\chapter{Design and Implementation}

In this chapter, the design and implementation of a programmable event processing framework within the Open vSwitch is presented. As a preview to the design the problem statement of the thesis is revisited, and the goals of the implementation are established in section 4.2. The rationale for an Open vSwitch implementation and the overview of the Open vSwitch is presented in sections 4.1 and 4.3. The breakdown of the design and the model of the envisioned system is expressed in section 4.4 and 4.5 respectively. An high-level implementation walk-through is detailed in section 4.6.

Before diving into the design and implementation of the system, it is important to revisit the problem statement and define the goals of the thesis. As outlined in Sections 1.1 and 1.2, the contribution of the thesis is in the area of event based systems which rely on high streaming data to detect and compose events to be processed. Typically such systems deal with massive streams of data arriving at high rates from a vast variety of nodes - be it sensory nodes or regular computational devices. In this Chapter a novel way of offloading the handling of such data onto the underlying network is presented. As outlined in section 1.2, offloading aspects of event processing onto the network can help in reducing the burden of computation on the event processing engines by detecting events b and taking appropriate actions before they enter the event processing engine. In order to create such a solution, the first question that needs to be answered is where in the network can aspects of an event processing engine be offloaded to? The implementation in this thesis focuses on the virtual switch.

With the advent of network and server virtualization environments, virtual switches (vSwitch) have increasingly become the mainstay deployments in all kinds of processing platforms. From large scale data centre networks to smaller Fog deployments, vSwitches allow for communication between different virtualized nodes of a host providing intelligent L2 routing with the aim of abstracting physical switches into an easily manageable logical switch. Since a vSwitch is a purely software solution - embedded or otherwise - it is much easier for administrators to manage and roll-out new functionalities on the switch. Additionally vSwitches are also deployed on the edge of a network. For these reasons the implementation in this thesis focuses on offloading aspects of event processing on to a vSwitch. Among the different implementation of vSwitches that are available for researchers today, we focus on the Open vSwitch as the core of our implementation. That leads to the next section, why Open vSwitch?

\section{Why Open vSwitch?}

Open vSwitch is a production quality switch built for multi-server virtualization environments with highly dynamic nodes. In 2014, an OpenStack survey \cite{OpenStack14} reported that nearly 40\% of production deployments used Open vSwitch, and by 2017 this number had increased to 60\% \cite{OpenStack17}. In addition to large scale deployments, Open vSwitch has gained popularity in emerging Fog deployments as discussed in sections 3.2 and 3.4. Another key factor in this design is the nature of event processing systems themselves which consist of a rules engine to create new rules based on user logic. Since Open vSwitch supports Openflow, a controller can be used to create new event-based rules akin to an event processing system. The figure 4.1 illustrates a take on Open vSwitch performing the role of an event processing engine. Here, a RYU based controller is used to configure event based rules on the Open vSwitch.

\begin{figure}[H]

\centering

\caption{Open vSwitch viewed as an Event Processing engine}

\includegraphics[height=8cm,width=14cm]{evsep01.pdf}

\end{figure}

\section{Goals of the implementation}

Goals of the implementation are defined as below:

\begin{itemize}

\item An Open vSwitch implementation is provided to detect event types and steer them to facilitate staged processing without having to context switch into an intermediate compute node.

\item The Open vSwitch implementation focuses on the DPDK datapath to take advantage of the accelerated packet I/O offered by the DPDK libraries.

\item The Open vSwitch is enabled to execute simple logical operations on the data items of an event and filter based on application logic.

\item A Ryu controller based implementation is provided to enable applications to offload user logic on to the vSwitch using a http and JSON based northbound API.

\end{itemize}

Having defined the goals of the thesis, it is important to understand the internals of the Open vSwitch and examine design decisions that need to be taken to achieve the goals outlined.

\section{An overview of Open vSwitch}

In this section a brief overview of the three most important components in the Open vSwitch \cite{pfaff2015design}

\begin{itemize}

\item \textbf{ovs-vswitchd}: ovs-vswitchd is the generic userspace daemon which speaks Openflow and controls all the switches in a system. The ovs-vswitchd daemon is responsible for the Openflow pipeline for packet processing and also determines how a packet has to be handled for the first time before it is cached in the datapath. The ovs-vswitchd daemon instructs the datapath on how to handle the packet depending on the configurable Openflow processing pipeline. The daemon communicates with the Open vSwitch kernel module via the netlink interface, and with the ovsdb-server using the custom ovsdb management protocol.

\item \textbf{ovsdb-server}: The ovsdb-server provides the runtime for the lightweight ovsdb database that holds switch level configurations. On reboot, the switch configurations are persisted in the ovsdb and are applied when the daemon comes back up. Currently, ovsdb supports state information in the form of 13 different tables for bridges, ports, interfaces, flow tables, controllers etc among others.

\item \textbf{Kernel Datapath}: The kernel datapath or the kernel module is the host operating system implementation of an exact match cache. It is unaware of Openflow or the state of the switch. It is kept simple for highly performant packet lookup and forwarding. The ovs-vswitchd receives the first packet from the datapath module, looks up the packet in the Openflow processing pipeline, gathers the actions and caches it in the kernel datapath. The following packets of the same flow are forwarded using the caches entries in the datapath without having to go to the userspace daemon(upcall).

\end{itemize}

In addition to the three components is the Openflow controller. The controller keeps the state of the network and can be installed in the same server or in a remote server. The controller speaks Openflow with the ovs-vswitchd daemon and can be used to remotely configure the Openflow processing pipeline of several remote switches. The controller may also be used to persist flow of the ovs-vswitchd daemon in case of a crash. The overview of these components is shown in figure 4.2 - sourced and reinterpreted from \cite{pfaff2015design}.

