In-Network Complex Event Processing

Masterarbeit
Advith Belegadde Nagappa
Tag der Einreichung:
1. Gutachten: Marcel Blöcher, M. Sc
2. Gutachten: Prof. Dr. Patrick Eugster, Ph.D

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Distributed Systems Programming

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 Advith Belegadde Nagappa

Contents

List of Figures

List of Tables

Abstract

In today's ubiquitous computing environment, the demands to filter data from the noise and react to patterns of events expeditiously has intensified the burden put on the already complex event processing ecosystem. The complex event processing ecosystem is composed of several contributing technologies working in synergy to afford the needed processing for computation and communication as events progress through the system achieving higher levels of abstractions with producers and consumers at each level. The ecosystem can be viewed as an elaborate overlay on top of the underlying network. The thesis proposes a paradigm of viewing the underlying network as a contributing technology to the complex event processing ecosystem by offloading event processing application context onto the network. With the advent of software defined networking and its' consequent separation of the control and data planes, the possibilities to tune network services to the bidding of user applications are immense. Whereas network function virtualization aspires to virtualize diversified network hardware into effortlessly serviced software solutions provisioned on commodity servers, the thesis aims to research the implications of moving event processing application context on to virtualized network components. The hypothesis to begin with is that offloading of application context onto the underlying network allows network architects to tune their services to latency sensitive applications. It also provides better insights into the traffic characteristics of the application which may be used for designing better staged processing, load balancing and reducing the overall clog on the network I/O stack. Finally, it presents network operators with opportunities to explore further monetization of the network and virtualization infrastructure and pro-actively scrutinize revenue models which blur the line between application land and the network.

1 Introduction

With the explosion of connected devices in today's ubiquitous computing environment, a whole new class of data-intensive applications have emerged which demand data processing at high-input rate. Such applications process unbounded volume of data arriving at a high rate. While having unbounded volume of data may have it's merits, it also means that applications have to evolve a different data-persistence strategy and be efficient in filtering out the signal from noise. Traditional query processing systems which issue a single non-persistent query on persistent data are unsuitable to live up to the demands of such applications. This led to the development and adoption of continuous query [CDTW00] and Stream Processing [CQ09] paradigms which in contrast apply persistent queries to streams of incoming data. Here a query is long-running and is evaluated continually against incoming data, and only the data which satisfies the conditions of the query is selected for further processing. Cugola et al. [CM12] classify Complex Event Processing (CEP) as an important characteristic of the Stream Processing paradigm. Being able to detect patterns in streams of data and trigger necessary user or application defined events is a key requirement of stream processing engines. Complex Event Processing which evolved exactly for this purpose, offers solutions for real-time detection of patterns in data streams and triggering of appropriate actions based on the user or application logic.

Because of the scale of modern day IT systems, large scale distributed CEP deployments like many other distributed applications are hosted on commodity servers. Although there is an increasing trend of taking advantage of Fog Cloud computing resources [BMZA12] for high velocity, high variety streaming applications, the interplay between the conventional cloud and Fog cloud is expected to increase as the demand for meaning and analysis increases; making Fog localization a supplement to a global cloud deployment. In the midst of all this enters into fray a Software Defined approach to Networking [[JKM$^+$13], [CFP$^+$07]] which aims to separate the control plane of the underlying network from the data plane, thereby enabling rapid deployment of network services and paving way for deploying network services as virtual functions within the virtualized network stack [[SGY$^+$09],[HGJL15]] and eventually to service function chaining [HP15].

Emergence of Network Functions Virtualization is symptomatic of the advances in multi-core commodity server architectures and high speed network cards. It allows network architects to leverage virtualization technologies to implement network functions and run it on commodity hardware instead of using dedicated network appliances. These virtual functions can be provisioned on-demand without the need for installing expensive equipment. Network Functions Virtualization and Software Defined Networking provides network operators with immense opportunities to monetize the underlying infrastructure. With user applications and network functions running within virtualized containers, a valid research question would be how much of the application context can the Network Functions be made aware of to extract greater performance of the application without disaffecting the network service. In this thesis, an attempt is made to ask the question in the context of Complex Event Processing. The latency-sensitive characteristic of a CEP application combined with high volume, high velocity, and low signal-to-noise ratio characteristics of the arriving data make it an optimal target for offloading certain application context onto the network.

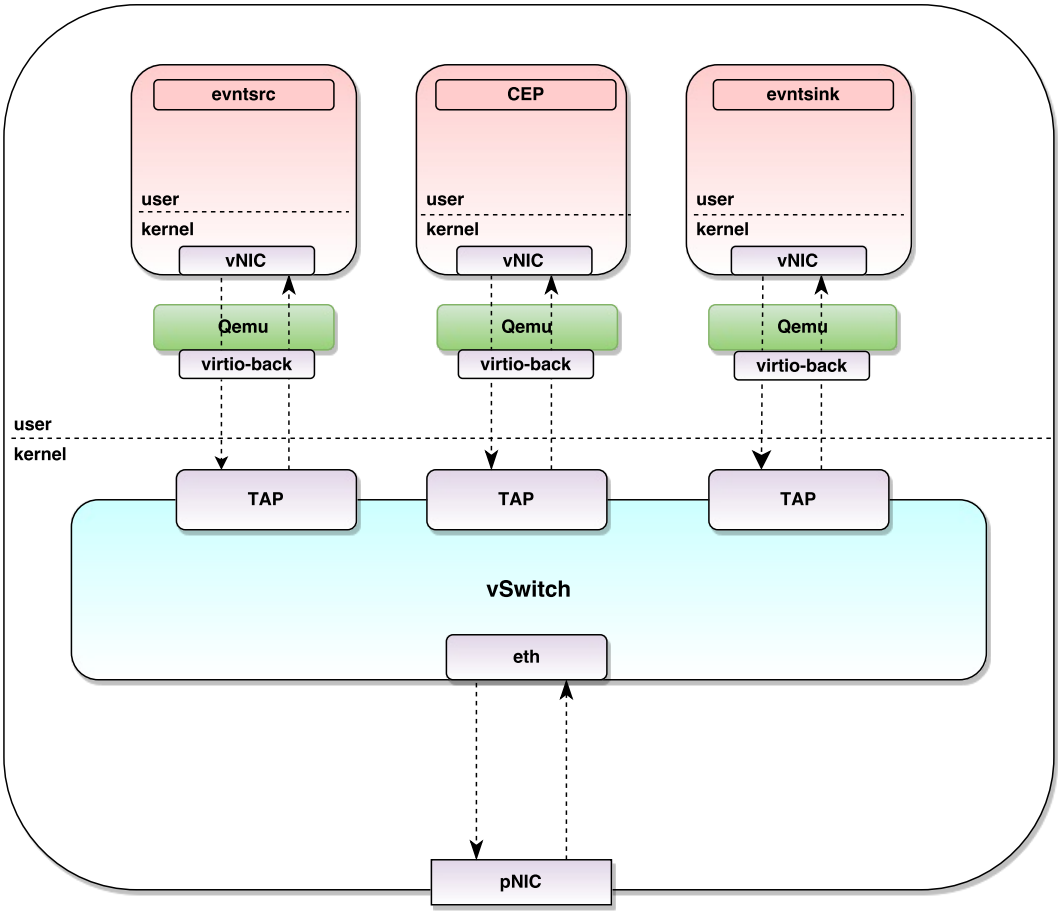
Buchmann et. al [HSB09] review various contributing technologies in the field of Event-Based systems. This thesis proposes a paradigm of viewing the underlying network as a contributing technology. Offloading application context onto the underlying network allows network architects to tune their network service to latency sensitive applications. It also provides better insights into the traffic characteristics of

the application which may be used for designing better staged processing, load balancing and reducing the overall clog on the network I/O stack. Finally, it presents network operators with opportunities to explore further monetization of the network and virtualization infrastructure and pro-actively scrutinize revenue models which blur the line between application land and the network.

1.1 Problem Statement

To understand the research question, let us consider a single node deployment of a CEP engine within a Linux Kernel Virtual Machine (KVM) [KKL+07] guest. Linux KVM uses a per-guest userspace QEMU [Bel05], a fast machine emulator to run one Operating System on another. KVM-QEMU uses the virtio [Rus08] network interface specification, which provides a series of Linux drivers for various hypervisor implementations. It specifies a standard with a virtio front-end driver within the guest kernel, a virtio-net device which is the back-end driver within the per-guest QEMU process, and a transport mechanism in the form of vring or virtqueue between the two. When the path of a packet at the host Physical NIC to the CEP application inside a Virtual Machine is examined:

Figure 1.1: Data path in virtualization environments



1. The packet at the physical device is handled by the NIC driver and lifted to the vSwitch. . The vSwitch switches the packet to the TAP device of the corresponding Virtual Machine.

2. The host injects an IRQ to notify the guest about the incoming packet. **(VM-ENTRY)** The guest schedules the QEMU userspace process within which resides the emulated virtio-net device.**(VM-EXIT)** The virtio-net device executes a *read* system call (user-kernel-user transition) to receive packet from the

TAP device and pushes into the virtqueue.

3. The virtio driver in the guest receives a callback **(VM-ENTRY)** once there is data in the virtqueue, and executes a *get_buf* to deliver the packet to the network stack of the guest. The packet is processed through the network stack of the guest and finally queued to the transport layer socket, ready to be processed by the CEP application running in user space of the guest. (guest kernel- guest user transition).

4. The raw bytes within the packet are de-serialized into an Object representation by the CEP application and normal application processing ensues.

A similar process is used - in reverse - to send a packet out. In many event processing systems, a multiple layers of processing is used where the events are directed from one engine to another, depending on the user application logic, either for consumption or for further processing. As it can seen there are multiple context switches, system calls and packet copies to deliver packets to the guest. Even after this point the packets have to go through the complex Linux networking stack [BRE+15] to be delivered to the CEP engine. This is especially problematic because of the following characteristics of Event Processing applications:

- High Data Arrival Rates
- Low Signal to Noise Ratio
- Staged Processing

Although CEP engines implement their own version of flow control to cope with the arrival rate, it results in latent reaction to events. In addition event processing may also be a staged process with different stages handled in different virtual machines, adding extra burden on the I/O stack of the host. These inefficiencies cannot be handled within the CEP engine. An effort to use the underlying network services to relieve the burden on the CEP engine is worth considering, particularly now more so because of the emergence of Network Functions Virtualization and Software Defined Networks.

1.2 Contribution

The main contribution of the thesis is in providing an application-aware virtual switch that detects user-defined event patterns and applies user-defined actions. The main idea behind this implementation is that an earlier detection of events and subsequent application of appropriate actions before the packets traverse through complex path described in section 1.1 would reduce the point-to-point latency between source and sink of the data. The implementation also aims to aid in staged event processing by utilizing the Layer 3 routing capabilities of the virtual switch and enabling it with application context.

Mekky et. al [MHM+14] provide an application-aware implementation of Open vSwitch, which is capable of content-based server selection and load balancing. But they rely on the controller heavily to adapt to packet flows, which is not ideal in a streaming scenario. Zhang wt. al [ZWRH14] implement a mem-cache aware SmartSwitch prototype to cache data blocks near worker nodes. The switch interprets application data inside the packets and redirects them to the appropriate servers. However, the implementation is done on a custom virtual switch and currently offers only redirection and load balancing capabilities. [HRW15] describe and implement a high-speed packet processing platform, NetVM, built using Intel's DPDK library. Within this implementation is an hypervisor based switch that is capable of steering traffic by learning the application context. However, NetVM comes packaged as a middlebox solution to be used in the network functions pipeline.

