistics from the ASIC at the defined polling times.
- Configure threshold on flow statistic parameters and check if the switch is able to notify the controller when the threshold exceeds.

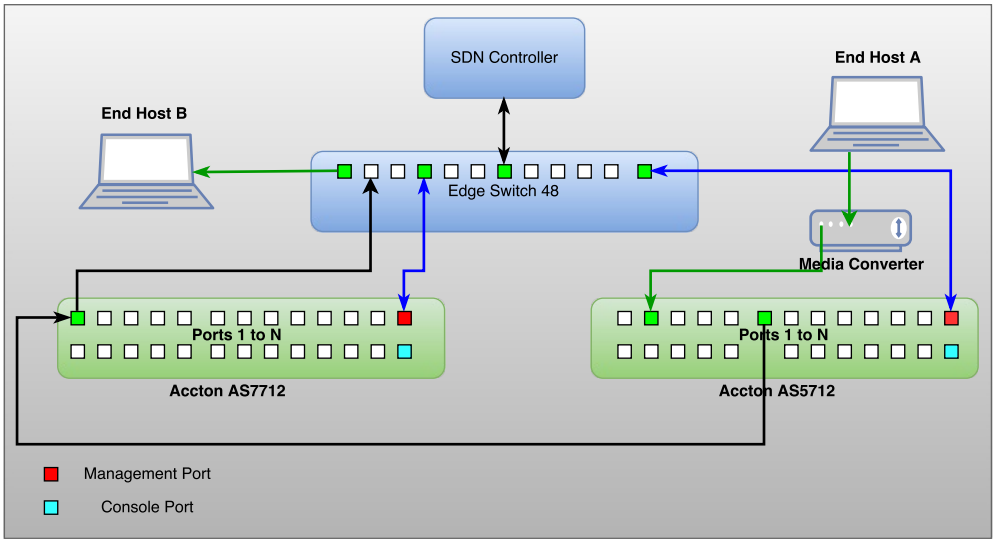


Figure 6.1: Experiment setup for the evaluation

The figure 6.1 depicts the inter-connections between the SDN controller, the programmable SDN switches and the end hosts. The following SDN switches are used in preparing the set-up:

- Accton AS5712.
- Accton AS7712.

Following sections provide a brief overview of the SDN switches and other components in terms of system specification and processing capacity.

6.0.1 Accton AS5712 and AS7712 switches

This switch is a data-center leaf device capable of processing packets at 720Gbps. It houses a Strata-XGS based Broadcom ASIC in the fast path and an Intel Atom x86 Quad-core 2.4Ghz processor as the co-located processor that could interact with the SDN controller. The switch encompasses 48 10Gb

SFP+ ports and 6 QSFP+ ports that could operate at 40Gb Ethernet speeds. It also has a management port and a console port (RJ45). The switch provides with 4000 TCAM flow-rule entries to be used with programmable APIs. Besides it also has L2 and L3 address tables that would be used as Forwarding information base.

This switch is also a data-center leaf/Top of the rack device capable of processing packets at 3.2Tbps. It houses a Broadcom's TomaHawk ASIC as the fast path processor and an Intel Atom x86 Quad-core 2.4Ghz processor as the co-located processor that could interact with the SDN controller. The switch encompasses 32 QSFP+ ports that could operate at 40Gb Ethernet speeds. It also has a management port and a console port (RJ45). The switch provides with 6K TCAM flow-rule entries to be used with programmable APIs. Besides, it also contains L2 and L3 address tables that would be used as Forwarding information base.

6.0.2 Configuration of the Traffic generator, MAC addresses, ports and route information

The SDN controller is a RYU 4.10 controller installed on a machine running ubuntu 14.10. The End-host 'A' is a windows machine running a traffic generator called Ostinato [SAS+14] [Sri10]. The End-host 'B' is merely a receiver of the traffic pumped through the switches from the End-host 'A'. The End-host 'A' is connected to a media converter which in turn runs an SFP cable to the Accton AS5712 switch. The media converter is required as the End-host does not have an SFP port and hence the SFP cable from the switch is terminated at the converter and an RJ45 cable is used further to connect the End-host 'A' with the switch. To match up the speeds with the End-host, the port on the switch is configured at 1Gb speed. Besides the SDN switches, there is also a general purpose switch (Edge switch 48) with 48 RJ45 ports and 4 SFP+ ports. This switch is used to connect the SDN controller to the SDN switches that the controller should manage.

Table 6.1: Configurations on the Switch

Configurations	Accton AS5712	Accton AS7712
Switch MAC Address	00:11:22:33:99:58	00:00:70:5B:C7:34
Next HOP MAC Address	00:00:70:5B:C7:34	00:00:70:5B:C7:35
Port Speed	Port 2 @ 1Gb Port 6 @ 10Gb	Port 2 split into 4 lanes (54,55,56,57) of 10Gb each.

The configurations on both the SDN switches are as below:

- The Accton AS5712 Port 2 is connected to the media converter (depicted by the green arrow in the figure) and it is configured to be on VLAN 10. The port speed is configured at 1Gb speed.
- The port 6 of the AS5712 switch is connected to the port 2 of the AS7712 switch as depicted by the black arrow in the figure 6.1. This link is configured to work at 10Gb speed as that is the minimum speed an Accton AS7712 switch could operate at. The link between these two switches is configured to be on the VLAN-20.
- As mentioned earlier, all the ports on the Accton AS7712 are QSFP+ ports. Hence, a breakout cable is used to split the port 2 into four 10Gb lanes. The four lanes thus form four logical ports numbered 54 through 57. The first lane i.e. logical port 54 connects the Accton AS5712 switch as described above. The second lane i.e. logical port 55 connects the switch to the Edge-switch on its SFP port as a 10Gb link. The link between the Accton AS7712 and the Edge-switch is configured to be on VLAN-10.

- The Ostinato traffic generator is used to craft the packets that are pumped into the Accton AS5712 through its port number 2.
- The MAC addresses are configured as shown in the table 6.1. The Accton AS5712 switch is configured to route any packets(with the destination IP 20.1.1.2 or any IP with subnet 55.0.0.0) arriving on ingress port 2 with VLAN-10 and destination MAC 00:11:22:33:99:58 to the next hop MAC .i.e Accton AS7712 via its egress port 6. The Accton AS7712 is configured to route any packets that arrive on its ingress port 54 with the destination IP 20.1.1.2 or any IP with subnet 55.0.0.0 through port 55 to the End-Host 'B' (on VLAN-10). It should be noted that both the switches are programmed to contain the routing information at the startup using the OpenNSL sample applications provided.
- The Accton AS5712 switch is assigned to the IP address 10.42.0.143 and the Accton AS7712 switch is assigned 10.42.0.93 as the management IP address. These IP addresses are required to manage the switches via their management ports. The SDN controller machine is assigned the IP address 10.42.0.1.