In addition to the above key components, the implementation provides several management utilities:

\begin{itemize}

\item \textit{ovs-vsctl}: This utility is used to configure the ovsdb-server with several switch configurations such as adding or deleting bridges, adding/deleting ports etc. The utility connects directly to the ovsdb-server -

\item \textit{ovs-ofctl}: This utility is used to monitor and configure the Openflow switches. It connects directly to the ovs-vswitchd daemon via Openflow and provides commands to add new flow rules to the Openflow pipelines among several other commands to monitor the flow statistics for the switches and ports.

\item \textit{ovs-dpctl}: This utility connects to ovs-vswitchd daemon and provides commands to monitor the cached datapath flows. It offers several commands such as add/delete flows to/from the datapath, monitor statistics etc among several others.

\item \textit{ovs-appctl}: This utility is a used for runtime configuration of the ovs-vswitchd daemon, such as configuring logging levels of the different modules within Open vSwitch.

\item \textit{ovsdb-tool}: This utility is used to manage the ovsdb files. It offers commands to create db schemas and compact existing schemas. It does not interact with the ovsdb-server.

\item \textit{ovsdb-client}: This utility offers a command-line interface to interact with the running ovsdb-server using the different json-RPC methods specified by ovsdb management protocol. It offers commands to run trasanctions such as insert, delete in to the ovsdb, dump tables, monitor columns, etc among others.

\end{itemize}

In the context of the thesis, most of the work focuses on the ovs-vswitchd daemon and the Openflow processing pipeline. In the following section the internal flow of data once a packet hits the physical device is presented.

\begin{figure}[H]

\centering

\caption{OVS components}

\includegraphics[height=10cm]{kernaldatapath.pdf}

\end{figure}

The interplay of the components of the Open vSwitch is shown in figure 4.3. The two interfaces that specify the handling of the packets are:\newline

\textbf{Datapath interface}: The datapath in the context of Open vSwitch is a flow cache capable of forwarding packets through a port. The datapath of an Open vSwitch is implemented specifically to the host machine and it relies on its clients for intelligence. In this case, the intelligence for the datapath is provided by the ovs-vswitchd. The generic datapath interface provided by Open vSwitch is dpif. The userspace implementation of the datapath interface is provided by the \textit{dpif-netdev} module. The dpif provides an interface with abstractions for three key components:

\begin{itemize}

\item Ports: Each datapath has a set of ports akin to Ethernet ports. In case of the dpdk or userspace datapaths, the implementation for these ports comes from \textit{struct netdev}, with \textit{netdev-dpdk} specifically handling dpdk ports.

\item Flow table: A flow table has three key entities: flow - which consists of L2,L3, and L4 headers. The datapath flow uses the \textit{struct dpif-flow}; mask - corresponding to each bit in a flow. The mask field is used for the megaflows, and not for the exact match cache; actions - which tell the datapath what to do with the packet.

\item Upcalls: The datapath notifies the clients, which is the ovs-vswitchd in this case, about a flow table miss or about userspace actions using upcalls. During the upcall, the whole packet is copied to the userspace using the struct \textit{dp\_packet}.

\end{itemize}

\begin{figure}[H]

\centering

\caption{OVS overview}

\includegraphics[height=6cm]{internals.pdf}

\end{figure}

\textbf{Ofproto Interface}: The ofproto interface implements the Openflow protocol implementation for the the dpif-netdev interface. It consists of three major components:

\begin{itemize}

\item oproto-dpif: This module implements the main provider of the Openflow abstraction. It is responsible for installing and removing datapath flows, monitoring and maintaining statistics etc. The ofproto.dpif module also is responsible for deciding whether a flow is to be cached or not.

\item ofproto-dpif-upcall: This module performs the role of retrieving the upcalls from the datapath. It performs simple processing of missed upcalls and interacts with ofproto-dpif for complex processing.

\item ofproto-dpif-xlate: This module performs the role of looking up the Openflow processing pipeline, extracting Openflow actions and translating into datapath actions.

\end{itemize}

\section{Design Breakdown}

Before design and implementation of the event processing framework within Open vSwitch, a brief breakdown of the design and rationale is provided. The design details are presented in the later sections. Key aspects that requires our focus are:

\begin{itemize}

\item Extraction of events from the payload: While work done by Mekky et. al \cite{mekky2014application} focus on creating Openflow actions to direct packets into a separate application module to work on the payload, the work in this thesis focuses on extracting the payload at line rate. The rationale behind this approach is that event queries are applied on each and every packet in the system; and directing the packets to a separate application module is expensive when the same action has to be repeated for every single packet.

\item De-serializing the event payload: The data items in the event stream are encoded. In the thesis a solution is provided for de-serializing hexadecimal encoded streams of data.

\item Accessing the event data: Once the event items are extracted and encoded from the event stream, they must be made accessible for processing within the Open vSwitch and also be accessible by Openflow rules that are installed from the controller. Due to this reason, the design focuses on modifying the Openflow 1.0 protocol spoken between the controller and the Open vSwitch. Although Openflow 1.3 is the most standard protocol in use, the work in the thesis uses Openflow 1.0 for simplicity. Porting from Openflow 1.0 to 1.3 is not considered to be a blocking factor.

\begin{figure}[H]

\centering

\caption{OVS dataflow: datapath to ofproto}

\includegraphics[height=12cm]{dpnetdev.pdf}

\end{figure}

\item Openflow pipeline support for events: For making the event detection programmable, additional tables are created within the Openflow processing pipeline, or the ofproto classifier. Handling the ofproto classifier is sufficient because ofproto-dpif takes care of creating the tables in the datapath classifier/exact match cache when needed.