There are several publications in the area of high performance packet processing [[GEW+15] , [NWL13]] within cloud networks. They mostly aspire to zero-copy I/O frameworks and to reduce the number of context switches needed during packet I/O. Vhost-net is an accelerator to the virtio framework, which provides vhost-net device in the kernel space which can DMA to virtio front-end in the guest. It aims to avoid the context-switch between the VM and QEMU during packet I/O. But this framework requires a modification to the host kernel itself to support the vhost-net device, since the device is not tied to KVM. DPDK [Sch14], netmap [Riz12], and PF_RING ZC [KN17] offer shared-memory and modified drivers to by-pass the default network stack and allow the application to directly manipulate the packet buffers using custom API. However, this also has the disadvantage of monopolizing the NIC for a single application. VALE [RL12] is a virtual ethernet switch for high-speed inter-VM communication based on netmap API and batched processing. This is however highly tuned for inter-VM communication only and also is an L2 switch. While the aforesaid works try to improve packet I/O using different techniques, they do not specifically cater to a characteristic in the Event Processing domain, where majority of packets are discarded by the application after processing, or are forwarded to another machine. More will be discussed about the research areas which inspired and provided groundwork for the thesis in Chapter 3.

The main priority of the thesis is to provide a production switch for offloading aspects of Event Processing applications onto the network. The vSwitch is not designed to be a complete replacement for an Event Processing engine, but rather as a network based complement to it. To this end, the following contributions have been made:

- A context-aware Open vSwitch implementation is implemented to detect event types and steer them to virtual machines and facilitate staged processing without having to context switch into an intermediate virtual machine.
- The Open vSwitch is enabled to execute stateful logical operations on the data items of an event and filter based on application logic.
- 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.
- Evaluation of the implementation in different deployment modes and a comparison to the performance of a source-tree based Open vSwitch deployment is presented.

1.3 Outline

The thesis document has been structured as follows. In Chapter 2, the different design and deployment paradigms, concepts, and technologies that provide a context to the thesis are explored with the intention to provide a brief introduction to the paradigms, protocols, libraries, development kits and deployment modes used for the design, implementation and evaluation of the thesis. Following that, in Chapter 3, a survey into the various publications and research projects which carry out inquiries in the same spirit of the thesis is laid out with an attempt to contrast and highlight the difference in contribution. In Chapter 4, the design and implementation of the vSwitch and Ryu based controller is detailed. In Chapter 5, a thorough evaluation of the implemented solution with a relevant discussion in presented. Chapter 6 includes a discussion on the conclusions and future work.

2 Background

2.1 Software Defined Networking

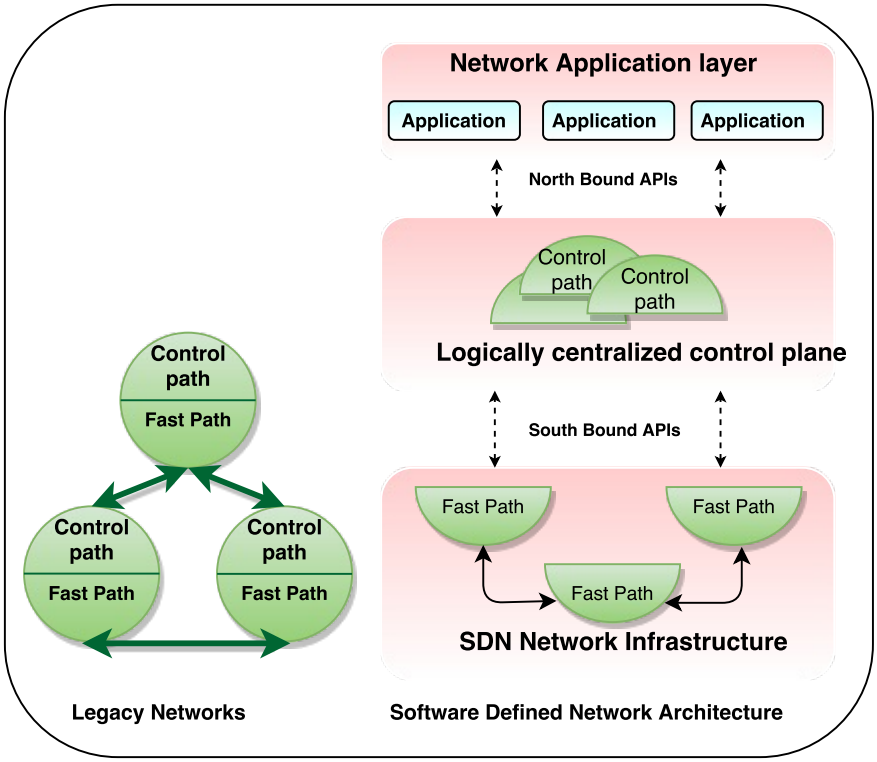
Software defined networking is defined as a dynamic network programming paradigm that uses open interfaces to enable programmability of network devices. The primary focus is on aggregating the so called control plane functionalities of all the network devices such as system configuration, management, and exchange of routing table information at a logically centralised location. The approach implies that the control path is decoupled from the fast path of the network device. While both the planes are set to evolve independently in a vendor agnostic manner, the fast path still utilises the policy information made available by the control path to treat the packets both in the incoming and outgoing directions.

The figure depicts a typical software defined network and provides the comparison with a legacy network. The devices in the traditional computer networks are comprised of both control plane and fast plane functionalities embedded into them directly as shown on the left-hand side of the diagram. Such devices lack the global network state information and require manual configuration during deployment. It also demands for a certain mechanism to frequently propagate the routing information due to the distributed nature of the control plane.

The software defined network architecture on the other hand is built on top of the commercial off-the-shelf hardware components. The architecture is decomposed into three distinct layers of functionalities as shown on the right-hand side of the diagram.

- Network infrastructure layer: It is the bottom most layer of the SDN ecosystem and contains the network devices themselves (both virtualized as well as physical devices) that are capable of switching/routing packets at line rate. They expose programmable interfaces to be leveraged by the upper layers of the SDN architecture.
- Controller layer: It is essentially the logically centralised control plane of the whole network infrastructure underneath. It forms the middle-ware of the SDN ecosystem and provides the necessary framework that binds the applications that require network services and the protocols that communicate with the network infrastructure. The vendor agnostic nature of the SDN architecture necessitates a set of communication mechanisms towards the network infrastructure which is commonly termed as southbound APIs. The OpenFlow, NETCONF and OVSDB protocols are examples of such mechanisms provided by the SDN controller.
- Application layer: The layer is composed of various network applications and the Northbound APIs exposed by the controller layer. Although there is no normative standard to describe what Northbound APIs are comprised of, Restful APIs are prevalent in present day SDN applications. Some SDN controllers such as the OpenDayLight also provides OSGi based interfaces for native network application development.

Figure 2.1: SDN Architecture



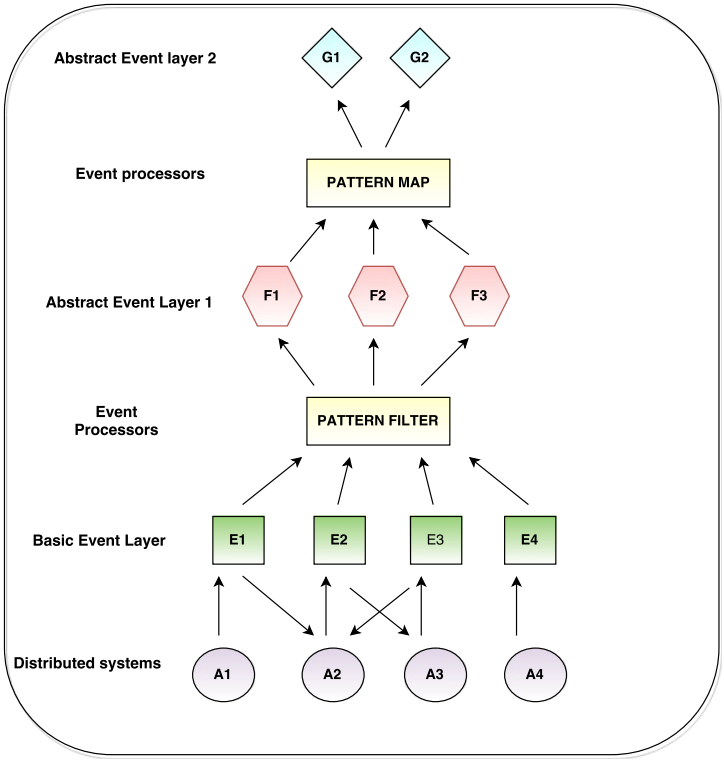
2.2 Complex Event Processing

Complex Event Processing (CEP) "is a defined set of tools and techniques for analyzing and controlling the complex series of interrelated events that drive modern distributed information systems." [SOP+15]. The Event Processing paradigm sees the world as a series of discrete events with the (producers) sources and sinks (consumers) of the events decoupled in space and time. In modern day distributed systems, events are delivered from sources to the sinks via an Infrastructure of distributed processors, which coordinate to provide processing and routing services. In such an Event based system, the sources and sinks may span a wide geographical area, connected by a CEP middleware, as described in [CM13]. The CEP middleware detects primitive events based on described patterns, interprets and combines them to identify higher level composite events. A CEP middleware, thus can be said to be comprised of three key components:

- Rules Engine, responsible for expressing rules used by the compute node.
- Compute Node, responsible for detecting primitive event patterns and interpreting them to generate complex events based on the rules.
- Notification Service, responsible for notifying the event sinks about the occurrence of events.

A CEP middleware is characterized by two important aspects: One, It decouples the source of the events from the sinks both in space and time; Two, it is capable of filtering events based on patterns and generating new events based on relationships among events using techniques of aggregation and composition.

Figure 2.2: Event Abstraction Hierarchy



2.2.1 Hierarchical event abstraction

Luckham et. al [LF98] present a hierarchical view of a complex event processing system based on a hierarchy of abstraction in events. As events move up the layer, they are subjected to different operations which transform the stream of events. At each layer of abstraction, the events may be delivered to interested sinks or sent to other compute nodes for further processing. A complex event processing system may thus be seen as series of operators applied on events depending on the desired event abstraction required by the consumers of events. Operator placement as such is a crucial research area in such systems. In section 3.1 a brief discussion is presented on the operator placement strategies within distributed complex event processing systems.

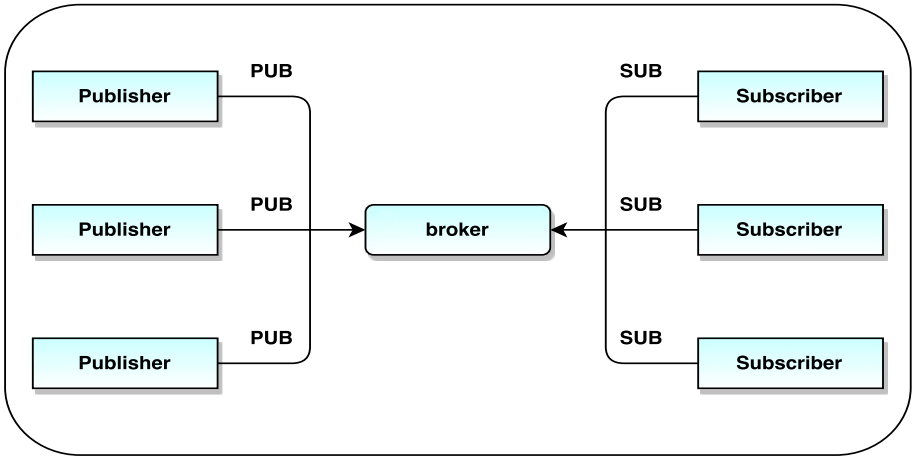
2.2.2 Communication models in complex event processing

One of the most important traits of a complex event processing system is the decoupled nature of communication between the producer and consumer of events.