6.1 Use-case: Evaluation of Monitoring agent movement

The message sequence diagram shown in the figure 6.2 showcases the monitoring agent movement scenario. At this point, the evaluation set-up is prepared as described in the previous section. For the evaluation, 6 flows were crafted with source IP addresses 50.0.0.1-50.0.0.6 and destination IP address is fixed as 20.1.1.2. All the flows contained VLAN packets with VLAN-ID 10 in the Ethernet header. The Ostinato traffic generator [Sri10] is used to craft the packets as a single stream to be pumped at 1000 packets/second. The RYU SDN Controller is started on the ubuntu machine. The controller is by default programmed to configure a monitoring agent with ID 5 to monitor the flow with Source-IP address 50.0.0.5 and Destination-IP address 20.1.1.2. This agent is configured on the switch Accton AS5712. Since, there are no external interfaces to the SDN controller to start and stop monitoring through user interface, a command line interface program called netopeer-cli [CES] which is compliant with the Netconf protocol is used for configuring the remaining monitoring agents on the switches. The CONFIGURE_DEVICE_ID message is sent to the switches from command line interface program netopeer-cli. The Accton AS5712 is configured with the ID 10 and the Accton AS7712 is configured with the ID 20. The TCAM threshold on both the devices is configured to '5' in the custom RPC message SET_FP_ENTRY_THRESHOLD. At this point, the Accton AS5712 switch has one monitoring agent-5 configured from the SDN controller.As depicted in the flow diagram, Five more monitoring agents are configured from the CLI onto the switch which effectively increased the number of monitoring agents to six. When the TCAM limit check timer expired on the Accton AS5712 switch, it reported the MON_SWITCH notification to the controller as expected. The message contained the TCAM usage limits and the previously configured threshold. The controller, upon reception of the notification, queried the switch for the flow-statistics that it collected for the monitoring agent-5 using MON_STATUS message. It was observed from the logs that the collected statistics were included in the new monitoring request message that was sent to Accton AS7712 switch on which the monitoring for the flow continued. The corresponding Wireshark log file has been uploaded to the code repository.

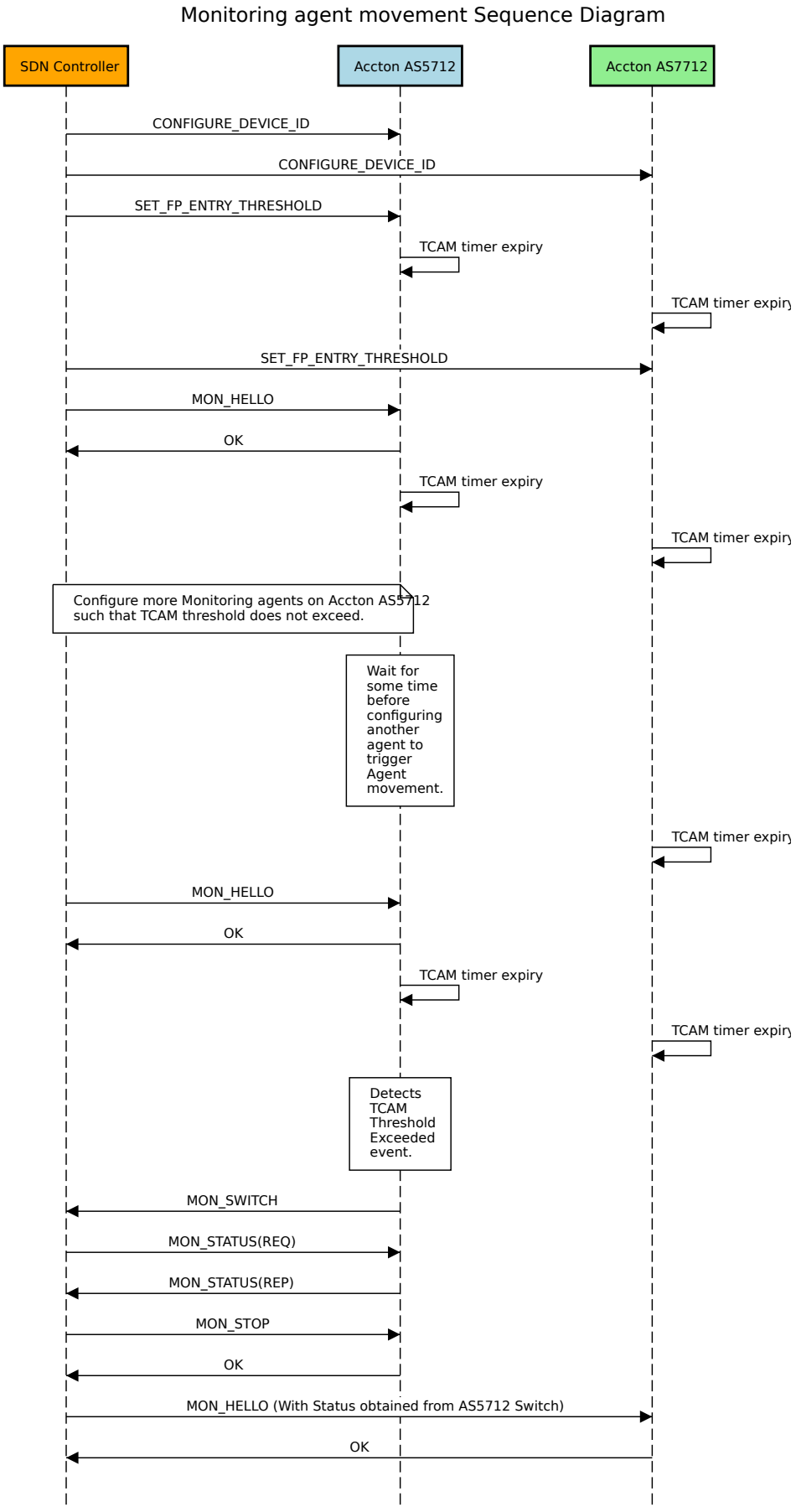


Figure 6.2: Message Sequence Diagram for agent movement scenario

6.2 Use-case: Polling statistics at defined time interval granularity

The following section throws light on two different polling strategies followed while fetching the statistics from the TCAM. The first method provides performance details when the agents are grouped together and associated a single timer which would dispatch an event handler upon expiry to fetch the statistics. The second strategy used 1 timer per monitoring agent and distributed the polling activity over time to compare the performances.

6.2.1 Polling accuracy using 1 timer by grouping the monitoring agents with similar polling requirements

The monitoring application's core functionality is to provide a configurable framework such that the flow/port statistics could be fetched at desired time intervals. The idea behind this use-case is to check if the timers used for polling are scheduled on time such that the statistics could be retrieved from the TCAM with certain guarantees. As part of this use-case, several monitoring agents i.e numbers ranging from 5 to 100 were configured on the switch with different polling requirements over multiple iterations of the experiment using the netopeer client program. The test-case files containing monitoring agent description are available with the code repository. The polling intervals were configured to be 1ms and 10 milliseconds for all the agents during separate iterations of the experiment. The monitoring application is instrumented to count the total number of samplings i.e. statistics polling done along with the delay caused in dispatching the event handler for polling. The instrumentation added also dumps the maximum and minimum delay (time elapsed) from the configured polling interval after which an agent with whom the timer is associated gets the opportunity to pull the statistics. The CPU utilization over each of the iteration of the experiment is also recorded using Unix "top" command through the terminal connection to the switch. The experiment is carried out on the Accton AS5712 switch. However, the set-up remained the same as depicted in the figure 6.1. Since only one switch is sufficient to carry out the polling, Accton AS5712 is chosen. The Ostinato traffic generator is configured to pump a different number of flows during each iteration of the experiment. The details pertaining to the number of flows pumped, polling intervals configured and the observed accuracy is as depicted in the results. The log files created by the monitoring application during each iteration are also present in the code repository.