\item Creating keys for event attributes: The \textit{odp-netlink} interface in Open vSwitch specifies the rules for creating attribute keys. These keys are used to look up the ofproto classifier and find the relevant rule installed for an event.

\item Support for logical operations: The operations on event attributes are performed using Openflow actions. Hence each operation is achieved using a newly defined action, which is triggered based on the detection criteria specified on event items.

\item RYU Controller design: The controller plays a big role in communicating Openflow rules to the Open vSwitch. Althought ovs-ofctl utility can also play this part, a controller such as RYU is more sophisticated, in that it allows creation of http API's that are accessible by remote webservers to create rules. Such an API support is provided which enables remote creation and deletion of event based rules. The RYU controller plays the role of the rules engine, whereas Open vSwitch plays the role of the execution engine.

\end{itemize}

The design overview presented above aims to create a sophisticated programmable event processing framework on the network, which can complement the event processing engines to achieve latency sensitive results, reduce the burden of computation and enable staged processing. In the coming sections, more focus is granted up on individual design decisions.

\section{System Model}

In this sections the supported semantics for event processing within the Open vSwitch is presented. Each event follows the below representation:\\1. Unique Tag\\

2. Event Type\\3. Event Attributes

To this end the event payload format processed by the framework within Open vSwitch is pre-defined. Each event has a unique tag, followed by the event type and two event attributes. Only the packets with the unique tag are processed through the event processing pipeline. In other cases, the packets are taken through the standard processing pipeline within Open vSwitch. The event payload handled by the event processing engine is show in figure 4.5. The timestamp in the event is used for evaluation purposes.

\begin{figure}[H]

\centering

\caption{Event Payload}

\includegraphics[width=10cm]{payload.pdf}

\end{figure}

\subsection{Event Detection Semantics}

Let 'E' be an event of type 't' and attributes '$a\_1$' and '$a\_2$'.

\begin{equation}

E\lbrace t,a\_1,a\_2 \rbrace

\end{equation}

The event detection semantic understood and processed by the framework are described below:

\begin{flushleft}

Event instances can be detected based on different variants.

\\1. Detect based on Event Type

\\2. Detect based on values in Event Attributes

\\3. Detect based on combination of Event Type

\\

\end{flushleft}

\begin{flushleft}

Event detection operations can be combined with logical operators:

\\

\textbf{Conjunction}: Event is signalled when all constituent input patterns evaluate to True.

\\

\textbf{Disjunction}: Event is signalled when atleast one constituent input pattern evaluates to True.

\\

\end{flushleft}

\begin{flushleft}

The detect operations that are implemented are notated as:

\begin{equation}

D(e.t) \quad | \quad stream

\end{equation}

\begin{equation}

D(e.t) \quad | \quad filter

\end{equation}

\begin{equation}

D(e.t \wedge e.a\_1) \quad | \quad stream

\end{equation}

\begin{equation}

D(e.t \wedge e.a\_1) \quad | \quad filter

\end{equation}

\begin{equation}

D(e.t \wedge e.a\_2) \quad | \quad stream

\end{equation}

\begin{equation}

D(e.t \wedge e.a\_2) \quad | \quad filter

\end{equation}

\begin{equation}

D(e.t \wedge (e.a\_1 \wedge e.a\_2)) \quad | \quad stream

\end{equation}

\begin{equation}

D(e.t \wedge (e.a\_1 \wedge e.a\_2)) \quad | \quad filter

\end{equation}

\begin{equation}

D(e.t \wedge (e.a\_1 \vee e.a\_2 )) \quad | \quad stream

\end{equation}

\begin{equation}

D(e.t \wedge (e.a\_1 \vee e.a\_2 )) \quad | \quad filter

\end{equation}

where \textit{D} is the detect operation; \newline

\textit{|} denotes the redirect operation; \newline

\textit{stream} is the logical stream to which the detected event is redirected to.\newline

\textit{filter} is the operation to filter the detected event \newline \newline

\end{flushleft}

\subsection{Compare Operation Semantics}

In addition to event detection based on match of event attributes, compare operations on event attributes are implemented. The set of implemented compare operators is given by:

\begin{equation}

\oplus \ni \lbrace <=,>= \rbrace

\end{equation}

and the possible operations using the compare operators is given by:

\begin{equation}

D(e.t \wedge (e.a\_1 \oplus value) \vee e.a\_2) \quad | \quad filter

\end{equation}

\begin{equation}

D(e.t \wedge (e.a\_1 \oplus e.a\_2) \quad | \quad filter

\end{equation}

\subsection{Stateful Operation Semantics}

Stateful operations in the context of the implementation are defined as those that use the knowledge of previous events to take an action on the currently detected event. The two stateful operations implemented are moving maxima and window.

\begin{equation}

D(e.t \wedge (e.a\_1 <=\quad \rightarrow maxvalue) ) \quad | \quad filter

\end{equation}

In this operation all the events with attribute lower than the given \textit{maxvalue} are filtered. However, on the detection of an event with a value higher than the given \textit{maxvalue}, the event is forwarded and the newly seen value becomes \textit{maxvalue}.

\begin{equation}

D(e.t \wedge (e.a\_1 <=\quad \overrightarrow{win} \quad maxvalue)) \quad | \quad filter

\end{equation}

Similar to 4.15, all the events with attribute lower than the given \textit{maxvalue} are filtered. However, on the detection of an event with a value higher than the given \textit{maxvalue}, the event is forwarded and the newly seen value becomes \textit{maxvalue}. The \textit{maxval} increases for a window of \textit{win} events after which it remains steady. if the window is 0, this is equivalent to less than or equal to operation defined in 4.14.