- Pub-Sub: In the publish-subscribe model of communication the senders of the message , otherwise called publishers, do not send the message to the receivers of the message, otherwise called subscribers. Rather the messages are published to a broker without the knowledge of the subscribers. The publishers advertise the topics to the broker and subscribers subscribe to a topic. A topic can have many publishers and many subscribers. On receiving a message on a topic, the brokers are responsible for delivering the message to the subscribers. Pub/Sub can be classified as a wide message delivery model of communication. Complex event processing systems are deployed using the pub-sub communication model for decoupled event delivery, where the consumers and producers of events are moving; with only the broker as a non-moving entity with compute nodes within

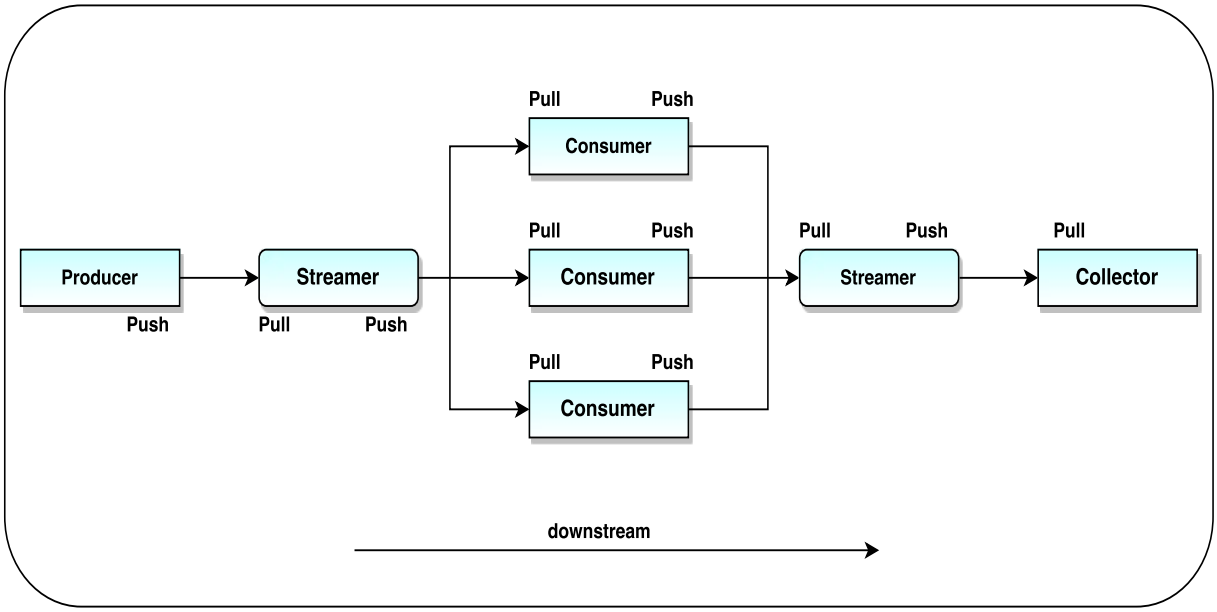
complex event processing systems behaving both as producers and consumers of events. There are several Pub-Sub based notification services in the market today. Google Cloud Pub/Sub [KG15] is one example of a thriving publish-subscribe message oriented middleware.

Figure 2.3: Publish-Subscribe communication model



- Push-Pull: In a push-pull model of communication the messages are distributed to multiple workers which are registered as its pull clients. The pull clients which can do processing on their own do not rely on any broker but instead have their own pull clients to which the events are redirected. The extent of processing or filtering of events is directly dependent on the implementation. However the events in this model of communication are sent evenly to multiple workers which may send the processed events to a single collector or different set of collectors create different stages for processing. The push-pull model is a pipe-lining pattern where all the subscribers need not receive the same message. Apache Kafka [Kaf14] is a high throughput distributed messaging system which uses the push-pull communication model to build robust data pipelines.

Figure 2.4: Push-Pull communication model



2.2.3 Event Query languages

Another crucial component of any complex event processing system is an event query language. Event query languages allow for the expression of user logic in a concise manner that is understood by the processing engine. Event query languages direct the processing engine to make sense of an event by describing the patterns for an event. Modern event query languages are equipped with capabilities to express temporal relationships among events with instructions for explicit and precise operations upon detection of events which share non-obvious relationships. Sophisticated features provided by modern event query languages include moving time windows, detection of causal relationships between events, aggregation of event data, generation of new events etc. For example, Esper [sRe15] offers a domain specific language for dealing with high-frequency time-based events with SQL-like queries with support for sliding windows and event-series analysis.

2.3 OpenFlow

The OpenFlow protocol is the default southbound API of choice in the SDN-enabled networks. Initially, it was considered as a communication mechanism that facilitates experimental protocols to be run in the computer networks [MAB+08]. It has now evolved into a rich set of APIs which can be leveraged to modify the forwarding behaviour of network devices. The OpenFlow capable devices are manufactured from multiple vendors and include routers, switches, virtual switches, wireless access points[OFA] and so on.

To delve deep into the OpenFlow protocol features, the specification adheres to the SDN principle by separating control functions from the devices and thereby centrally managing the global forwarding policies in the network. The protocol has a set of messages defined for communication between the SDN controller and the underlying network. The OpenFlow capable switches operate based on the match-action policies programmed into their forwarding database. The match-action policies are defined on one or more flow-tables and a group table built into OpenFlow-capable switches. The flow-tables are essentially a set of rules containing packet headers/characteristics against which the packets in a flow are matched in order to make a packet modification/forwarding decision. The set of linked flow tables that provide matching, forwarding, and packet modification in an OpenFlow switch is termed as a pipeline [Fou12] and hence the packets are said to undergo pipeline processing. The pipeline processing of a packet may require that a packet traverses through one or all of the flow-tables defined in the incoming or outgoing direction. It should be noted that the packets are subjected to processing by different set of flow-tables defined distinctly for ingress and egress directions. The SDN controller utilizes OpenFlow messages to add/modify/remove rules into and from the flow-tables. The rules in a flow-table are assigned priorities to resolve match conflicts and highest priority rule entry always take effect and the action(s) associated against the rule selected is executed. Besides actions, OpenFlow also allows several packet/byte level counters to be tagged along with a flow-rule. Such counters are useful in traffic monitoring, shaping and policing activities. The protocol, for the aforementioned purpose, also allows the SDN controller to describe a meter table with per flow meter entries (associated with flow rule entries) to achieve traffic shaping and policing.

Further, to provide an overview of the OpenFlow protocol communication mechanism, it supports three different type of messages:

- Controller-to-switch: These messages are used to manage and obtain the status of the switches. For example, the controller can request a switch for its basic capabilities using the Features message.

- Asynchronous messages: These kind of messages are triggered by the switches to inform the controller about change in operational status or other network events. For example, PACKET_IN message is triggered to indicate a packer arrival event.

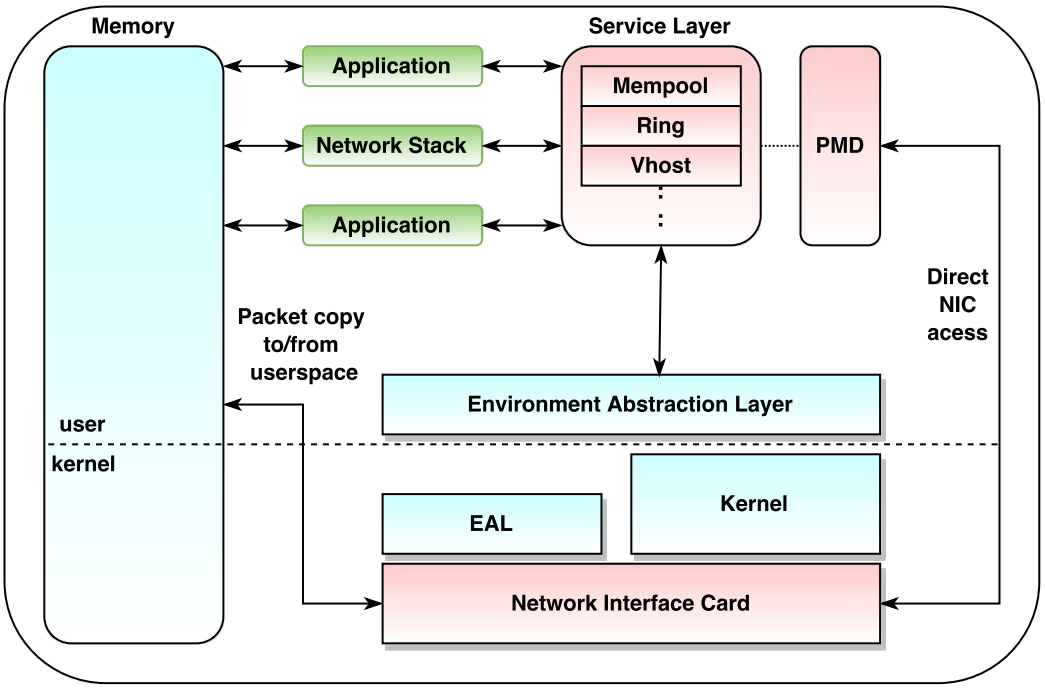
- Symmetric messages: These messages are typically heartbeat messages, error messages or initial Hello messages initiated by either of the parties i.e. the SDN controller or the network device. Some experimenter messages may also fall under this category.

A detailed description of all the messages and their implied functionalities can be obtained from the OpenFlow specification document[Fou12]. The OpenFlow protocol provides reliable message delivery through its channel connection built on top of TLS or pure TCP. The standard does not automatically impose any acknowledgements for the messages sent nor does it ensure ordered processing of messages and is left open for the switch implementations to handle the same.