The bar chart illustrates the accuracy of scheduling 1-millisecond timer for polling statistics for a various number of flows. The behaviour is observed over a period of 2-3 minutes while pumping UDP traffic at the rate of 1000 packets/second to 120K packets/second with default payload provided by the Ostinato traffic generator.

Figure 6.3: polling accuracy with poll-time+/-1 tolerance

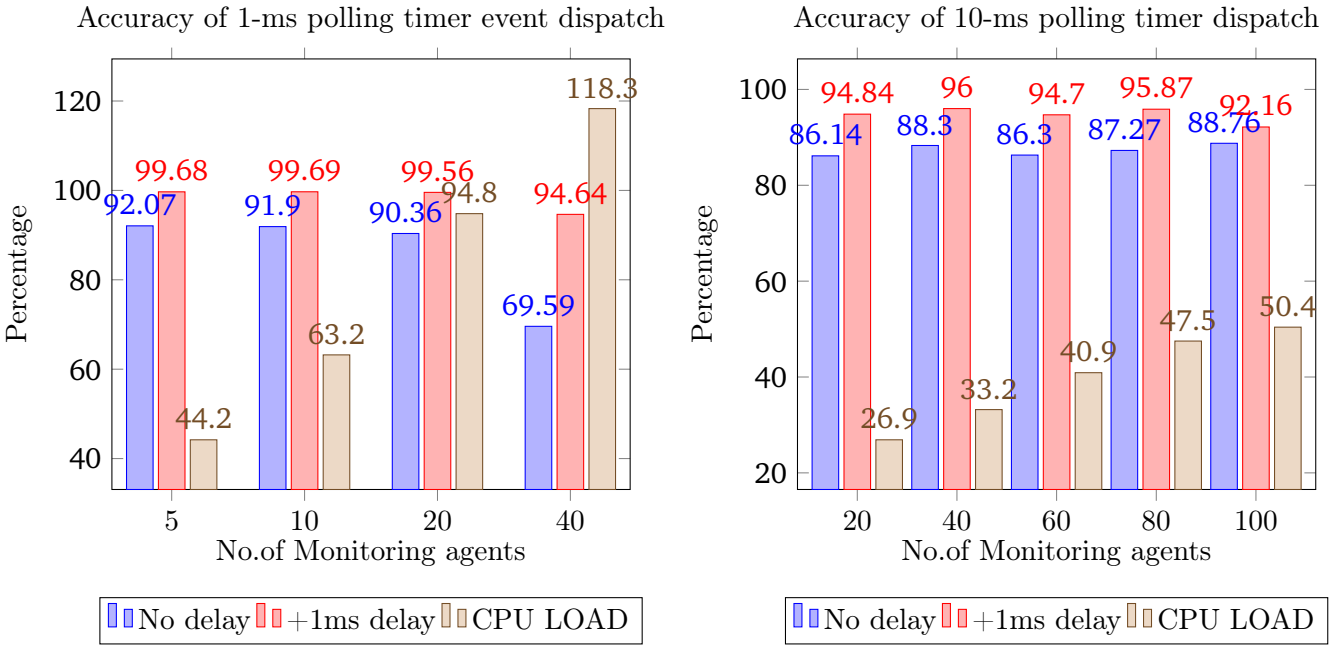


Table 6.2: Dispatching 1ms timer

No. of flows	Max delay	Min delay
5	25	1
10	32	1
20	32	1
40	36	1

Table 6.3: Dispatching 10ms timer

No. of flows	Max delay	Min delay
20	25	1
40	32	1
60	30	1
80	45	1
100	46	1

With regards to timer scheduling accuracy, when 5 flows were configured on the switch (each with 1 millisecond polling requirement), the monitoring application grouped all the flows under a single timer as they had similar polling requirement and as it could be seen 99.68% (considering the 1ms delay) of times the timer was scheduled to execute the poll event handler such that the statistics for all the configured flows could be collected from the TCAM. The experiment was repeated by configuring 10 flows, 20 flows and 40 flows over 3 more separate iterations. All the flows were configured with 1-millisecond polling accuracy. While the timers were scheduled on time for about 99.69% and 99.56% for polling 10 flows and 20 flows, it dropped to 94.64% when 40 flows were requested to be polled. It is also worth noting that the polling is delayed by 1ms for about 25.05% of the times in the last iteration containing 40 flows. However, during the earlier iterations of 1ms timer experiment (5,10 and 20 agents), 1ms deviation from the poll-time is only in the range 7-9% and hence can be considered fairly accurate. From the observations, It can be concluded that up to 20 monitoring agents are polled accurately as they experienced delays less number of times.

Turning to CPU load utilization, due to the polling activities, it could be seen from the graph that initially monitoring application used up 44.2% of the CPU and the usage level kept increasing over next iterations of the experiment as each iteration encompassed polling requirement for more number of flows. The CPU utilization crossed 100% limit when monitoring requirement for polling 40 flows was configured.

The second bar chart on the right depicts the timer scheduling accuracy when flows were configured for polling every 10 milliseconds. Again the application aggregated all the monitoring requirements under a single timer of 10-ms as they all had similar polling requirement. As it can be seen, the experiment was conducted for 20,40,60,80 and 100 flow monitoring over 5 different iterations. The accuracy of scheduling the timer was observed to be the highest at 96% while polling 40 flows and it dropped to the least value of 92.16% for 100 flows. With regards to CPU utilization levels for experiments with 10ms polling requirement, it was observed that the CPU load caused by the monitoring application remained low at 26.9% for 20 flows and it kept increasing approximately at the rate of 7% over next iterations and reached 50.4% while polling statistics for 100 flows.

The table 6.2 refers to the delays observed in dispatching timer expiry event handler for polling the statistics. As mentioned earlier, for calculating the delay (i.e.time it would take for invoking the polling event handler s_timer_event between any two polling intervals), an instrumentation is introduced in the application that would dump the delays experienced between 0 to 100 in milliseconds. It is observed that maximum time delays caused in dispatching event handler are in the range 25-36 milliseconds across all the iterations carried out to verify 1ms polling. The maximum delay varied between 25 and 46 milliseconds across several iterations of the experiment in the case of 10ms timer dispatch as depicted in the table 6.3.

The likely explanation for the delayed polling and longer execution of polling mechanism is as below:

- The time that is taken by an event handler upon timer expiry has an effect on the delay. If the event handler executes for a period longer than the polling interval then naturally event handler would be dispatched after a delay over the next interval (although timer expires on time) as the event reactor loop would wait for the already executing handler to return.

- The system calls executed by the event reactor loop of the CZMQ library to wait for any messages from the Netconf server and also the driver API executed by the monitoring application to fetch flow statistics may cause the application to block for a random amount of time due to CPU scheduling policies. This would affect the timer dispatch accuracy as well.

- The CPU utilization level could be directly attributed to number of extra CPU instructions that the polling mechanism/event handler has to execute when more flows are configured.

It should be noted that during every iteration of the experiment (that lasted for about 2-3 minutes as said earlier), the delay crossed 100 milliseconds mark occasionally for about 8-9 times. However, since most of the times the polling was done at the desired intervals, it could be concluded that the performance of the framework which follows aggregation of similar polling requirements under a single timer strategy is fairly accurate.