\section{Implementation walk-through}

In this section, the key aspects of implementing a programmable event processing framework within an Open vSwitch are discussed. The implementation aims to meet the specifications defined in the system model in section 4.5 and follows the design strategy discussed in section 4.4. Wherever appropriate the implementation is discussed along with data or control flow diagrams, snippets of code or data structures used within the implementation.

\subsection{Event extraction and de-serialization}

In this section, the methodology for event parsing and extraction is presented. Before chalking up a strategy, the flow of data in the DPDK/userspace datapath is examined more closely in figure 4.4. As we can see from the figure the packet is handed over to the ofproto interface in its entirety using the data structure \textit{dp\_packet}. But if the event data is extracted only after passing it to ofproto, this would mean that any support of the exact match cache or the datapath classifier that can be provided by the DPDK datapath to the event processing extension is ruled out. Hence the parsing and extraction of the event payload is done within the dpif-netdev implementation using the flow module. Once the event is extracted it is de-serialized. Both the steps are done within the flow module of the Open vSwitch.

\begin{figure}[H]

\centering

\caption{Event extraction and access}

\includegraphics[height=12cm]{flowextract01.pdf}

\end{figure}

\subsection{Accessing event data}

Once the data items are extracted and de-serialized, they are be made accessible to the implementations of both the ofproto interface and datapath interface. The flow module of the Open vSwitch specifies the data structures for the Openflow fields, which are accessible by both the userspace dpif-netdev and ofproto implementation. Additionally, since the extraction and de-serialization of the event data is performed within the flow module, the support for event data items is added into the userspace flow data structure \textit{struct flow}. Doing so ensures that the event data is accessible to the ofproto and the datapath interface, and also to the yet to be discussed meta-flow module which specifies the interface for the Open vSwitch implementation of Openflow 1.0 tables. Figure 4.6 shows the interplay of the modules while extracting, de-serializing and pushing the data to the relevant data structure for access within other modules. The flow module also provides another data structure \textit{struct flow\_wildcards} which follows the same structure as flow, except that each 1-bit in wildcard indicates that the corresponding bit in \textit{struct flow} must match. For the event data items, only an exact match is supported. Hence all the bits of the struct \textit{flow\_wildcards} for event data items are set. Below snippet shows the addition of support for event data in struct flow and the map used to access the flow by other interfaces. \newline

\begin{lstlisting}[language=c]

struct flow {

..

..

..

/\* L4 (64-bit aligned) \*/

ovs\_be64 e\_type;

ovs\_be64 e\_attr1;

ovs\_be64 e\_attr2;

..

..

}

FLOWMAP\_SET(map, e\_type);

FLOWMAP\_SET(map, e\_attr1);

FLOWMAP\_SET(map, e\_attr2);

miniflow\_push\_be64(mf, e\_type, htonll(event[0]));

miniflow\_push\_be64(mf, e\_attr1, htonll(event[1]));

miniflow\_push\_be64(mf, e\_attr2, htonll(event[2]));

\end{lstlisting}

\begin{figure}[H]

\centering

\caption{Openflow Pipeline}

\includegraphics[height=8cm,width=19cm]{pipeline05.pdf}

\end{figure}

\subsection{Modelling Openflow pipeline for event processing}

As briefly alluded to in the section 4.6.2, the event items are to be added into the Openflow processing pipeline. The specification of Openflow 1.0 protocol is available within Open vSwitch in the Openflow-1.0 modules. The meta-flow module defines the interface to implement the Openflow processing pipeline, i.e the fields of the ofproto classifier. Before diving into the implementation details, we examine the ofproto classifier pipeline in figure 4.7. Openflow supports pipelined processing of packets with the help of multiple tables which in turn have multiple flow rule entries. In figure 4.7, the incoming packet at the ingress port is matched at Table 0, with the action of rewriting the mac destination address is added to the action-set and submitted to Table 16; At Table 16, the packet is matched on it's mac destination address, and the action of re-writing its IP destination address is added to action-set is performed and it is forwarded to Table 17; At Table 17, the packet is matched by a flow rule for its IP destination address, and the action of re-writing its Transport destination address is added to the action-set and the packet is forwarded to Table 21; At Table 21 the packet is matched by a flow rule for its Transport destination address, and checked for GO-TO instruction in rule action. Since no GO-TO instruction is found, the pipeline processing stops and the action-set is executed on the packet and the packet is forwarded through the egress port. For the sake of simplicity, the focus in this thesis document will be on a single Openflow table. Although enabling one Openflow table to handle event data is sufficient to create an Openflow pipeline for event data, such an use case is not helpful initially. Figure 4.8 shows an individual flow table and the steps taken to enable event fields within it.

\begin{figure}[H]

\centering

\caption{Openflow Flowtable - Add event data fields}

\includegraphics[height=5cm]{flowtable.pdf}

\end{figure}

\subsection{Adding event data fields to Openflow tables}

As discussed in section 4.6.2 the data structure flow is modified to handle event data and provide access of the event data to ofproto interface. Predictably, similar support is to be added to the Openflow table using the meta-flow module as shown in figure 4.8. Below is the snippet which shows the creation of the event type as an Openflow field within the meta-flow module. \newline

\begin{lstlisting}

/\* "e\_type".