2.4 Intel Data Plane Development Kit

Intel Data Plane Development Kit (DPDK) [DPD] is set of data plane libraries and userspace drivers that enable fast packet processing. Intel DPDK runs in Linux userspace and thus allows application programs to bypass the kernel for packet processing. As a consequence users of the DPDK libraries have the flexibility to run third-party fast path network stacks and develop fast packet capture algorithms all in the userspace. By enabling kernel bypass, DPDK reduces the number of cycles taken to send and receive packets. The main components of DPDK, as illustrated in figure are:

Figure 2.5: DPDK Components



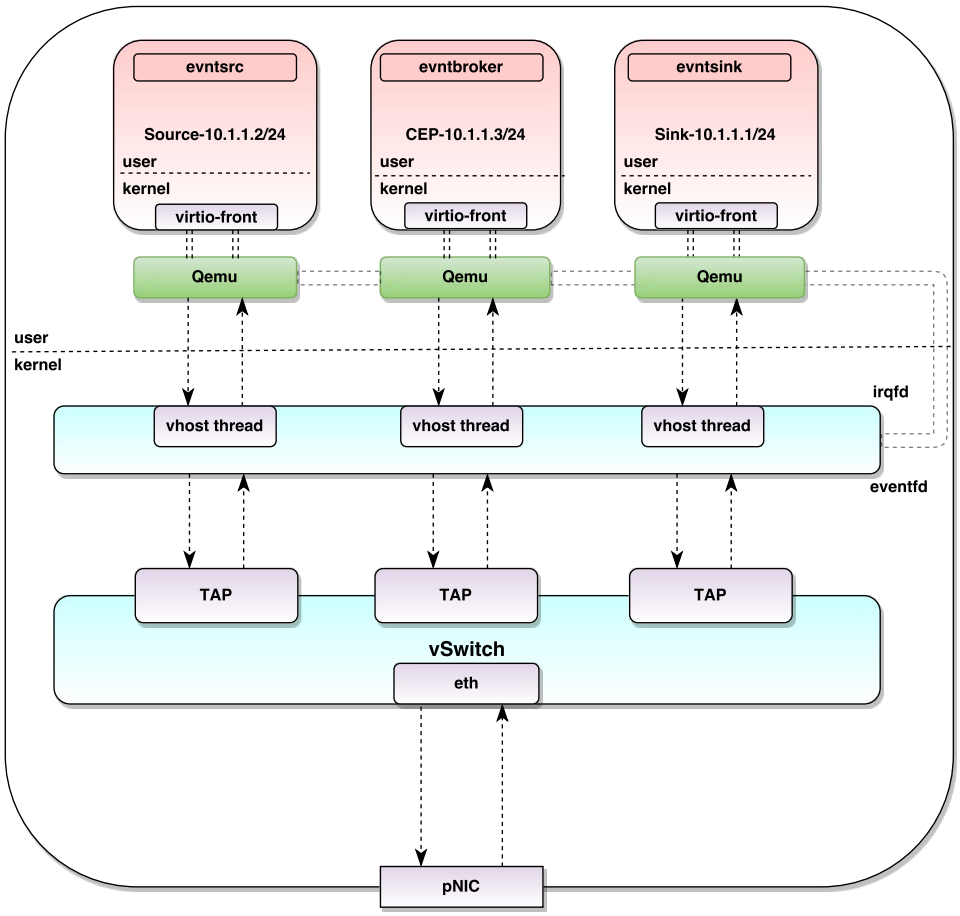
- Environment Abstraction Layer: The EAL provides an interface for the DPDK applications and libraries to access resources such as memory and hardware. The libraries and the applications are not aware of the allocation of these resources. The EAL is responsible for physical memory allocation, multi-process support, Interrupt-handling, PCI access, CPU feature recognition and core affinity setting among others.

- Service Layer: The service layer of DPDK consists of various libraries to enable DPDK applications to make use of the features provided by the EAL. The service layer can be seen as the provider of ’system calls’ to the DPDK applications. For example the ring library provides a lockless FIFO ring implementation for Rx and Tx queues with support for burst queue and dequeue; The Mempool library implements a pool-based aligned memory allocator with features such as per-core cache and hugepages; The vhost library provides a userspace virtio driver that allows the users to manipulate the vring. Full list of DPDK libraries can be found at [DPD]
- Poll Mode Drivers: The DPDK Poll Mode Drivers (PMD) provides userspace device drivers and APIS to set up queues and devices. The PMD is capable of accessing the transmit and receive queues directly without interrupts by continual polling. This ensures that the packets at the receive queues are directly received by the userspace application when receiving and the packets from the user space are directly at transmit queues when sending.

2.4.1 DPDK Vhost library

The DPDK vhost library requires a special mention because of its support for vSwitch implementations. The vhost library provides a userspace vhost driver to manage vhost threads for corresponding virtio front-end devices in guest. Before understanding the DPDK vhost library, it would be useful to understand the vhost-net acceleration standard on virtio. As illustrated in figure 1.1 in section 1.1, the Qemu userspace process normally provides the virtio back-end implementation for the virtio-net front-end device in guest. This results in a situation where Qemu is involved intensely in packet I/O thereby increasing the context switch required. As an improvement to this virtio standard of having a front-end in guest and back-end in Qemu, the vhost-net standard came in to being. In the vhost-net implementation, a vhost-net device is introduced within the kernel space which acts as the back-end to the virtio front-end device in the guest. Keeping the vhost-net device in kernel ensures that Qemu process in not needed for every packet transfer between the host and guest. Instead Qemu is involved in setting up the event file descriptors and the Interrupt file descriptors. Vhost creates a thread per virtio device which waits on the eventfd provided by Qemu. The host machine uses the eventfd to communicate with the guest by binding the eventfd to an interrupt source or listening to the eventfd which is triggered by the KVM when the guest writes the NOTIFY register.

Figure 2.6: Vhost-net acceleration



As illustrated in figure 2.5, vhost-net acceleration relies on Qemu for device feature negotiation, migration etc. The vhost driver at the kernel is not a full fledged virtio device implementation, because Qemu still controls some parts of the set up. However a virtqueue exists between the virtio-front end in the guest and Qemu. The vhost thread in the kernel can DMA to and from the virt queue giving it access to the virtio-front end without Qemu intervention. This implementation can be contrasted with the DPDK vhost implementation which is illustrated in figure 5.13. In the DPDK vhost implementation, the vhost driver is moved completely to userspace. A dpdkvhostuser port attached to the vSwitch registers callbacks to the vhost library in userspace, which is called when the vhost device is activated/deactivated in the guest. For each virtio device in the guest, a vhost device is created by the vhost driver(vhost library). The dpdkvhostuser port (vhost-device) and the virtio device share a virtqueue that enables them to directly exchange packets without intervention from userspace Qemu. So a packet at the physical NIC is lifted into the userspace vSwitch where it is switched to the dpdkvhostuser port attached to the guest VM. Since the dpdkvhostuser port shares a virtqueue with the virtio device in the guest, the guest receives the packet with minimum overhead.

3 Related Work

There has been a great deal of research in the areas of Complex Event Processing, Software Defined Networks, and High Performance packet processing. Since this thesis derives motivation from all the three, related work is surveyed and organized according to the domain. In this chapter, the outcomes of the research works and takeaways from the perspective of the thesis are briefly discussed to give thee rational or motivation behind the implementation of the thesis.

3.1 Deployment Strategies and Operator Placement in Event Processing

Since the thesis focuses upon creating a data plane solution specifically tuned for the Event Processing/Event-Based systems, it is important that key factors impacting the design of such systems are studied in detail. Cugola et. al [CM13] discuss different deployment strategies for distributed complex event processing systems. The strategies described in the paper can be classified based on:

- **Organizing processors into processing trees**: Primitive events from sources are collected, filtered, and processed as the move upwards in the tree. This allows for incremental evaluation at intermediate processors in a manner similar to staged processing.
- **Forwarding advertisements to processors**: Each processor in the processing tree maintains an advertisement and subscription table. When receiving an event notification, each processor computes the set of subscription that match the event and determine which node in the processing tree receives it.
- **Rule deployment**: The rules are recursively partitioned into partial rules and deployed in the root to source of the processing tree.
- **Notification Forwarding**: Two concepts of event forwarding are discussed - Push based and Pull based. In Push based forwarding, a processor forwards all matched primitive events to the parent; whereas in Pull based forwarding the processor stores the matched event until it is explicitly asked by the parent.

In addition to deployment strategies, Cugola et. al present an analysis of content-based filtering of primitive events. Their analysis highlights the advantage of content-based filtering close to source to reduce the network traffic in both distributed and centralized deployment. Their work informs and motivates the thesis at many levels. With a content-aware switch and a SDN based controller to deploy rules, a logical processing tree based on event types can be constructed to process different event types at different processors. Since the controller has a logical view of the network, rules can be deployed in such a way that one processing node is responsible for a pre-decided event types. The events from one processing node can be then chained to another based on the logical processing tree using a push based forwarding strategy.

Pietzuch et. al [PLS+06] discuss the NP-hardness of the operator placement problem and propose a heuristics based approach. They propose a stream based overlay network in between the physical network and the stream processing system and delineate three challenges in the operator placement problem: a) Achieving good application query performance b) Use network efficiently c) Reuse existing operators when appropriate. Zhou et. al [ZOTW06] present a cost model for dynamic load balancing in distributed stream processing systems and identify the importance of load balancing in streaming applications. While publications such as [BFÇ12] and [CV14] focus on reducing the end to end latency, [ABQ13] and [CYY16]

focus on reducing the inter-node traffic, and [RDR10], [PLS+06] focus on reducing the overall network usage. While all the concerns are valid for the distributed event processing environment, the research focuses using the application layer for optimization. Even when an In-Network solution is discussed, the solution is based on deploying processing nodes as close to the data source as possible. The discussed research works provide good motivation for selecting optimization parameters in Event Processing systems, however the solution provided in the thesis is a pure data-plane solution, focusing utilizing the underlying network service instead of the application level processing node.

3.2 Fog Computing and Software Defined Internet of Things

There has been an exponential growth of connected devices, and it is unanimously accepted that the number of connected devices is only going to increase from the current 10 billion nodes to 50 billion nodes over the next few years. This presents us with two interesting challenges:

- **Handling massive streams of data from the sensory nodes**: Large cloud deployments of today are built to handle massive amounts of data from web and other batch processing applications. But when there are billions of sensory nodes located in widespread geographic location producing streams of sensor data, transporting all the data to the cloud is counter intuitive. For one, it creates unnecessary traffic in the network because most of the sensing data is discarded anyway and for another it creates a lot of latency. And as properties of sensory data goes, it is mostly useless; but when it is not, it is most likely very important. This problem has been discussed even before the explosion of IoT networks. The operator placement problem discussed in the previous section to some extent attempts to address this problem by moving the operators closer to the sources of data in order to react to events faster and reduce network traffic. Fog computing is a local cloud deployment paradigm which does exactly this. Bonomi et. al [BMZA12] discuss in detail the characteristics of IoT and advantages of Fog deployment in IoT.
- **Maintaining the massive network for the sensory nodes**: As billions of wired and wireless sensory nodes join the network, traditional networking is insufficient to utilize the paradigm fully. Software Defined Networking which we discussed in Chapter 2.2, introduces significant flexibility for resource management by separating the data plane from a software based control plane. Software Defined Internet of Things promises an architecture to seamless manage, configure, and spawn billions of sensory nodes to create Sensing-as-a-Service(SaaS) paradigm. Research done in [EMIH15] , [QDG+14] and [FGHN15] detail such an architecture and provide prototype deployments with network models.

Truong et. al [TLGD15] deploy a Software defined Fog computing Adhoc vehicular network, whereas Xu et. al [XMR16] prototype a Software Defined Fog computing Message Queueing Telemetry Transport (MQTT) broker for IoT devices. Prototypes built by [XMR16] and [FGHN15] deploy Open vSwitch as the Openflow switch in their respective SDN based Fog deployments. This highlights the versatility of Open vSwitch in both large scale global cloud and small scale local Fog deployments. These works motivate the selection of Open vSwitch for an application aware deployment in cloud, be it in large data centres or in IoT networks.