During the experiments related to 10ms polling requirement, the execution time of the event handler for polling all the monitoring agents is recorded and is as in the table 6.4. The execution time is calculated using an instrumentation in the source code that calculates the difference between clock times taken at the start and end of a polling event handler, using the monotonic clock provided by CZMQ library. For simplicity, the observed values are rounded to the next highest multiple of 10. The values are considered from the most idealistic polling intervals ruling out a few outliers that cause huge delay. As described earlier, since all the flows over these 5 iterations of the experiment were configured with 10ms polling interval requirement, the application had grouped them together and since the monitoring

agent with ID-1 was the first agent to be configured, it was given the responsibility of executing the polling for all the monitoring agents as well when the polling timer expired every 10ms. It could be seen from the table that the execution time to fetch flow statistics for 20 flows is in the range 650-700 microseconds which implies that monitoring agent whose statistics is retrieved at the end experiences some delay beyond its polling requirement of 10ms. It is not clear at the moment if algorithms that would run on the switch require polling accuracy at the level of microsecond granularity. However, it is quite alarming that when 100 flows are configured with 10ms polling requirement, the last monitoring agent in the group experienced approximately about 2900-2950 microseconds delay in getting its flow statistics which translates to 3ms delay. The argument holds good for other iterations of 40, 60 and 80 flow statistics polling as well.

Table 6.4: Polling event-handler execution time for all the flows with 10ms timer interval.

No. of flows	Polling event handler execution time
20	650-700 microseconds
40	1300 microseconds.
60	1800 microseconds
80	2400 microseconds
100	2900-2950 microseconds

While following polling requirement aggregation strategy, If on the one hand a few monitoring agents experience a delay in getting the statistics, on the other hand, the CPU load needs to be balanced out as it can be seen from the bar chart. One way to balance the load is to spread the polling activity over time-domain such that a few flows are configured with a certain polling interval requirement while the other flows are configured with a different polling accuracy. This would increase the CPU idle time and hence reduce the CPU load. However, it has to be empirically evaluated and this work leaves it open for further investigation on CPU load balancing. But, to address the issue of latency experienced by the flows in a group based polling, a separate experiment is carried out explanation of which is available in the next subsection.

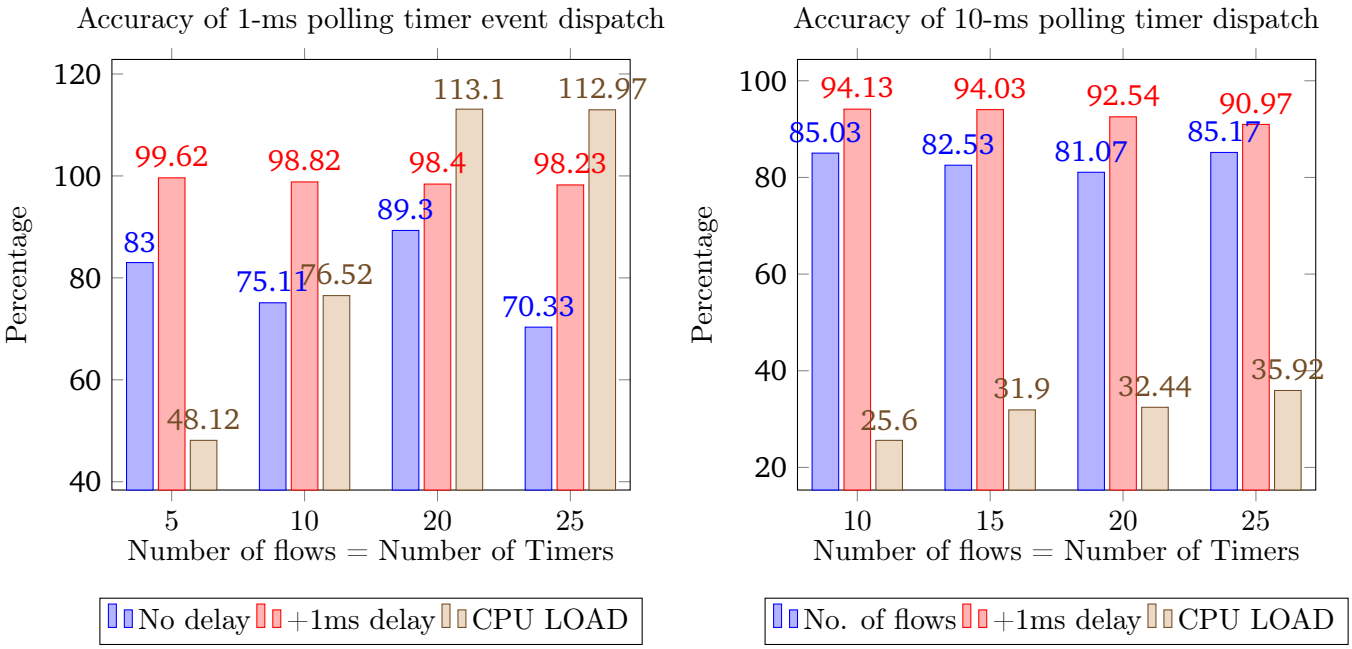
6.2.2 Polling accuracy using 1 timer per monitoring agent

To calculate the time it would take for a monitoring agent to fetch the flow statistics from the TCAM, an experiment was conducted with 40 monitoring agents configured for polling statistics at 1-millisecond polling interval. From the results, it is observed that during each interval, every monitoring agent executed the event handler to fetch statistics for about 33-50 microseconds. As it ensures less execution time, more iterations were carried out while assigning separate timers and event handlers for each monitoring agent. The details are as below:

The bar chart on the left of the figure 6.4 illustrates the polling done for the monitoring agents with polling interval 1ms. It should be noted that to analyse the accuracy of polling, a delay tolerance of 1ms is considered in both the experiments. Each monitoring agent is associated with its own timer and event handler. The accuracy is inferred to be the highest at 98.4% as the 1ms delay in polling is only 9.1% of the time while polling 20 monitoring agents. Although the overall accuracy of polling remained in the range 98-99% while polling statistics of 5, 10 and 25 flows, it is seen that these iterations of the experiment suffered a 1ms delay in the range of 16-28% in getting the events dispatched for polling the statistics. As far as the CPU utilization is concerned, since polling statistics from the TCAM every 1ms is heavily CPU intensive, as expected, the CPU load kept increasing as the number of monitoring agents increased and reached a maximum of 113% when 25 agents were configured on the switch.

The experiment repeated for 10ms polling interval is depicted in the bar graph on the right side of the figure 6.4

Figure 6.4: Polling performance with 1 timer per agent, accuracy=(poll-time +/- 1)



It appears that the accuracy (considering 1ms delay) of polling more or less remained same for iterations with 10 and 15 monitoring agents at 94% while it dropped to 90% when 25 monitoring agents were configured. The iterations with 10,15 and 20 monitoring agents experienced a delay of 1ms in the range between 9 and 11%. But, the situation improved for 25 monitoring agents as it experienced 1ms delay in getting its event dispatched for only 5.8% of the times.

With regards to the CPU load, since the polling interval is configured to be at 10ms, as expected, the load on the CPU is lower than the iterations with 1ms polling interval of the previous experiment. It could be seen that the load is at 25.6% for 10 flows and constantly increased to a maximum of 35.92% when 25 monitoring agents were present on the switch.