\*

\* Event Type.

\*

\* Type: be64.

\* Maskable: bitwise.

\* Formatting: decimal.

\* Prerequisites: UDP.

\* Access: read/write.

\* NXM: NXM\_OF\_EVNT\_TYP(113) since v2.6.

\* OXM: OXM\_OF\_EVNT\_TYP(46) since OF1.2 and v2.6.

\* OF1.0: exact match.

\* OF1.1: exact match.

\*/

MFF\_EVNT\_TYP,

\end{lstlisting}

The comments for the meta-flow field are expected to be in the exact format specified for the python helper function extract-ofp-fields. Failing to do so results in build breakdown. In addition to adding fields in the meta-flow module, the fields are provided with necessary helper functions which are accessed by ofproto and ofp-parse interfaces while responding to flow-mod or flow stats request commands from the controller. One such helper function defined to set e\_type from the corresponding met-flow field is shown below. This function is used to read the meta-flow field and populate match before dumping the buffer for dump-flow commands. Similarly, other helpers are defined. \newline

\begin{lstlisting}[language=c]

void

mf\_set\_value(const struct mf\_field \*mf,

const union mf\_value \*value, struct match \*match, char \*\*err\_str)`{

..

..

case MFF\_EVNT\_TYP:

match\_set\_event\_type(match, value->be64,OVS\_BE64\_MAX);

break;

..

..

}

\end{lstlisting}

\subsection{Adding support for event based rules}

Adding event fields in the meta-flow module enables support for event data in the Openflow table. It however is not sufficient to enable event based rules and detection of events based on extracted data. This is done in two stages

\begin{itemize}

\item Provide support to event fields in add-flow command of ovs-ofctl utility to send the relevant data in Openflow message. This command is also used by the controller. More about the controller implementation is discussed later.

\item Add support to decode event fields from the incoming Openflow message and insert the event based rules into the classifier.

\end{itemize}

Before diving into the details, the important interfaces are examined:

\begin{itemize}

\item \textbf{ofp-parse interface}: The ofp-parse interface specifies the layer of communication between ovs-vswitchd and Openflow. In case of the flow-mod commands(OFPC\_ADD), ofp-parse module converts the string value pairs of the event fields and inserts them into the match field of data structure \textit{struct ofp\_util\_flow\_mod}.

\item \textbf{ofp-util interface}: The ofp-util interface defines the the \textit{struct ofp\_util\_flow\_mod} to consume and respond to the flow-mod commands. The struct \textit{struct ofp\_util\_flow\_mod} is composed of a struct match as a member.

\item \textbf{match interface} The match interface provides the data structure for flow classification - \textit{struct match}. Each match structure consists of a flow and a corresponding wildcard. The ofp-parse and ofp-util interface combine together to extract the string-value pairs in a flow-mod command into relevant flows and wildcards within a \textit{struct match}.

In addition to creating event based flow rules, functionality is also implemented to dump event based flow rules on to the buffer as in case of dump-flows command. For this operation, within the match module, helper functions to get meta-flow event fields into the \textit{struct match} are also defined. This is used by the ofp-parse interface to populate \textit{struct match} and hand it over to nx-match interface for encoding. An example of such an helper function is shown in the below snippet: \newline

\begin{lstlisting}[language=c]

void

match\_set\_event\_type\_masked(struct match \*match, ovs\_be64 e\_type, ovs\_be64 mask){

match->flow.e\_type = e\_type & mask;

match->wc.masks.e\_type = mask;

}

\end{lstlisting}

\item \textbf{nx-match module}: The flow and the wildcards from the match interface are encoded as fields defined by the meta-flow module using the nicira extension helper functions specified by the nx-match module. Doing so enables dump-flow support to event-based rules.

\end{itemize}

Figure 4.9 shows how a OFPFC\_ADD command with event fields is parsed and normalized into the match interface using \textit{struct ofputil\_flow\_mod}. To enable correct parsing of the event fields, Openflow 1.0 support is configured in Openflow-1.0 header file: \newline

\begin{lstlisting}[language=c]

enum ofp10\_flow\_wildcards {

..

OFPFW10\_EVNT\_ATTR1 = 1 << 22,

OFPFW10\_EVNT\_ATTR2 = 1 << 23,

OFPFW10\_EVNT\_TYP = 1 << 24,

..

}

\end{lstlisting}

This informs the parser that the relvant event fields are in the 22, 23 and 24 position of the character string. After which the relevant event fields are populated and normalized. Normalization ensures that the corresponding wildcards for the match flow entries are appropriately set. Once normalized the flow-mod command is encoded as a OFPT\_FLOW\_MOD message and send to the ovs-vswitchd daemon which actively polls for Openflow messages.

\begin{figure}[H]

\centering

\caption{Openflow: Parse event fields}

\includegraphics[height=10cm,width=14cm]{flowadd.pdf}

\end{figure}

Figure 4.10 shows an Openflow message of type OFPTYPE\_FLOW\_MOD is parsed to extract the relevant rules and added into the classifier. The ovs-vswitchd daemon actively polls for Openflow messages. On receiving a message, the message type is decoded. If the message is of type 'OFPTYPE\_FLOW\_MOD' the message is decoded by a decode function. The first phase of the decode process involved pulling the message which is in the custom Openflow buffer, ofpbuf, into the Openflow1.0 match structure struct ofp10\_match. This structure is also extended to support event fields, very much like struct flow: \newline

\begin{lstlisting}[language=c]

struct ofp10\_match {

..