3.3 Application-aware Data Plane Processing in SDN

Mekky et. al [MHM+14] prototype an Application-aware implementation of Open vSwitch(OVS). The prototype implementation, as shown in the figure 3.1 - sourced and reinterpreted from [MHM+14] - creates a separate application module within the Open v Switch with its own separate app-table. The app-table is

staged after the Openflow flow table pipeline. The application module is responsible for handling the flow out of the app-table and executing app-actions. An app-action is capable of doing the following operations:

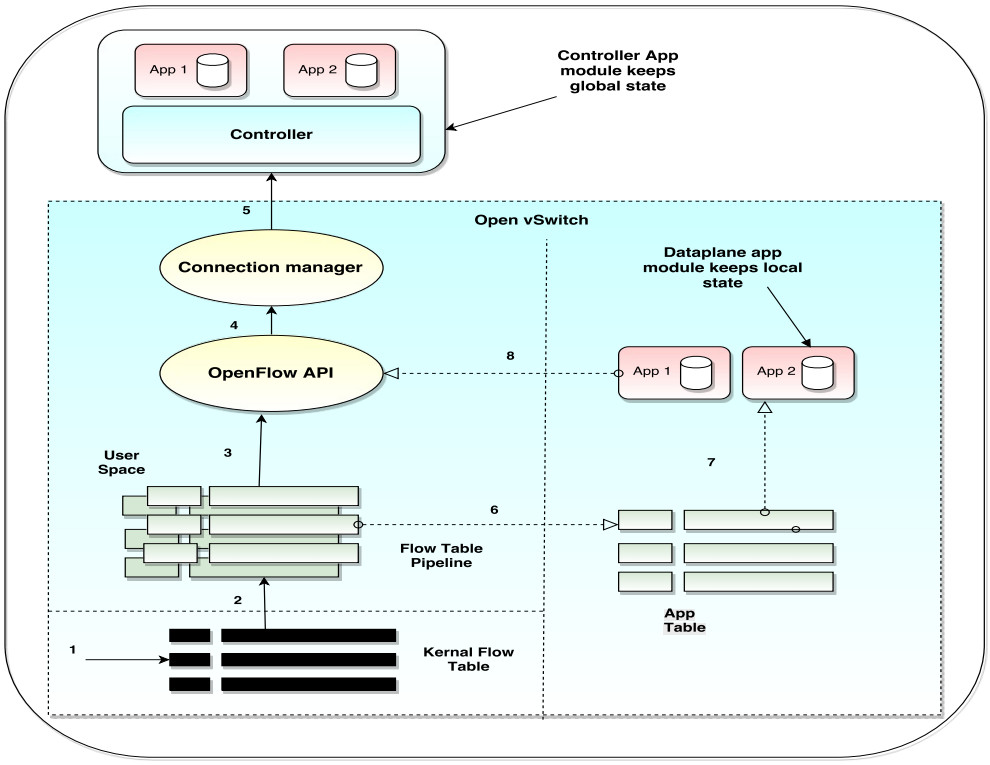
- Determine actions for a packet and modify the state if required.
- Generate or modify the rule set for processing this flow.
- Remove flows from the Openflow tables and generate packet out to other switches.
- Send custom vendor messages to controller.

A table-miss in the standard Openflow table is sent to the app-table instead of the controller as in the standard Open vSwitch. The app-table is configurable with custom app-actions. These app-actions are executed within the application module in the userspace. In summary, the application module within the OVS runtime, behaves like a light-weight controller capable of receiving select flow, executing them in userspace, and taking decisions by calling the Openflow APIs. The research work also provides valuable information about the interception of packets within the OVS as a design choice. Mekky et. al lay out three options to the said end:

- Intercept the messages between controller and OVS at the connection manager.
- Intercept the packet after looking up the Openflow flow table.
- Add app-actions directly to flow table and apply actions directly when the rule is matched.

Although L4-L7 match is supported in the application module, the implementation does not scale to Event Processing paradigms where event patterns can occur in any packet of the stream. In such a scenario, the prototype implementation lifts each packet to the application module and out of the Openflow processing pipeline inducing processing latencies. This work provides motivation to keep processing in-line for scenarios such as stream processing where rules have to be applied for each packet in the stream.

Figure 3.1: Application Aware Data Plane Processing



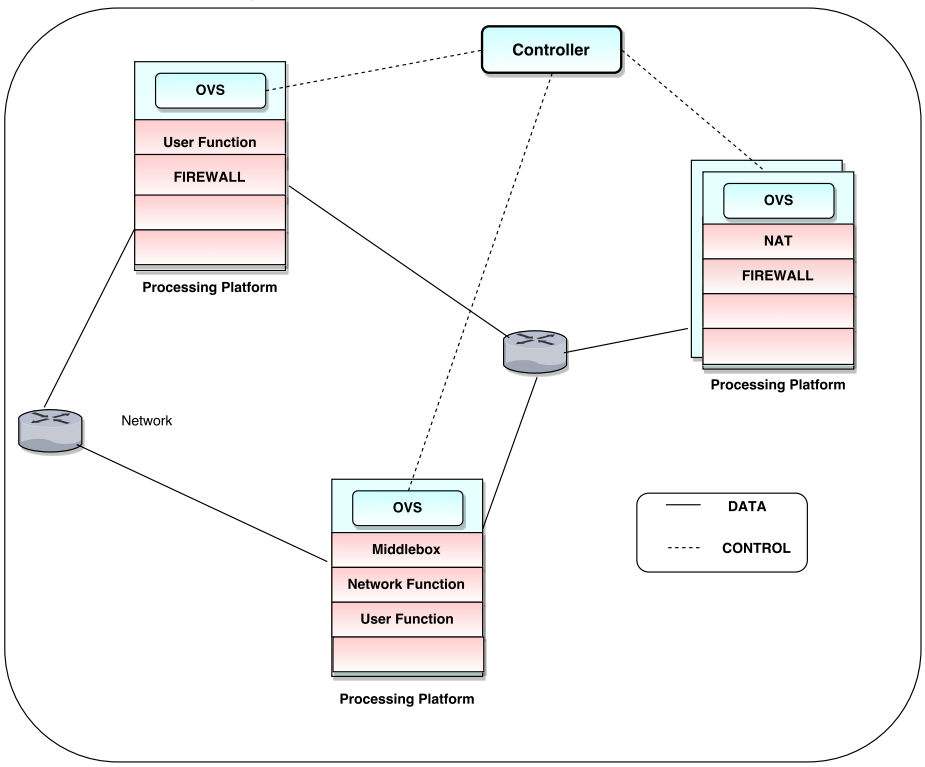
3.4 In-Net: In-Networking Processing for the Masses

Stoenescu et. al [SOP+15] prototype a ClickOS based [MAR+14] processing platform running on commodity servers. Their results promise the possibilities of running single, in-expensive commodity servers scattered around in the network with the vision of providing a controller-based access to both network-providers and end users. The processing platform, which relies on ClickOS is capable of hosting upwards of thousand clients per box. In this model, end-users can demand for adhoc network service, in the box which can be provisioned instantaneously. While they also provide various security configurations and static analysis as a way to measure the validity of the provided service, the deployment model used provides a very interesting insight into the motivation for work done in the thesis. In the work done by Martins et. al on ClickOS, they consider two options to for the ClickOS switch

- **VALE based deployment**: The first model uses a modified VALE [RL12], a Virtual Local Ethernet switch to connect the virtual machines. VALE is a software switch which uses netmap (Discussed in Section 2) API and batched packet processing for high performance switching between guest virtual machines machines.
- **Open vSwitch**: The second model uses a standard Open vSwtich for connecting the guest virtual machines.

The results produced express that the performance of the modified VALE switch is better than that of the standard Open vSwitch; explained partly by VALE's usage of the netmap API - which allows the packet buffers to be mapped to the VM's memory space - and other modified APIs enabling batched packet processing.

Figure 3.2: In-Network processing model



However the implementation by Stoenescu et. al [SOP+15] uses the Open vSwitch within their processing platform, highlighting the versatility of Open vSwitch. The ability of the Open vSwitch to speak

Openflow and thereby be configured by controllers in the Software Defined Networking model makes it a valuable option for deployment in In-Networking processing scenarios. One such deployment scenario is detailed in the figure 3.2 - sourced and reinterpreted from [SOP+15]. As we can see, the processing platforms are dispersed all around the network. Each processing platform consists of a Open vSwtich which is capable of speaking Openflow. The users and network operators rely on the Openflow controller to request, provision or service network functions. Although the performance of Open vSwitch does not compare well to that of a hardware unit, its versatility makes it a great design choice. Moreover with the advent of DPDK, the performance of Open vSwith can be vastly improved, all the more consolidating Open vSwitch as a design choice for the implementation in the thesis.

3.5 SmartSwitch: Bluring the Line Between Network Infrastructure and Cloud Applications

Wei Zhang et. al [ZWRH14] prototype an application-aware networking platform that performs functions that are characteristically performed by compute platforms. They explore how the boundaries between applications and network infrastructure can be obscured by modern day packet-processing technologies. They prototype a memcache-aware switch, that achieves application level redirection to memcached servers and local caching to allow immediate responses to hot requests. The paper examines three areas as use cases for their effort:

- **Load balancing**: Although SDN allows for a much more flexible policy shaping of the data plane, it still does not allow decision making based on data plane information. The packets still have to be send to load balancers and proxies for content based routing. An application aware implementation may try to overcome this problem by enabling the software switch to perform redirection based on the application data. .
- **Storing Data within the network**: Focusing on content centric networks where forwarding requests and responses are based on names rather than location, an application aware switch may be equipped with a Hadoop-aware implementation and populated with cache keys and respective names for the keys. Doing so allows allows the switch to direct requests to nearest cache locations.
- **Computation in the network**: When application data has to traverse through multiple Wide Area Networks, Wei Zhang et. al propose that it is more intelligent to process or store only the information that is relevant for the application. To this end they discuss the ability to deploy computation within the network to process and filter the data instead of blind routing/forwarding at the network. They contrast this with providing hardware based switching which incurs high cost and is not flexible to changing demands of the application.

The Smartswitch prototype is implemented along the above lines and is evaluated against TwitterâĂŹs TwemProxy server. Initial evaluation results show that the Smartswitch significantly reduces the latency for cached data. The work done in this paper provides motivation to continue exploring the possibilities to offload application context into the network and possibly tune the network services for specific applications. With the advent of Network Functions Virtualization and the resulting adhoc provisioning of network services in the network pipeline, this paper provides invaluable direction for the thesis.

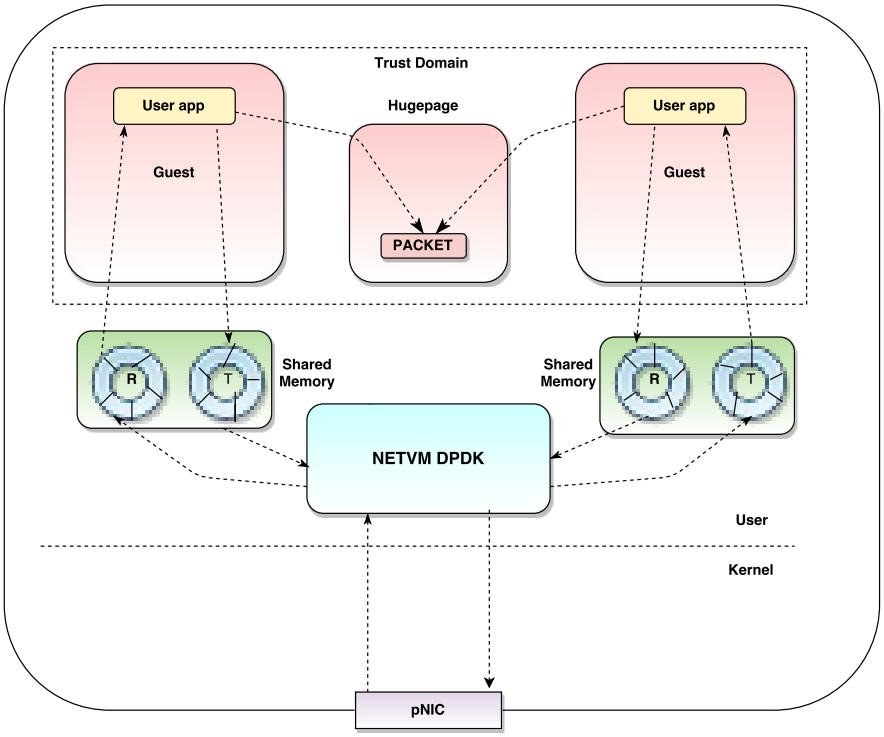
3.6 NetVM: High Performance and Flexible Network Virtualization

Hwang et. al [HRW15] present a high speed network virtualization platform which promises line speed performance for high bandwidth network functions. They innovate on top of the Linux KVM and Intel DPDK platforms to provide:

- A virtualization platform for network service provisioning which can perform at the level of custom hardware packet processing.