6.2.3 Summary of polling accuracy

From the two experiments i.e. aggregating the monitoring agents under one single timer and associating separate timers to each monitoring agent, it can be concluded that the performance is better and reliable when the monitoring agents with similar polling requirements are grouped for statistics polling as they experience a relatively low delay in scheduling the event handler. Except for the iteration with 40 monitoring agents with 1ms polling requirement which experienced 1ms delay for about 25% of times, the grouping strategy fared quite accurately and all the remaining iterations that followed grouping strategy experienced 1ms delay in the range 7-9% only. It implies that up to 20 flow rules could be monitored with an accuracy of 99.56% using 1ms polling interval when the group strategy is employed.

The second strategy to associate a separate timer does not really show a consistent behaviour across multiple iterations of the experiment. Hence, it can be concluded that the algorithms that expect polling accuracy at millisecond granularity could rather follow grouping strategy while using the monitoring

framework designed and developed as part of this work. If the monitoring algorithms are designed to be delay tolerant and if they are designed to work with the statistics obtained over a period of time for several flows, they could follow the second strategy of associating separate timers. However, it could also have an adverse effect on the CPU load as such an algorithm would be continuously carrying out computations on the dataset to produce useful measurement results.

6.3 Use-case: Event detection and Link utilization through Monitoring Agent

This section is dedicated to showcase the various event detection possibilities readily available with monitoring application.

6.3.1 Event detection: Detecting bytes and packets exceed event in a flow

As part of this experiment, a monitoring agent was deployed on the switch (using the netopeer client program) that could keep track of the number of packets and bytes received within a flow. The packet threshold is defined as 500 packets and the bytes threshold is defined as 5000 bytes. The monitoring agent depicted in the listing 8.10 is configured to carry out the action "NOTIFY_CNTRLR_AFTER_N_INTERVALS". The action signifies that an agent tracks if the specified thresholds exceed over 'N' successive intervals and reports only positive results. In this case, a number of successive intervals (time-window) over which an agent is expected to track the events is configured using the parameter "number-of-poll-intervals" as depicted in the listing 8.10. A flow with source IP 50.0.0.1 and destination IP 20.1.1.2 is configured to be monitored by the agent. The Ostinato generator is used to pump the traffic with the mentioned characteristics in the packet header. The "EVENT_N_INTERVALS.ossn" session is created in the traffic generator and is available in the code repository. The session pumps 10000 packets initially ensuring that the packet threshold does not exceed and later it adds more packets to the traffic flow by rapidly generating traffic such that both the packets and bytes thresholds exceed continuously. It was observed that the agent successfully detected the packets and bytes threshold exceeded event over 5 successive intervals and reported the same at the end of the defined time window to the SDN controller.

```
1 eventTime: Tue Mar 14 20:03:00 2017
2 <MON_EVENT_NOTIFICATION xmlns="urn:ietf:params:xml:ns:netconf:notification
     :1.0">
3 <mon-id>1</mon-id>
4 <device-id>10</device-id>
5 <flow-events>
6 <stat-packets>17194</stat-packets>
7 <stat-bytes>1100416</stat-bytes>
8 </flow-events>
9 </MON_EVENT_NOTIFICATION>
```

Listing 6.1: MON_EVENT_NOTIFICATION

The listing 6.1 is an event notification from the experiment and it provides the number of packets and bytes accounted until the event notification was generated. The log files generated by the monitoring application as well as the notifications captured through Netopeer Netconf client program are present in the "log_event_reporting" folder of the code repository.

6.3.2 Link Utilization: Reporting port and flow statistics over an interval

Link utilization is basically used to find out the amount of bandwidth being used by a flow on a given direction/physical link connected to any port of a network device. An experiment was also carried out to

calculate the link utilization on the ingress port 2 of the Accton AS5712 switch using the same testbed. A flow with source-destination IP pair 50.0.0.1 and 20.1.1.2 was pumped at the rate of 10K packets per/sec from the Ostinato generator. The monitoring agent was configured as in the listing 8.11. The agent expressed the need to poll both the port statistics as well as the specific flow statistics using the symbolic constant "FP_STATS". The polling interval is configured at 10seconds and link utilization threshold is also configured to be at 10%. The monitoring agent is expected to report if the flow it is monitoring consumes more than 10% of the total volume of traffic on the port over any interval. It was observed that the monitoring agent appropriately and continuously reported the number of unicast and non-unicast packets received on the port during every interval of 10 seconds along with the number of packets received as part of the mentioned traffic flow.

```
netconf> eventTime: Mon Mar 27 22:34:26 2017
<MON_LINK_UTILIZATION xmlns="urn:ietf:params:xml:ns:netconf:notification
   :1.0">
<mon-id>1</mon-id>
<device-id>10</device-id>
<IfInUcastPkts>97441</IfInUcastPkts>
<IfInNUcastPkts>0</IfInNUcastPkts>
<stat-packets>99615</stat-packets>
</MON_LINK_UTILIZATION>
```

Listing 6.2: MON_LINK_UTILIZATION

This kind of event detection or link utilization will be helpful in detecting heavy hitters[ZSS+04] leading to various issues pertaining to bandwidth allocation. It should be noted that link threshold should be configured as shown in the listing for this feature to function appropriately. The listing is a snapshot of the MON_LINK_UTILIZATION notification, received by the CLI program, from the AS5712 switch.

The other way of fetching the link utilization is to follow standard SDN way to fetch flow and port-level statistics from the SDN controller. The monitoring application has exposed the PORT_STATISTICS and MON_STATUS APIs for this purpose.

7 Conclusion and Future work

As part of the presented work, a non-exhaustive list of the literature relevant to SDN monitoring is studied in detail and observed that most of the network monitoring solutions proposed for SDN-enabled networks are slanted towards harnessing OpenFlow messages. The disadvantage of this approach is primarily due to the fact that the OpenFlow protocol itself is not designed for network monitoring purposes. Also, network monitoring is SDN controller centric which would potentially make it a single point of failure for the network. Some of the works even highlighted the need to distribute the monitoring tasks amongst all the network devices while the controller would still coordinate with them for statistics periodically. As part of the thesis, it has been showcased that the statistics polling could be done at different levels of accuracy and time interval granularity on the switch itself and therefore making it possible to run monitoring algorithms on the devices. While doing so, the work exposes necessary communication mechanisms for the controller to keep track of the monitoring activities as it is still the best possible point to analyse the network health with its ability to obtain global network view. The controller could still commission the monitoring agents on the best fit devices in the network for any particular monitoring task. It has also been showcased that, to overcome the memory limitations on the switch, a mechanism to continuously track the memory utilization (similar to Vacancy events[RX14] in latest OpenFlow standards) could be used as in this work and if required a few monitoring agents/tasks could be moved to some other device(s) with the help of the SDN Controller. The polling mechanism coupled with event notification capabilities of the monitoring application can be readily used to calculate link utilization/hierarchical heavy hitter detection. It also facilitates detection of a change in traffic patterns by defining thresholds on the number of packets and bytes received per flow.

7.1 Future work

- The RYU SDN controller developed as part of the work exposes the required APIs to carry out monitoring. However, it leaves it open for implementation, an application that would provide an user-interface that could make use of the monitoring application on the controller side. Also, the application has created interfaces to be used with an external algorithm that would wisely decide upon the flows that should be monitored and the network devices on which monitoring should be carried out. The application could also be made to interact with other RYU applications that learn traffic flows using the RYU's event-based programming paradigm.