ovs\_be64 e\_attr1;

ovs\_be64 e\_attr2;

ovs\_be64 e\_type;

}

\end{lstlisting}

Once the \textit{struct op10\_match} is created, the relevant event fields are extracted and put into a \textit{struct match} and normalized. The ofpbuf is not directly pulled into \textit{struct match} because the match interface is inaccessible outside the context of ovs-vswitchd. Instead the ofp-util interface is used to convert the data Openflow structures to a form understandable and accessible by the Openflow implementation within Open vSwitch, i.e ofproto. Once the rules are stored in the form of a match, the ovs-vswitchd daemon calls the ofproto interface to add the rules into the classifier.

\begin{figure}[H]

\centering

\caption{Openflow: Event rule creation}

\includegraphics[height=12cm,width=14cm]{ruleinsert.pdf}

\end{figure}

At this point the implementation supports programmable event rules and support form the three event fields specified by the system model. However the event detection mechanism is yet to be implemented. In the next subsection, a closer look into the implementation steps needed to support matching based on event type and event attributes is presented.

\subsection{Detection based on event data}

In this section, the implementation process to support classifier lookup based on event fields. In section 4.6.2, event data is pushed into the flow structure, and in section 4.6.3 to 4.6.5, event rule support is added to the Openflow processing tables- i.e ofproto classifier. To look up the classifier with the event data items, event keys are extracted from the flow. Figure 4.11 illustrates the lookup process followed by the ofproto-dpif-xlate module.

\begin{figure}[H]

\centering

\caption{ofproto: Rule look-up}

\includegraphics[height=5cm]{ofproto-dpif-xlate.pdf}

\end{figure}

Userspace keys to support the lookup are added to odp-netlink. In the below snippets, the set up steps for e\_type key are shown. The snippet focuses on the key steps to be taken to enable keys and excludes the details, which can be found in the source code. \newline

\begin{lstlisting}[language=c]

enum ovs\_key\_attr {

..

#ifndef \_\_KERNEL\_\_

/\* Only used within userspace datapath\*/

OVS\_KEY\_ATTR\_EVNT\_TYP,

#endif

..

};

struct ovs\_key\_etype {

ovs\_be64 e\_type;

};

\end{lstlisting}

and the keys are parsed and extracted to be used by ofproto-dpif. \newline

\begin{lstlisting}[language=c]

/\* parse\_l2\_5\_onward \*/

if (!is\_mask) {

..

..

expected\_attrs |= UINT64\_C(1) << OVS\_KEY\_ATTR\_EVNT\_TYP;

}

if (present\_attrs & (UINT64\_C(1) << OVS\_KEY\_ATTR\_EVNT\_TYP)) {

const struct ovs\_key\_etype \*etype\_key;

etype\_key = nl\_attr\_get(attrs[OVS\_KEY\_ATTR\_EVNT\_TYP]);

put\_etype\_key(etype\_key, flow);

}

/\* odp\_flow\_key\_from\_flow and odp\_flow\_key\_from\_mask\*/

struct ovs\_key\_etype \*etype\_key;

etype\_key = nl\_msg\_put\_unspec\_uninit(buf, OVS\_KEY\_ATTR\_EVNT\_TYP,

sizeof \*etype\_key);

\end{lstlisting}

At this stage the implementation supports event rules that implement operations notated through 4.2 to 4.11, by combining existing event semantics with drop and mod actions provided by Openflow. In Section 5.4.7, many of these operations are evaluated with their respective rule format supported by the RYU controller. The operations are also contrasted with their expression in a standard Event Query language.

\subsection{Enabling compare operations support}

As discussed in section 4.2, the thesis also aims to provide support for logical operations on event attributes. In section 4.5.2 the two compare operators {>=, <=} are outlined for implementation. In this section the implementation of the compare operations within Open vSwitch is presented. The operations are implemented as Openflow actions which are triggered after detecting a event type. The two compare operations are implemented as:

\begin{itemize}

\item \textbf{set\_max} : A \textit{set\_max:val} action ensures that any event with attribute value higher than \textit{val} is filtered out for a particular event type.

\item \textbf{set\_min} : A \textit{set\_min:val} action ensures that any event with attribute value lower than \textit{val} is filtered out for a particular event type.

\end{itemize}

The \textit{set\_max}, \textit{set\_min} actions provide implementation support for operations notated in 4.13. The implentation can be extended for any binary operaiton The code snippets illustrate how the new actions are added to the Openflow action list, with \textit{set\_max} action used for illustration. \newline

\begin{lstlisting}[language=c]

OFPACT(SET\_MAX,ofpact\_attr, ofpact, "set\_max")

/\* OFPACT\_SET\_MAX.

\*

\* Used for OFPAT10\_SET\_MAX.\*/

struct ofpact\_attr {

struct ofpact ofpact;

uint64\_t attr; /\* Attribute value. \*/

};

\end{lstlisting}

In addition implementation support is provided to parse, decode, encode and set attributes for actions. The implemetnation for the user actions is done in the ofproto-dpif-xlate module which as discussed before and illustrated in figure 4.11 is responsible for deriving actions and performing them. \newline

\begin{lstlisting}[language=c]

case OFPACT\_SET\_MAX:{

if(htonll(flow->e\_attr1) <= ofpact\_get\_SET\_MAX(a)->attr){

xlate\_normal(ctx);

}

break;

}

\end{lstlisting}

In addition to comparing attributes to values provided in event rules; another class of action is implemented which allow creation of rules to compare between the attributes themselves. This is done via:

\begin{itemize}

\item \textbf{attr\_cmp}: A \textit{attr\_cmp:mode} action compares the event attributes, and filters out the events based on the mode triggered by \textit{val}. if \textit{mode} is 0, only events with equal attributes is forwarded, rest are dropped. if \textit{mode} is 1, only events with attribute 1 greater than attribute 2 are forwarded, if \textit{mode} is 2, only events with attribute 2 greater than attribute 1 are forwarded. Here \textit{mode} triggers the operation to be used between attributes. Similarly implementations can be extended for any other binary operation.