- A zero-copy delivery mechanism between VM's via their shared-memory framework.
- A hypervisor switch for intelligent load balancing via state-dependent or data-dependent approach.
- An architecture which enables the compositing of complex network services on multiple VMs.
- Security domains that differentiate between trusted and untrusted VM's for packet processing.

Figure 3.3: NetVM



Zero-Copy Packet Delivery

NetVM implements a Hypervisor switch, accelerated by the DPDK library. The implementation leverages DPDK huge pages support (discussed in detail in Chapter 2) and adds its own zero-copy framework. As shown in figure 3.3 - sourced and reinterpreted from[HRW15] - NetVM achieves zero-copy packet delivery by using two strategies:

- **Shared Descriptor Rings**: Shared memory regions are created between the hypervisor and individual guests. This region is used by the guests and the hypervisor switch to descriptors of an incoming or outgoing packet. The packet descriptor contains the location of the packet buffer and also the corresponding action for the packet. As a consequence a packet need not be copied between the hypervisor and the guest; only the packet descriptor is inserted into the corresponding Receive/Transmit descriptor ring within the shared memory location.
- **Shared Hugepages**: Huge pages are shared within a group of trusted guests. The DPDK enabled NetVM core polls the Physical NIC and DMA's the packet directly into the huge page region. Depending on the destination VM for the packet, the NetVM inserts a packet descriptor in the above mentioned shared descriptor ring of the corresponding VM. The VM on receiving the descriptor, extracts the packet buffer location in the huge page area, and directly accesses the packet. Similarly when a packet has to be transmitted to another guest VM, the sending VM first places the packet descriptor in the descriptor ring. NetVM core extracts packet action from the descriptor and places

the packet descriptor in the descriptor ring of the destination VM. The destination VM can now extract the packet buffer location and thereby access the packet without a single copy.

Apart from a zero-copy framework, Hwang et. al present other decisions involved during the design of their virtualizaiton platform:

Lockless Design

Conventionally shared memory locations are managed using locks, which serialize data accesses and increase the overhead involved in communication. A design which uses locks is a huge bottle neck in high speed networking because of the context switch required to gain and release the locks. Therefore NetVM uses a lockless approach by implementing parallel queues and assigning dedicated cores to service them. At any point there is only one Producer thread running on a queue, which is run on the hypervisor core, and there is one consumer thread that performs packet processing that is run on the guest VM. Since there is only one producer thread and one consumer thread, there is no need for synchronization. Additionally, even in the case of the shared huge pages, only one guest is ever in control of a packets descriptor. As an extension, only one application accesses the packet at a given moment ruling out the need for expensive synchronization primitives.

Numa-Aware Design

In Multi-processor systems cores on different socket accessing the same cache line results in expensive cache invalidation messages. NetVM sidesteps this issue by allocating and using huge pages in a way that each socket has its share of huge page and the guest threads which access this region of huge pages are pinned to a particular socket. NetVM achieves NUMA-awareness by creating as many Receive/Transmit threads in the hypervisor as there are sockets and each thread is used to process the packets local to that socket. So when the packet is accessed by the host or the guest, it always remains in the local memory. This design creates a pipeline for packet processing such that even when packets traverse from one guest to another, the threads used to process them remain are selected such that there are no cache overheads.

Huge Page Virtual Address Mapping

Huge pages represent a huge contiguous memory area to the guests. But because of the NUMA aware design, these huge pages are non contiguously allocated by the hypervisor. So normally the address of a packet at the hypervisor does not make sense to the guests. And looking up these addresses adds an overhead while performing line-rate packet processing. This challenge is overcome by NetVM by exposing huge pages to the guests using a emulated PCI devices. The guests poll the emulated device and maps its memory to userspace. This makes the huge pages appear contiguous to the guest. After this point, NetVM uses a precomputed lookup table and bit operations to convert packet address to a huge page index and offset.

Trusted and Untrusted Domains

NetVM creates security domains within a processing host. A virtual machine is assigned to a trust group which decides its access to a range of memory. Virtual machines not given access to a trust group cannot access the huge page areas allocated to another trust group. Hwang et. al also discuss the possibilities to subdivide security groups such that the DPDK classification engine is used to decide which huge page pool a packet must be DMA'd into, thereby creating a mechanism to slice flows depending on trust groups.

4 Design and Implementation

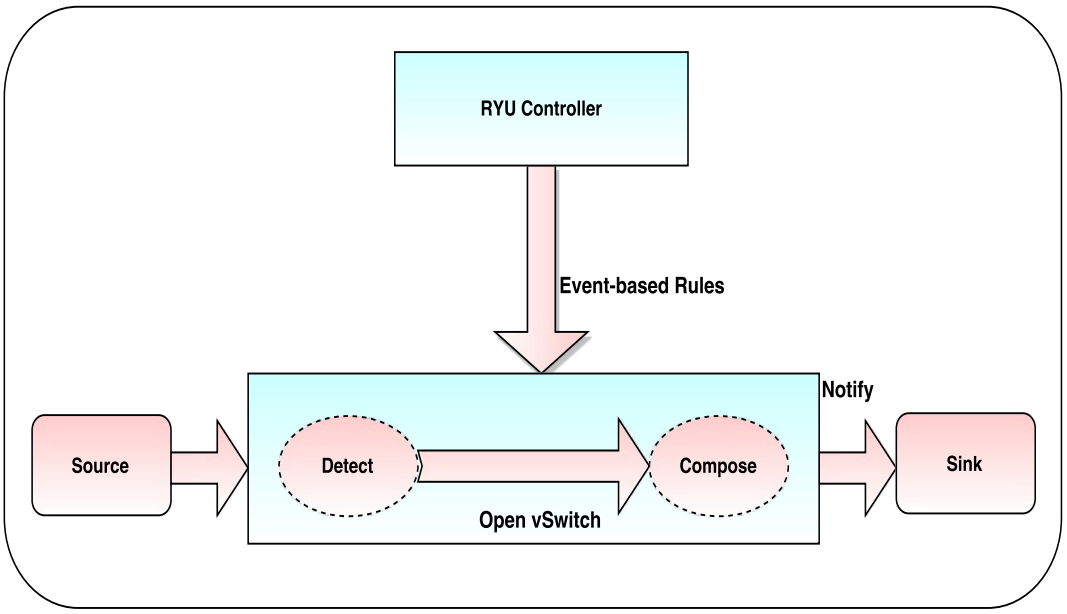
In this chapter, the design and implementation of a programmable event processing framework within the Open vSwitch is presented. As a preview to the design the problem statement of the thesis is revisited, and the goals of the implementation are established in section 4.2. The rationale for an Open 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?

4.1 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 [Sup14] reported that nearly 40% of production deployments used Open vSwitch, and by 2017 this number had increased to 60% [Sup17]. 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.

Figure 4.1: Open vSwitch viewed as an Event Processing engine



4.2 Goals of the implementation

Goals of the implementation are defined as below:

- 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.
- The Open vSwitch implementation focuses on the DPDK datapath to take advantage of the accelerated packet I/O offered by the DPDK libraries.
- The Open vSwitch is enabled to execute simple logical operations on the data items of an event and filter based on application logic.
- 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.

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.

4.3 An overview of Open vSwitch

In this section a brief overview of the three most important components in the Open vSwitch [PPK+15]

- **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.
- **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.

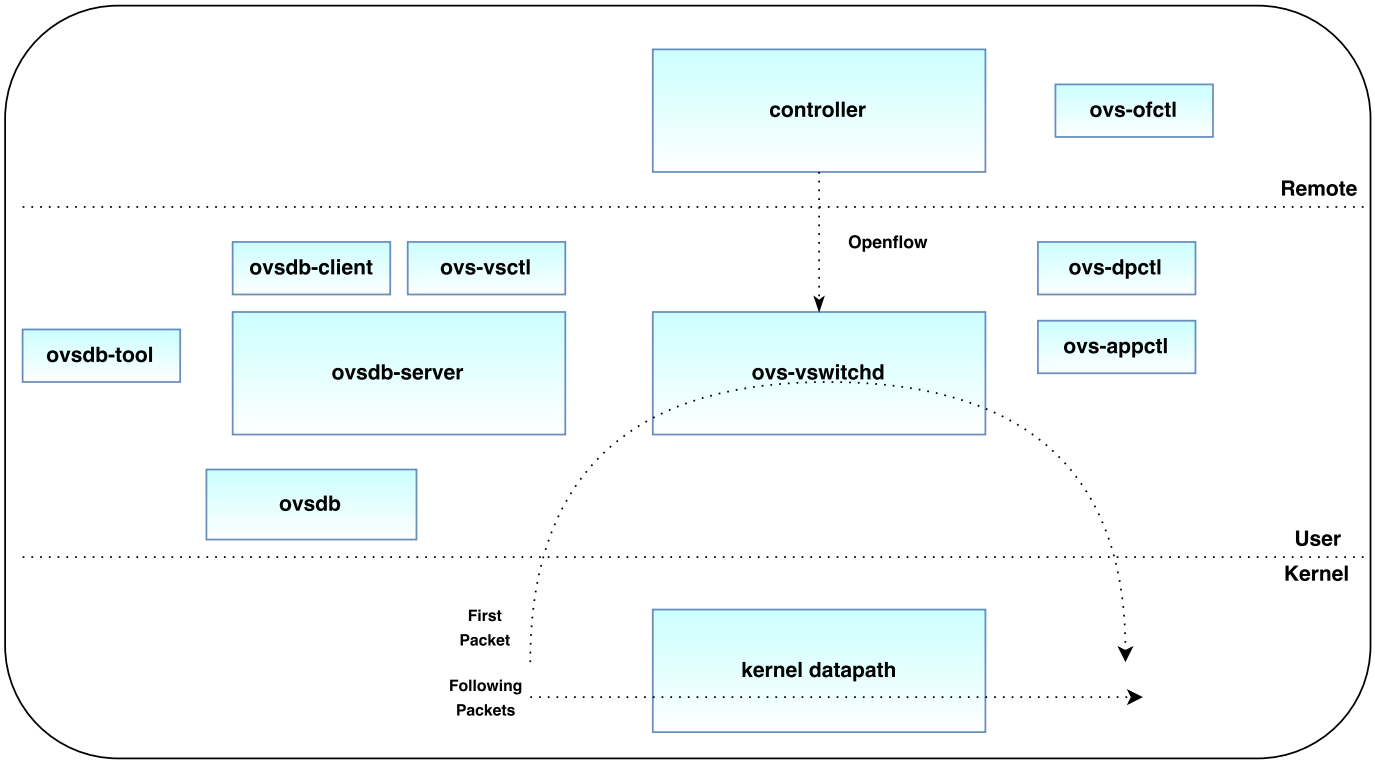
- **Kernel Datapath**: The kernel datapath or the kernel module is the host operating system implementation of an exact match cache. It is unaware of Openflow or the state of the switch. It is kept simple for highly performant packet lookup and forwarding. The ovs-vswitchd receives the first packet from 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).