- The OpenNSL library exports APIs to fetch buffer statistics, class of service queue statistics and also to monitor the links. Since the present work focused predominantly on innovative ways to overcome TCAM resource limit and flow monitoring on the devices itself to aid global detection tasks such as link utilization and change in traffic pattern, the above-mentioned APIs are not considered in this work. However, it would be a valuable add-on to monitor local resources such as packet buffer allocation to different class of service queues. Thereby, traffic engineering activities could be done with more accuracy and should also be able to give time guarantees on buffer tuning to achieve Quality of Service agreements. The existing work from Broadcom which is called as Broadview does the buffer allocation monitoring outside the switch. Additionally, A feature to track the CPU load on the switch could also be developed and integrated into the framework.

- The monitoring agents install flow rules into a TCAM table that can match only IP source-destination pairs. In future, a separate TCAM table to match the destination port could also be considered as it gives more control on the granularity of the flows selected for monitoring.

8 Appendix

The following sections provide insight into software development tools, software build and installation on the switch. It also lists the XML format of all the monitoring messages.

8.1 Monitoring messages and their XML format

```
<?xml version="1.0"?>
<monitoring-model xmlns="http://monitoring-automata.net/sdn-mon-automata" xmlns:nc="urn:ietf:params:xml:ns
    :netconf:base:1.0">
<monitoring-agent nc:operation="create">
<mon-id>50</mon-id>
<mon-type>FP</mon-type>
<device-id>20</device-id>
<port-index>2</port-index>
<poll-time>25</poll-time>
<state-machine>
<TotalStates>1</TotalStates>
<state-table-row-entries>
<state>1</state>
<input-events>
<num_of_row_evnts>0</num_of_row_evnts>
</input-events>
<flow_to_install>
<src_ip>50.0.0.1</src_ip>
<dst_ip>20.1.1.2</dst_ip>
<src_ip_mask>255.255.255.255</src_ip_mask>
<dst_ip_mask>255.255.255.255</dst_ip_mask>
</flow_to_install>
<num_of_actions>2</num_of_actions>
<action>
<A1>NOTIFY_CNTLR</A1>
</action>
<next-state>255</next-state>
</state-table-row-entries>
</state-machine>
</monitoring-agent>
</monitoring-model>
```

Listing 8.1: MON_HELLO message

```
<?xml version="1.0"?>
<monitoring-model xmlns="http://monitoring-automata.net/sdn-mon-automata" xmlns:nc="urn:ietf:params:xml:ns
    :netconf:base:1.0">
<monitoring-agent nc:operation="remove">
<mon-id>50</mon-id>
</monitoring-agent>
</monitoring-model>
```

Listing 8.2: MON_STOP message

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
<get_field_processor_group_status xmlns="http://monitoring-automata.net/sdn-mon-automata">
<fp_monitoring_group>TCAM-MONITORING-TABLE</fp_monitoring_group>
</get_field_processor_group_status>
</rpc>
```

Listing 8.3: FP GROUP STATUS Operation

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<set-fp-entry-threshold xmlns="http://monitoring-automata.net/sdn-mon-automata">
<fp-entry-threshold>
<total-entries-threshold> 245</total-entries-threshold>
<total-counters-count>100</total-counters-count>
<min-free-entries-per-device>3755</min-free-entries-per-device>
</fp-entry-threshold>
</set-fp-entry-threshold>
</rpc>
```

Listing 8.4: CONFIGURE TCAM Threshold Operation

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
<mon_status xmlns="http://monitoring-automata.net/sdn-mon-automata">
<mon-id>200</mon-id>
<device-id>50</device-id>
</mon_status>
</rpc>
```

Listing 8.5: MONITORING STATUS Operation

```
<?xml version="1.0"?>
<monitoring-model xmlns="http://monitoring-automata.net/sdn-mon-automata" xmlns:nc="urn:ietf:params:xml:ns
    :netconf:base:1.0">
<monitoring-agent nc:operation="merge">
<mon-id>100</mon-id>
<poll-time>10</poll-time>
</monitoring-agent>
</monitoring-model>
```

Listing 8.6: MON_PARAM_CHANGE message

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
<port_statistics_clear xmlns="http://monitoring-automata.net/sdn-mon-automata">
<clear-port-stats>
<port-index>2</port-index>
</clear-port-stats>
</port_statistics_clear>
</rpc>
```

Listing 8.7: PORT STATISTICS CLEAR Operation

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<configure-device-id xmlns="http://monitoring-automata.net/sdn-mon-automata">
<switch-identification>10</switch-identification>
</configure-device-id>
</rpc>
```

Listing 8.8: CONFIGURE DEVICE ID Operation

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
<port_statistics xmlns="http://monitoring-automata.net/sdn-mon-automata">
<port-statistics-get>
<port-index>2</port-index>
</port-statistics-get>
</port_statistics>
</rpc>
```

Listing 8.9: GET PORT STATISTICS Operation

8.2 Monitoring agent for event detection

```
<?xml version="1.0"?>
<monitoring-model xmlns="http://monitoring-automata.net/sdn-mon-automata" xmlns:nc="urn:ietf:params:xml:ns
    :netconf:base:1.0">
<monitoring-agent nc:operation="create">
<mon-id>1</mon-id>
<mon-type>FP</mon-type>
<device-id>20</device-id>
<port-index>2</port-index>
<poll-time>100</poll-time>
<number-of-poll-intervals>5</number-of-poll-intervals>
<state-machine>
<TotalStates>1</TotalStates>
<state-table-row-entries>
<state>1</state>
<input-events>
<num_of_row_evnts>2</num_of_row_evnts>
<statPktsThreshold>500</statPktsThreshold>
<statBytesThreshold>5000</statBytesThreshold>
</input-events>
<flow_to_install>
<src_ip>50.0.0.1</src_ip>
<dst_ip>20.1.1.2</dst_ip>
<src_ip_mask>255.255.255.255</src_ip_mask>
<dst_ip_mask>255.255.255.255</dst_ip_mask>
</flow_to_install>
<num_of_actions>1</num_of_actions>
<action>
<A1>NOTIFY_CNTLR_AFTER_N_INTERVALS</A1>
</action>
<next-state>255</next-state>
</state-table-row-entries>
</state-machine>
</monitoring-agent>
</monitoring-model>
```

Listing 8.10: Monitoring agent for event detection

8.3 Monitoring agent for LINK UTILIZATION

```
<?xml version="1.0"?>
<monitoring-model xmlns="http://monitoring-automata.net/sdn-mon-automata" xmlns:nc="urn:ietf:params:xml:ns
    :netconf:base:1.0">
<monitoring-agent nc:operation="create">
<mon-id>1</mon-id>
<mon-msg-type>MON_HELLO</mon-msg-type>
<mon-type>FP_STATS</mon-type>
<device-id>10</device-id>
<port-index>2</port-index>
<poll-time>10</poll-time>
<link-util-threshold>10</link-util-threshold>
<state-machine>
<TotalStates>1</TotalStates>
<state-table-row-entries>
<state>1</state>
<input-events>
<num_of_row_evnts>0</num_of_row_evnts>
</input-events>
<flow_to_install>
<src_ip>50.0.0.1</src_ip>
<dst_ip>20.1.1.2</dst_ip>
<src_ip_mask>255.255.255.255</src_ip_mask>
<dst_ip_mask>255.255.255.255</dst_ip_mask>
</flow_to_install>
<num_of_actions>1</num_of_actions>
<action>
<A1>NOTIFY_LINK_UTILIZATION</A1>
</action>
<next-state>255</next-state>
</state-table-row-entries>
</state-machine>
</monitoring-agent>
</monitoring-model>
```