\end{itemize}

The \textit{attr\_cmp} action provides the implementation of operations notated in 4.14.

\subsection{Enabling Stateful Operations}

In this section the implementation of stateful operations notated in section 4.5.3 is detailed. To implement these operations two actions are implemented:

\begin{itemize}

\item \textbf{mov\_max}: A \textit{mov\_max:maxval} action filters the event if the attribute is lower than \textit{maxval}. However if the attribute value higher than \textit{val} is encountered, the event is forwarded and the newly seen value becomes the new \textit{maxval}. This action provides implementation support to the operation notated in 4.15. Below the pseudo-code of the \textit{mov\_max} action is shown:\newline

\begin{lstlisting}[language=c]

case OFPACT\_SET\_MOV\_MAX:

mov\_max= ofpact\_get\_SET\_MOV\_MAX(a)->attr

item= search\_max(type)

if(item!=NULL):

cur\_max = item->data

if(attr1 > cur\_max):

delete\_max(item)

insert\_max(type, attr1)

else:

cur\_max = mov\_max

insert\_max(type, mov\_max)

if(attr1 < cur\_max):

break

else:

xlate\_normal(ctx)

break

\end{lstlisting}

\item \textbf{set\_win,win\_max}: The combination of \textit{set\_win:win,win\_max:maxval} actions performs a similar role done by the \textit{mov\_max}. However the \textit{maxval} is incremented only for the specified window \textit{win}, after which it remains constant. To understand this action, let us look at an example for event type 'TEMP', with action \textit{set\_win:10,win\_max:50}. Firstly all events of type 'TEMP' with attribute values lower than 50 are filtered. An event of type 'TEMP' and attribute value 60 is forwarded, and 60 becomes the new \textit{maxval}. And going forward, a 'TEMP' event with value 58 will be filtered. This increment of \textit{maxval} continues till the 10th window, or otherwise the 10th time the rule is hit. From thereon, the highest value of attribute seen within the defined window will remain the \textit{maxval}. This action provides implementation support to the operation notated in 4.16. The below pseudo-code glimpses through the implementation.

\begin{lstlisting}[language=c]

case OFPACT\_SET\_WIN:

window= ofpact\_get\_SET\_WIN(a)->attr

winItem=search\_window(type)

if(winItem == NULL):

insert\_window(type, window, 0)

else:

delete\_window(winItem)

insert\_window(type, window, ++(winItem->counter))

break;

case OFPACT\_SET\_WIN\_MAX:

mov\_max= ofpact\_get\_SET\_WIN\_MAX(a)->attr

winItem=search\_window(type)

item= search\_max(type)

if(>window <= counter):

if(item!=NULL):

cur\_max = item->data

else

cur\_max = mov\_max

else if(window > counter):

if(item!=NULL):

cur\_max = item->data

if(attr1 > cur\_max):

delete\_max(item)

insert\_max(type, htonll(flow->e\_attr1))

else:

cur\_max = mov\_max

insert\_max(type, mov\_max)

if(attr1 < cur\_max):

break

else:

xlate\_normal(ctx);

break

\end{lstlisting}

\end{itemize}

\subsection{API support for event rules via RYU}

In this section the implementation of an Openflow based event rules engine is presented. To implement support for event rules the RYU controller is modified. Figure 4.12 shows the flow of data within the RYU controller started from the API to the corresponding \textit{OFPT\_FLOW\_MOD} message.

\begin{figure}[H]

\centering

\caption{RYU: North to South - Event Rule creation}

\includegraphics[height=10cm,width=14cm]{ryu01.pdf}

\end{figure}

The implementation is broken down into four parts:

\begin{itemize}

\item \textbf{ofproto}: In the ofproto module all the event related fields are initialized and their positions in the Openflow message is set. The position set in this module mirror the Open vSwitch support for the event fields. In addition to this, the ofproto module also defines the packing structure the flow-mod message sent to Open vSwitch. In the below snippet, the mapping of event fields and actions between the controller and Open vSwitch is shown. \newline

\begin{lstlisting}[language=python]

\_OFP\_MATCH\_PACK\_STR = 'IH' + OFP\_ETH\_ALEN\_STR + 's' + OFP\_ETH\_ALEN\_STR + \

'sHBxHBB2xIIHHQQQQ'

OFPAT\_SET\_WIN\_MAX = 12 # window moving max value for attribute1.

OFPAT\_SET\_MOV\_MAX = 13 # moving max value for attribute1.

OFPAT\_SET\_MIN = 14 # min value for attribute1.

OFPAT\_SET\_MAX = 15 # moving max value for attribute1.

OFPAT\_SET\_WIN = 16 # window value

OFPAT\_SET\_CMP = 17 # compare mode

OFPFW\_EVNT\_ATTR1 = 1 << 22

OFPFW\_EVNT\_ATTR2 = 1 << 23

OFPFW\_EVNT\_TYP = 1 << 24

\end{lstlisting}

\begin{lstlisting}[language=c]

struct ofpact\_map of10[] = {

..