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 [PPK+15]. In addition to the above key components, the implementation provides several management utilities:

- *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 -
- *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.
- *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.
- *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.
- *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.
- *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.

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.

Figure 4.2: OVS components

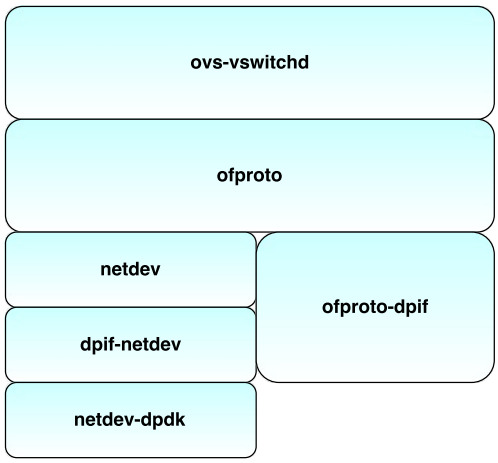


The interplay of the components of the Open vSwitch is shown in figure 4.3. The two interfaces that specify the handling of the packets are:

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 *dpif-netdev* module. The dpif provides an interface with abstractions for three key components:

- 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 *struct netdev*, with *netdev-dpdk* specifically handling dpdk ports.
- Flow table: A flow table has three key entities: flow - which consists of L2,L3, and L4 headers. The datapath flow uses the *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.
- 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 *dp_packet*.

Figure 4.3: OVS overview



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

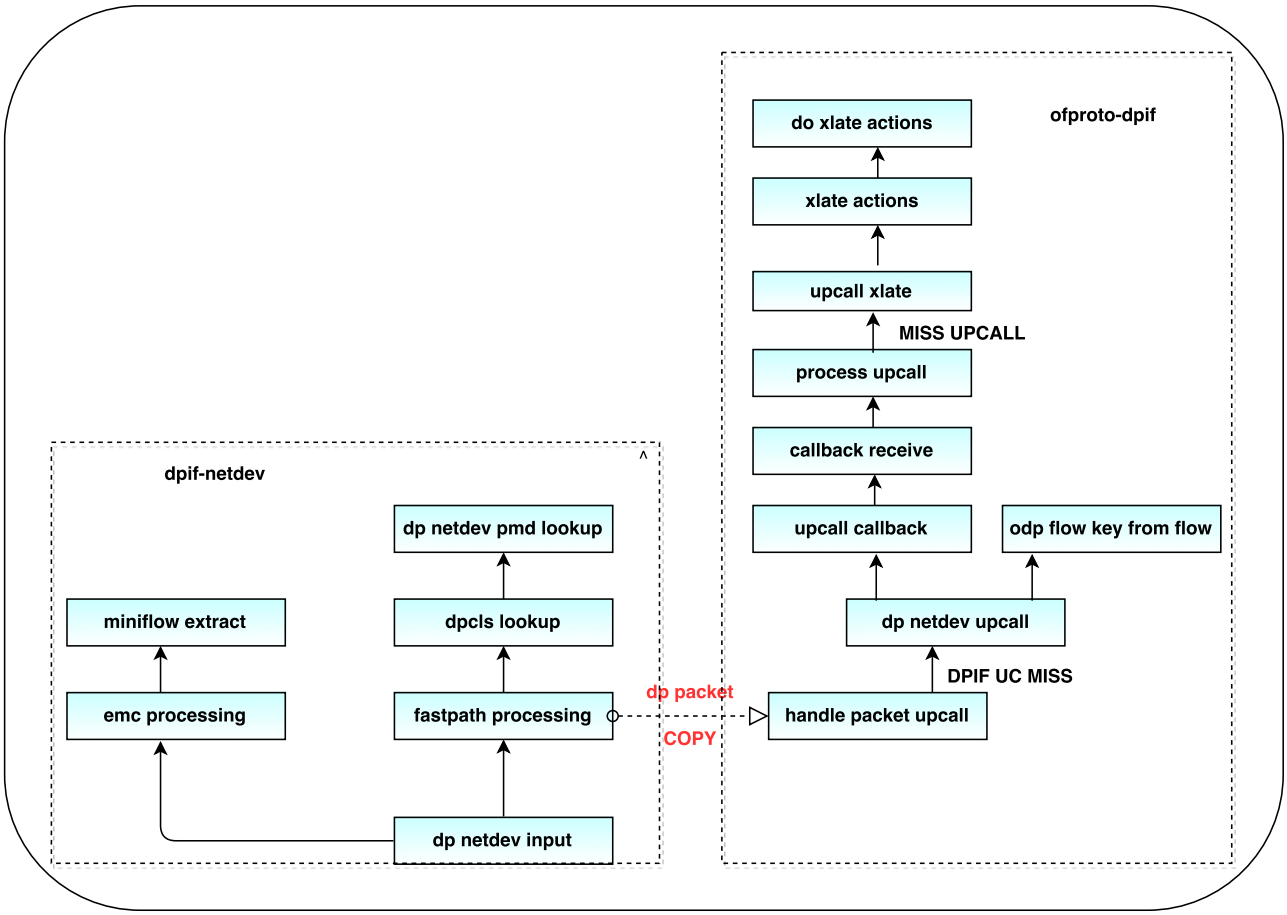
- oproto-dpif: This module implements the main provider of the Openflow abstraction. It is responsible for installing and removing datapath flows, monitoring and maintaining statistics etc. The ofproto.dpif module also is responsible for deciding whether a flow is to be cached or not.
- ofproto-dpif-upcall: 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.
- ofproto-dpif-xlate: This module performs the role of looking up the Openflow processing pipeline, extracting Openflow actions and translating into datapath actions.

4.4 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:

- Extraction of events from the payload: While work done by Mekky et. al [MHM+14] 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.
- 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.
- 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.

Figure 4.4: OVS dataflow: datapath to ofproto



- 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.
- Creating keys for event attributes: The *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.
- 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.
- 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.

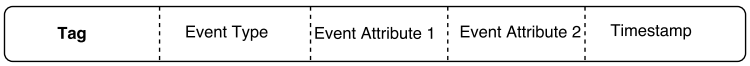
The design overview presented above aims to create a sophisticated programmable event processing framework on the network, which can complement the event processing engines to achieve latency sensitive results, reduce the burden of computation and enable staged processing. In the coming sections, more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

4.5 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.

Figure 4.5: Event Payload



4.5.1 Event Detection Semantics

Let 'E' be an event of type 't' and attributes 'a_1' and 'a_2'.

$$E\{t, a_1, a_2\} \tag{4.1}$$

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

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

Event detection operations can be combined with logical operators:
Conjunction: Event is signalled when all constituent input patterns evaluate to True.
Disjunction: Event is signalled when atleast one constituent input pattern evaluates to True.

The detect operations that are implemented are notated as:

$$D(e.t) \quad | \quad stream \tag{4.2}$$

$$D(e.t) \quad | \quad filter \tag{4.3}$$

$$D(e.t \wedge e.a_1) \quad | \quad stream \tag{4.4}$$

$$D(e.t \wedge e.a_1) \quad | \quad filter \tag{4.5}$$

$$D(e.t \wedge e.a_2) \quad | \quad stream \tag{4.6}$$

$$D(e.t \wedge e.a_2) \quad | \quad filter \tag{4.7}$$

$$D(e.t \wedge (e.a_1 \wedge e.a_2)) \quad | \quad stream \tag{4.8}$$

$$D(e.t \wedge (e.a_1 \wedge e.a_2)) \quad | \quad filter \tag{4.9}$$

$$D(e.t \wedge (e.a_1 \vee e.a_2)) \quad | \quad stream \tag{4.10}$$

$$D(e.t \wedge (e.a_1 \vee e.a_2)) \quad | \quad filter \tag{4.11}$$

where D is the detect operation;
$|$ denotes the redirect operation;
stream is the logical stream to which the detected event is redirected to.
filter is the operation to filter the detected event

4.5.2 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:

$$\oplus \ni \{<=, >=\} \tag{4.12}$$

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

$$D(e.t \wedge (e.a_1 \oplus value) \vee e.a_2) \quad | \quad filter \tag{4.13}$$

$$D(e.t \wedge (e.a_1 \oplus e.a_2) \quad | \quad filter \tag{4.14}$$

4.5.3 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.

$$D(e.t \wedge (e.a_1 <= \quad \rightarrow maxvalue)) \quad | \quad filter \tag{4.15}$$

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

$$D(e.t \wedge (e.a_1 <= \quad \overrightarrow{win} \quad maxvalue)) \quad | \quad filter \tag{4.16}$$

Similar to 4.15, all the events with attribute lower than the given *maxvalue* are filtered. However, on the detection of an event with a value higher than the given *maxvalue*, the event is forwarded and the newly seen value becomes *maxvalue*. The *maxval* increases for a window of *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.

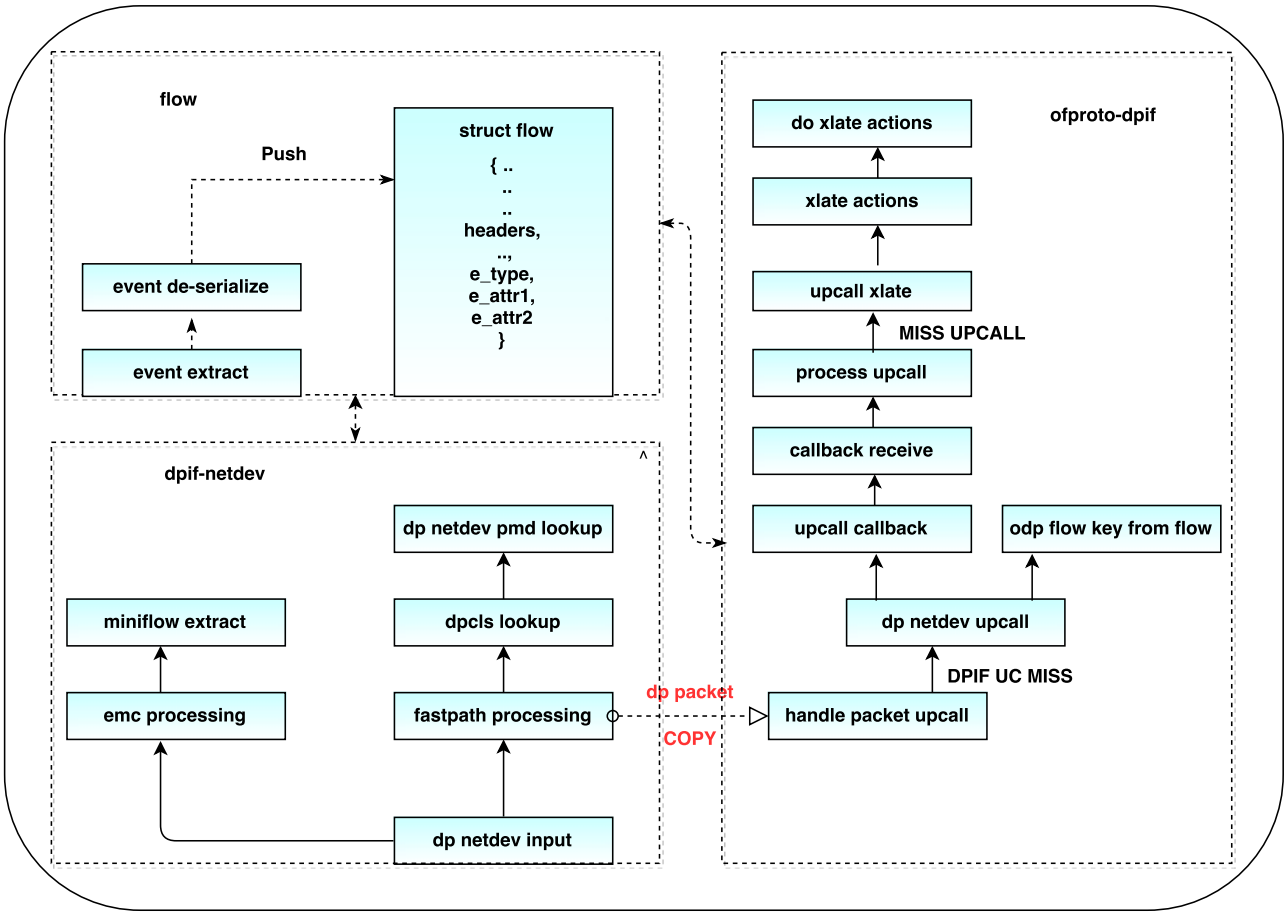
4.6 Implementation walk-through

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.