Listing 8.11: Monitoring agent for link utilization

8.4 System software installation on the switch

A fundamental building block for an user-space application to run on a switch is an operating system. The bare-metal switches that are used to evaluate the work contain Intel x86 processors. In this work, Open Network Linux[She14] [ONL] is used as the operating system running on the x86 processor. The ONL installation is carried out as below:

- Download the ONL binaries from the website. The ONL distribution consists of three images. Belwo mentioned version of the binaries were considered for installation.
  - ONL-2.0.0-ONL-OS-DEB8-2016-09-04.1726-3f30495-AMD64.swi
  - ONL-2.0.0-ONL-OS-DEB8-2016-09-04.1726-3f30495-AMD64-SWI-INSTALLER
  - ONL-2.0.0-ONL-OS-DEB8-2016-09-04.1726-3f30495-AMD64-INSTALLED-INSTALLER
- Format a flash drive into linux compatible file system, preferably ext4.
- Rename the file "ONL-2.0.0-ONL-OS-DEB8-2016-09-04.1726-3f30495-AMD64-INSTALLED-INSTALLER" as "onie-installer".
- Follow the below steps to copy the binaries onto the flash drive:
  - sudo mkdir /mnt/usb
  - sudo mount /dev/sdb1 /mnt/usb
  - sudo cp ONL-2.0.0-ONL-OS-DEB8-2016-09-04.1726-3f30495-AMD64-INSTALLED-INSTALLER /mnt/usb/onie-installer
  - sudo umount /mnt/usb
  - copy the other two binaries also onto the same path as above.
- Insert the flash drive into the USB port of the switch and power on the switch.
- Since there is no OS pre-installed on the switch, it will boot into ONIE [SE] which is a small footprint Linux created for installing a network OS on a bare-metal switch[Tog].
- Select "install OS" option when the switch boots up.
- The installation process should happen automatically from this point and when it completes, the switch reboots and the linux GRUB displays Open Network Linux as an option to select.
- Selecting the Open Network Linux will load the OS and prompt for the password later. The username is "root" and password is set to "onl" by default.
- Known issue during OS installation: The default boot configuration at the time of installation of ONL requires a DHCP server to be present. But, with the OS version chosen, DHCP fucntionality is missing which prevents the system to boot up the OS completely. To overcome the issue, one has to stop the DHCP discovery when the "loader#" prompt appears on boot up and then execute the commands "sed -i /NETAUTO/d /mnt/onl/boot/boot-config" followed by "autoboot" to make sure that the system does not look for a DHCP server during the start-up. It takes a system reboot again at this point to get the OS loaded properly for the first time. However, the switch would boot up successfully without issues on subsequent restarts.

8.5 Monitoring application build and installation on the switch

There are two possibilities to compile the monitoring-application source code. One way is to build the source on a Linux machine for testing purposes. In this case, the OpenNSL library will have to be disabled and the stubs should be used such that wrappers around library APIs always return success upon invocation. The second way is to develop, test and deploy on the switch itself. Since the switch runs a standard Linux distribution, all the required tools for software development could be installed on the switch. However, for convenience (such as GUI availability), it would be preferable to develop on a separate Linux machine.

The monitoring application depends on the following libraries.

- OpenNSL library (3.2.0.5): The driver API library to program the ASIC.
- OVSLIB (openvswitch-2.3.1): For the hash-map implementation.
- DZLOG (): Used as a logging library.
- CZMQ (3.0.2) & ZeroMQ (4.1.5): For inter-process communication and the event loop reactor pattern implementation.

It also requires autotools, libtools and pkg-config to be installed on the switch. Once all the libraries and dependencies are set-up, the compilation and installation is a one step process which is to use the automake's "make install" command from the directory that contains the makefile for the monitoring application. To start the application, run the command "Monitor-automata" from the terminal of the switch. The dzlog[Sim] library linked to the monitoring application requires that a configuration file by name "zlog.conf" be present in /etc/ path on the switch. This configuration file allows to specify logging at different levels such as error, information and debug.

8.6 Netconf server build and installation on the switch

The Netconf Server application requires following libraries to be present on the switch for its proper functioning. autoconf, automake, gcc, libnetconf, libtool, libxml2, libxml2-devel, m4, make, openssl, openssl-devel and pkgconfig. It also require previously mentioned ZeroMQ libraries as it communicates with the monitoring application via ZeroMQ sockets. The compilation is similar to the monitoring application. On the switch, from the source directory, "autoreconf -iv" should be executed first. It would generate a configure script that could be used to set-up the compilation environment. A simple "make install" would compile and install the Netconf server module on the switch. To start the Netconf server, execute the command "ofc-server -v 3" from the terminal. The command starts the application as a daemon process.

Bibliography

[AGM+10] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). SIGCOMM Comput. Commun. Rev., 41(4):–, August 2010.

[Akg13] Faruk Akgul. ZeroMQ. Packt Publishing Ltd, 2013.

[BBC06] Fred Baker, Jozef Babiarz, and Kwok Ho Chan. Configuration Guidelines for DiffServ Service Classes. RFC 4594, August 2006.

[BBCC14] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. ACM SIGCOMM Computer Communication Review, 44(2):44–51, 2014.

[BBMB16] Othmane Blial, Mouad Ben Mamoun, and Redouane Benaini. An overview on sdn architectures with multiple controllers. J. Comput. Netw. Commun., 2016, April 2016.

[Bjo10] Martin Bjorklund. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020, October 2010.

[BTS09] Shikhar Bhushan, Ha Manh Tran, and Jürgen Schönwälder. Ncclient: A python library for netconf client applications. In International Workshop on IP Operations and Management, pages 143–154. Springer, 2009.

[CB10] N.M. Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. Comput. Netw., 54(5):862–876, April 2010.

[CES] CESNET. Netopeer-client. Accessed: 1-2-2017.

[CFSD90] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple network management protocol (snmp), 1990.

[Cis] Cisco. TCAM width. `http://www.cisco.com/c/en/us/support/docs/switches/nexus-9000-series-switches/119032-nexus9k-tcam-00.html`. Accessed: 2017-3-1.

[CT] CISCO-TCAM. A brief note on TCAM. `https://supportforums.cisco.com/document/60831/cam-content-addressable-memory-vs-tcam-ternary-content-addressable-memory`. Accessed: 2016-11-1.

[dPFSEG15] P. R. da Paz Ferraz Santos, R. P. Esteves, and L. Z. Granville. Evaluating snmp, netconf, and restful web services for router virtualization management. In 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pages 122–130, May 2015.

[EBBS11] Rob Enns, Martin Bjorklund, Andy Bierman, and JÃijrgen SchÃűnwÃď lder. Network Configuration Protocol (NETCONF). RFC 6241, June 2011.

[EJR+00] F. Engel, K.S. Jones, K. Robertson, D.M. Thompson, and G. White. Network monitoring, September 5 2000. US Patent 6,115,393.