{ OFPACT\_SET\_WIN\_MAX, 12},

{ OFPACT\_SET\_MOV\_MAX, 13},

{ OFPACT\_SET\_MIN, 14},

{ OFPACT\_SET\_MAX, 15},

{ OFPACT\_SET\_WIN, 16},

{ OFPACT\_SET\_CMP, 17}

..

}

enum ofp10\_flow\_wildcards {

..

OFPFW10\_EVNT\_ATTR1 = 1 << 22,

OFPFW10\_EVNT\_ATTR2 = 1 << 23,

OFPFW10\_EVNT\_TYP = 1 << 24,

}

\end{lstlisting}

\item \textbf{nx-match}: In the nx-match module, the methods to serialize the match structure is defined with methods to put each event fields and the respective wild cards into the defined header fields This modules takes care of the serialization of the match fields and their wildcards. In the below snippet, the serialization process is show for the event type field. \newline

\begin{lstlisting}[language=python]

@\_register\_make

@\_set\_nxm\_headers([ofproto\_v1\_0.NXM\_OF\_EVNT\_TYP])

class MFEVNTTYP(MFField):

@classmethod

def make(cls, header):

LOG.debug('VLOG in point 76')

return cls(header, MF\_PACK\_STRING\_BE16)

def put(self, buf, offset, rule):

LOG.debug('VLOG in point 86')

return self.putm(buf, offset, rule.flow.e\_type, rule.wc.e\_type\_mask)

def serialize\_nxm\_match(rule, buf, offset):

if rule.flow.e\_type != 0:

if rule.flow.nw\_proto == 17 and rule.wc.e\_type\_mask == UINT64\_MAX:

header = ofproto\_v1\_0.NXM\_OF\_EVNT\_TYP

else:

header = 0

if header != 0:

offset += nxm\_put(buf, offset, header, rule)

\end{lstlisting}

\item \textbf{ofctl}: The ofctl modul consists of the implementation for the \textit{mod\_flow\_entry} method which handles the north-bound http enabled flowadd API. In this module, the event fields and actions are extracted from the json messages, reads them to previously defined match fields which are sent to the ofproto parser module to pack them into the desired format. \newline

\begin{lstlisting}

#read from json into e\_type string:value pair

match = dp.ofproto\_parser.OFPMatch(wildcards,.. e\_type,..) #call OFPMatch

#read from json into set\_max string:value pair

elif action\_type == 'SET\_MAX':

val = int(a.get('val', 0))

actions.append(dp.ofproto\_parser.OFPActionSetMax(val)) #call OFPActionSet

\end{lstlisting}

\item \textbf{ofproto-parser}: The ofproto-parser module is modified to taking the match and action strings containing event related messages and convert them into Flow and Action structures supporting the relevant event fields and actions. The ofproto-parser module packs relevant structures using the definition provided in the ofproto module and then serializes the message using the methods provided in nx-match modules. The message sent out to the Open vSwitch is an \textit{OFPT\_FLOW\_MOD} message which is defined in position 14 within the Open vSwitch, and hence is packed similarly in RYU. \newline

\begin{lstlisting}[language=python]

OFPT\_FLOW\_MOD = 14 # Controller/switch message

\end{lstlisting}

In Open vSwitch:

\begin{lstlisting}[language=c]

enum ofpraw {

/\* OFPT 1.0 (14): struct ofp10\_flow\_mod, uint8\_t[8][]. \*/

OFPRAW\_OFPT10\_FLOW\_MOD}

\end{lstlisting}

The below snippet, shows the methods added to parse and serialize individual event \textit{set\_min} action. \newline

\begin{lstlisting}[language=python]

class OFPActionAttrVal(OFPAction):

def \_\_init\_\_(self, val):

super(OFPActionAttrVal, self).\_\_init\_\_()

self.val = val

@classmethod

def parser(cls, buf, offset):

type\_, len\_, val = struct.unpack\_from(

ofproto.OFP\_ACTION\_ATTR\_VAL\_STR, buf, offset)

assert type\_ in (..,

ofproto.OFPAT\_SET\_MIN,

..)

assert len\_ == ofproto.OFP\_ACTION\_ATTR\_VAL\_SIZE

return cls(val)

def serialize(self, buf, offset):

msg\_pack\_into(ofproto.OFP\_ACTION\_ATTR\_VAL\_STR,

buf, offset, self.type, self.len, self.val)

@OFPAction.register\_action\_type(ofproto.OFPAT\_SET\_MIN,

ofproto.OFP\_ACTION\_ATTR\_VAL\_SIZE)

\end{lstlisting}

\end{itemize}

\subsubsection{Enable Northbound API}

In addition to changes in the differnent modules, the Northbound API support is provided to the RYU application by mapping the /flowentry/add command to the mod\_flow\_entry method within ofctl. The format of the json and the usage to add event based rules into the Open vSwitch is discussed in detail along with examples in the next chapter. \newline

\begin{lstlisting}[language=python]

uri = path + '/flowentry/{cmd}'

mapper.connect('stats', uri,

controller=StatsController, action='mod\_flow\_entry',

conditions=dict(method=['POST']))

\end{lstlisting}

\section{Summary}

In this chapter the design and implementation of a programmable event processing framework within Open vSwitch with the RYU controller used as the rules engine was presented. The chapter characterized the goal of the thesis, presented a design breakdown and depicted a system model for events. Three classes of event semantics were expressed within the system model after which the steps taken to implement the system was recounted with necessary illustrations of data flow and snippets of code necessary for fair representation. In the next chapter, the evaluation of the system across several parameters is chronicled with a brief discussion of relevant results.