[ERWC14] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. Performance characteristics of virtual switching. In Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on, pages 120–125. IEEE, IEEE, 2014.

[GYG13] A. Gelberger, N. Yemini, and R. Giladi. Performance analysis of software-defined networking (sdn). In 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, pages 389–393, Aug 2013.

[ID] Intel-DPDK. Packet processing pipe line discussion in DPDK. http://dpdk.org/doc/guides-16.04/prog_guide/packet_framework.html#pipeline-library-design. Accessed: 2017-3-2.

[INT] INTEL. Performance Evaluation of Intel DPDK based Virtual switches. https://01.org/sites/default/files/page/intel_dpdk_vswitch_performance_figures_0.10.0_0.pdf. Accessed: 2017-2-13.

[Jia13] Weirong Jiang. Scalable ternary content addressable memory implementation using fpgas. In Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '13, pages 71–82, Piscataway, NJ, USA, 2013. IEEE Press.

[JYR11] Lavanya Jose, Minlan Yu, and Jennifer Rexford. Online measurement of large traffic aggregates on commodity switches. In Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[Kre13] Radek KrejcÃŋ. Building netconf-enabled network management systems with libnetconf. In Filip De Turck, Yixin Diao, Choong Seon Hong, Deep Medhi, and Ramin Sadre, editors, IM, pages 756–759. IEEE, 2013.

[LIB] LIBNETCONFGUIDE. LIBNETCONF Developer manual. https://rawgit.com/CESNET/libnetconf/master/doc/doxygen/html/d9/d25/transapi.html#understanding-parameters. Accessed: 2016-09-11.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38(2):69–74, March 2008.

[MWCS14] Mehdi Malboubi, Liyuan Wang, Chen-Nee Chuah, and Puneet Sharma. Intelligent SDN based traffic (de)aggregation and measurement paradigm (istamp). In 2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014, pages 934–942. IEEE, 2014.

[NBBB98] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), December 1998. Updated by RFCs 3168, 3260.

[Net] Juniper Networks. TCAM usage on Juniper devices. https://www.juniper.net/techpubs/en_US/junos/topics/reference/command-summary/show-pfe-tcam-usage-acx-series.html. Accessed: 2017-3-1.

[NSBT16] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Rules placement problem in openflow networks: a survey. IEEE Communications Surveys & Tutorials, 18(2):1273–1286, 2016.

[OD] BroadCom OF-DPA. An Overview of OF-DPA. https://www.broadcom.com/products/ethernet-connectivity/software/of-dpa. Accessed: 2017-3-1.

[ONFa] ONF. OpenFlow Switch Specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf. Accessed: 2016-09-13.

[ONFb] ONF. SDN DEFINITION. http://www.reactivemanifesto.org/. Accessed: 2016-10-20.

[ONL] ONL. Open network linux. Accessed: 15-3-2017.

[OO] ONF-OFCONFIG12. OF-CONFIG 1.2 protcol specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf. Accessed: 2016-09-09.

[Opea] BroadCom OpenNSL. An Overview of OpenNSL library. http://broadcom-switch.github.io/OpenNSL/doc/html/OPENNSL_OVERVIEW.html. Accessed: 2017-2-28.

[Opeb] OpenNSLBroadcom. OpenNSL features. https://www.broadcom.com/products/ethernet-connectivity/software/opennsl/#overview. Accessed: 2016-11-1.

[OT] ONFTS-017-TTP. OpenFlow Table Type Patterns. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/OpenFlow%20Table%20Type%20Patterns%20v1.0.pdf. Accessed: 2017-11-1.

[PD13] Ben Pfaff and Bruce Davie. The Open vSwitch Database Management Protocol. RFC 7047, December 2013.

[PH] KIU Shueng Chuan et.al Pieter Hintjens, Uli KÃűhler. Czmq api guide. Accessed: 1-12-2016.

[Pie13] Hintjens Pieter. Zeromq: messaging for many applications. O'Reilly Media, page 484, 2013.

[RFC] RFC_IETF. Monitoring based on NetFlow. https://www.ietf.org/rfc/rfc3954.txt. Accessed: 2016-1-20.

[RX14] Tiantian Ren and Yanwei Xu. Analysis of the new features of openflow 1.4. In 2nd International Conference on Information, Electronics and Computer, pages 73–77. Atlantis Press, 2014.

[SAS+14] Shalvi Srivastava, Sweta Anmulwar, AM Sapkal, Tarun Batra, Anil Kumar Gupta, and Vinodh Kumar. Comparative study of various traffic generator tools. In Engineering and Computational Sciences (RAECS), 2014 Recent Advances in, pages 1–6. IEEE, 2014.

[SCF+12] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. Past: Scalable ethernet for data centers. In Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, CoNEXT '12, pages 49–60, New York, NY, USA, 2012. ACM.

[Sch95] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. Commun. ACM, 38(10):65–74, October 1995.

[SE] Curt Brune Scott Emery, Pete Bratach. Onie-documentation. Accessed: 15-3-2017.

[SEKC16] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab. Sdn controllers: A comparative study. In 2016 18th Mediterranean Electrotechnical Conference (MELECON), pages 1–6, April 2016.

[Sha] ShapingPolicing. Traffic shaping and policing overview. http://www.cisco.com/c/en/us/td/docs/ios/12_2/qos/configuration/guide/fqos_c/qcfpolsh.html. Accessed: 2016-12-20.

[She14] Rob Sherwood. Tutorial: White box/bare metal switches. In Open Networking User Group meeting, New York, 2014.

[Sim] Hardy Simpson. Zlog user guide. Accessed: 1-2-2017.

[Sri10] P Srivats. Ostinato: An open, scalable packet/traffic generator. FOSS. IN, page 80, 2010.

[Str] BroadCom StrataXGS. StrataXGS Architecture by Broadcom. https://people.ucsc.edu/~warner/Bufs/StrataXGS_Trident_II_presentation.pdf. Accessed: 2017-2-28.

[TC08] Hector Trevino and Sharon Chisholm. NETCONF Event Notifications. RFC 5277, July 2008.

[Tog] Reza Toghraee. Onie os installation on bare-metal switcches. Accessed: 15-3-2017.

[TRT] RFC TRTCM, IETF. A Two Rate Three Color Marker. https://tools.ietf.org/html/rfc2698. Accessed: 2017-3-3.

[YJM13] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, nsdi'13, pages 29–42, Berkeley, CA, USA, 2013. USENIX Association.

[YLZ+13] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. In Proceedings of the 14th International Conference on Passive and Active Measurement, PAM'13, pages 31–41, Berlin, Heidelberg, 2013. Springer-Verlag.

[Yum] YumaWorks. Open Source NETCONF Implementation by YumaWorks. https://www.yumaworks.com/netconfd-pro/netconf/. Accessed: 2016-09-09.

[ZCZ+15] Kai Zheng, Zhiping Cai, Xin Zhang, Zhijun Wang, and Baohua Yang. Algorithms to speedup pattern matching for network intrusion detection systems. Comput. Commun., 62(C):47–58, May 2015.

[Zha13] Ying Zhang. An adaptive flow counting method for anomaly detection in sdn. In Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, pages 25–30, New York, NY, USA, 2013. ACM.

[ZSS+04] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, IMC '04, pages 101–114, New York, NY, USA, 2004. ACM.