4.6.1 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 *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.

Figure 4.6: Event extraction and access



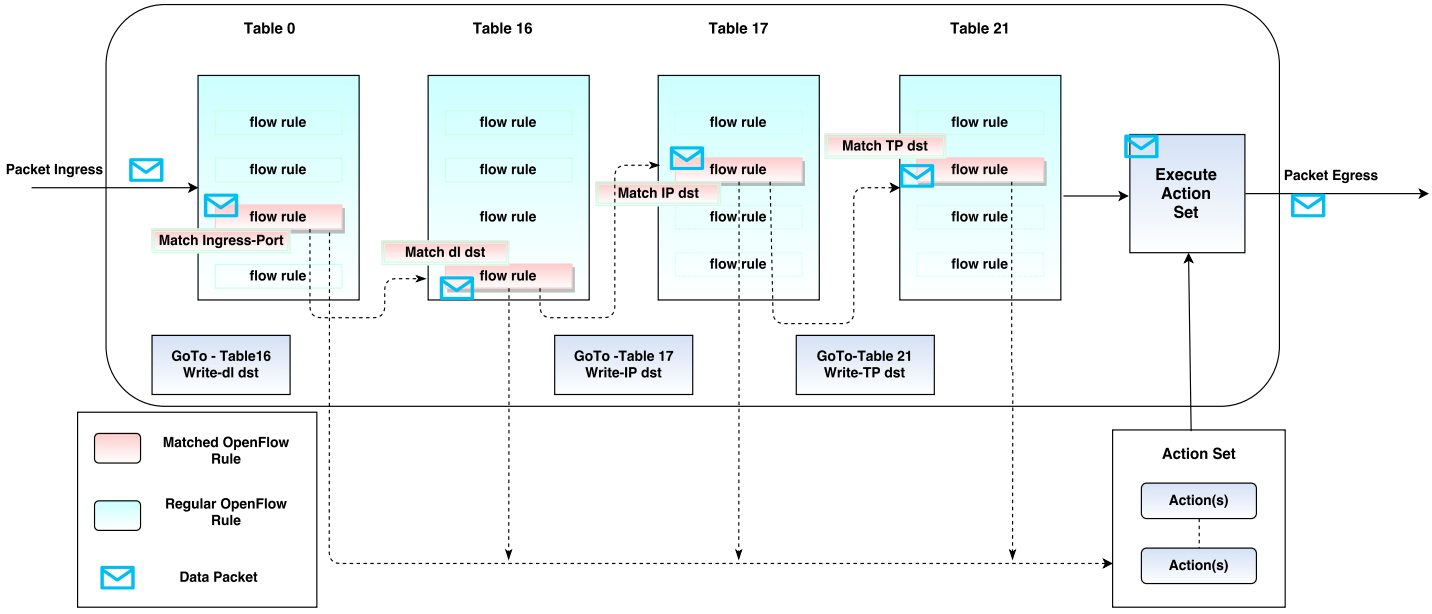
4.6.2 Accessing event data

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 *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 *struct flow_wildcards* which follows the same structure as flow, except that each 1-bit in wildcard indicates that the corresponding bit in *struct flow* must match. For the event data items, only an exact match is supported. Hence all the bits of the struct *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.

```
1 struct flow {
2 ..
3 ..
4 ..
5 /* L4 (64-bit aligned) */
```

```
6  ovs_be64 e_type;
7  ovs_be64 e_attr1;
8  ovs_be64 e_attr2;
9  ..
10 ..
11 }
12
13 FLOWMAP_SET(map, e_type);
14 FLOWMAP_SET(map, e_attr1);
15 FLOWMAP_SET(map, e_attr2);
16
17
18 miniflow_push_be64(mf, e_type, htonll(event[0]));
19 miniflow_push_be64(mf, e_attr1, htonll(event[1]));
20 miniflow_push_be64(mf, e_attr2, htonll(event[2]));
```

Figure 4.7: Openflow Pipeline

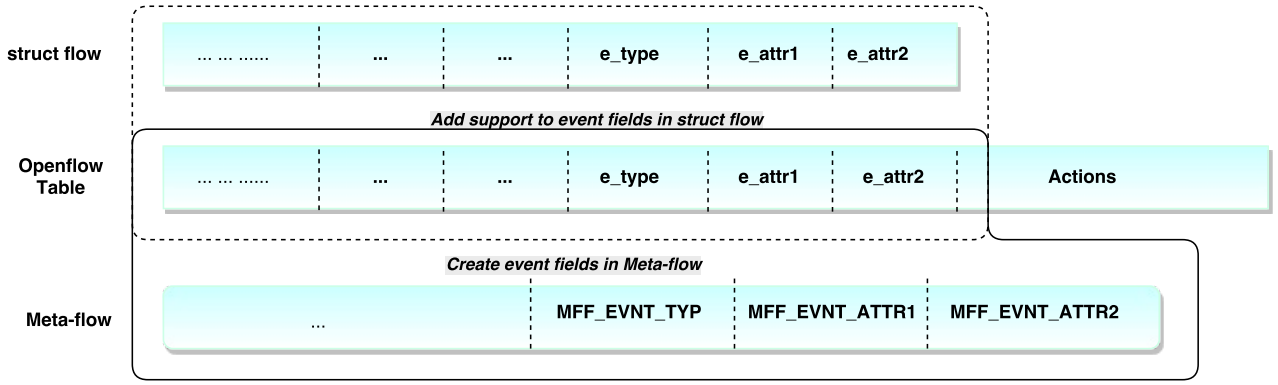


4.6.3 Modelling Openflow pipeline for event processing

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.

Figure 4.8: Openflow Flowtable - Add event data fields



4.6.4 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.

```
1  /* "e_type".
2  *
3  * Event Type.
4  *
5  * Type: be64.
6  * Maskable: bitwise.
7  * Formatting: decimal.
8  * Prerequisites: UDP.
9  * Access: read/write.
10 * NXM: NXM_OF_EVNT_TYP(113) since v2.6.
11 * OXM: OXM_OF_EVNT_TYP(46) since OF1.2 and v2.6.
12 * OF1.0: exact match.
13 * OF1.1: exact match.
14 */
15 MFF_EVNT_TYP,
```

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.

```
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;
  ..
  ..

}
```

4.6.5 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

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

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

- **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 *struct ofp_util_flow_mod*.
- **ofp-util interface**: The ofp-util interface defines the the *struct ofp_util_flow_mod* to consume and respond to the flow-mod commands. The struct *struct ofp_util_flow_mod* is composed of a struct match as a member.
- **match interface** The match interface provides the data structure for flow classification - *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 *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 *struct match* are also defined. This is used by the ofp-parse interface to populate *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:

```
void
match_set_event_type_masked(struct match *match, ovs_be64 e_type, ovs_be64 mask){
```

```
3 match->flow.e_type = e_type & mask;
4 match->wc.masks.e_type = mask;
5 }
```

- **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.

Figure 4.9 shows how a OFPFC_ADD command with event fields is parsed and normalized into the match interface using *struct ofputil_flow_mod*. To enable correct parsing of the event fields, Openflow 1.0 support is configured in Openflow-1.0 header file:

```
1 enum ofp10_flow_wildcards {
2 ..
3 OFPFW10_EVNT_ATTR1 = 1 << 22,
4 OFPFW10_EVNT_ATTR2 = 1 << 23,
5 OFPFW10_EVNT_TYP   = 1 << 24,
6 ..
7 }
```

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.

Figure 4.9: Openflow: Parse event fields

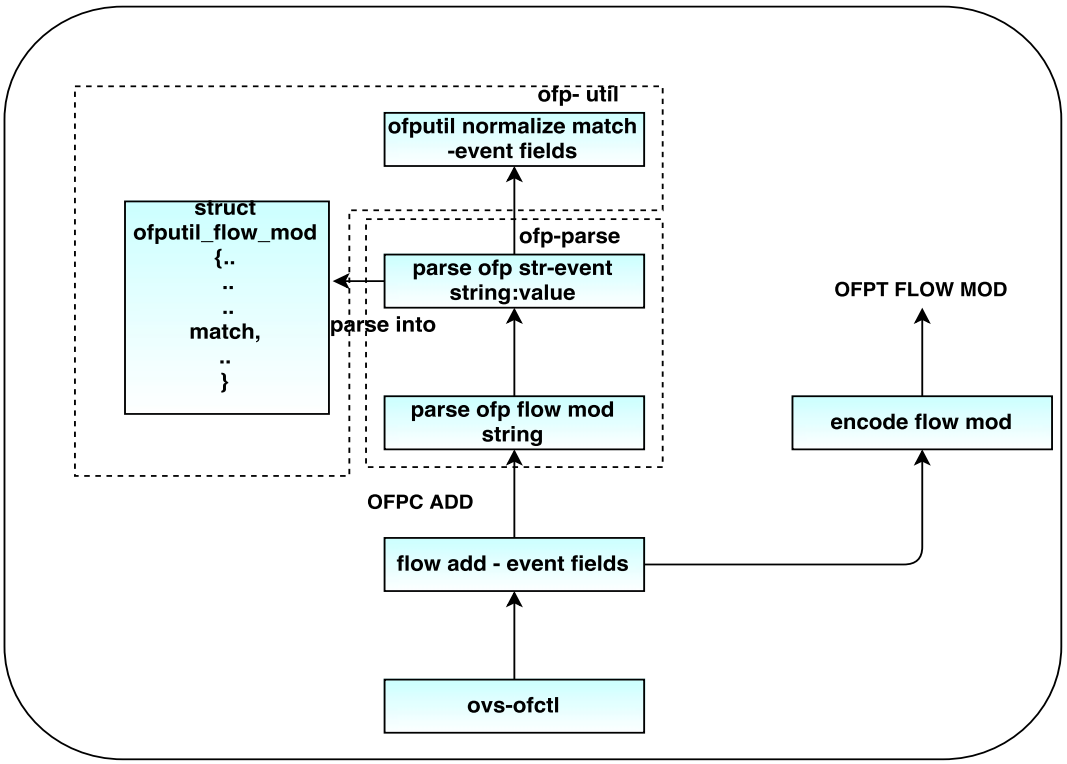
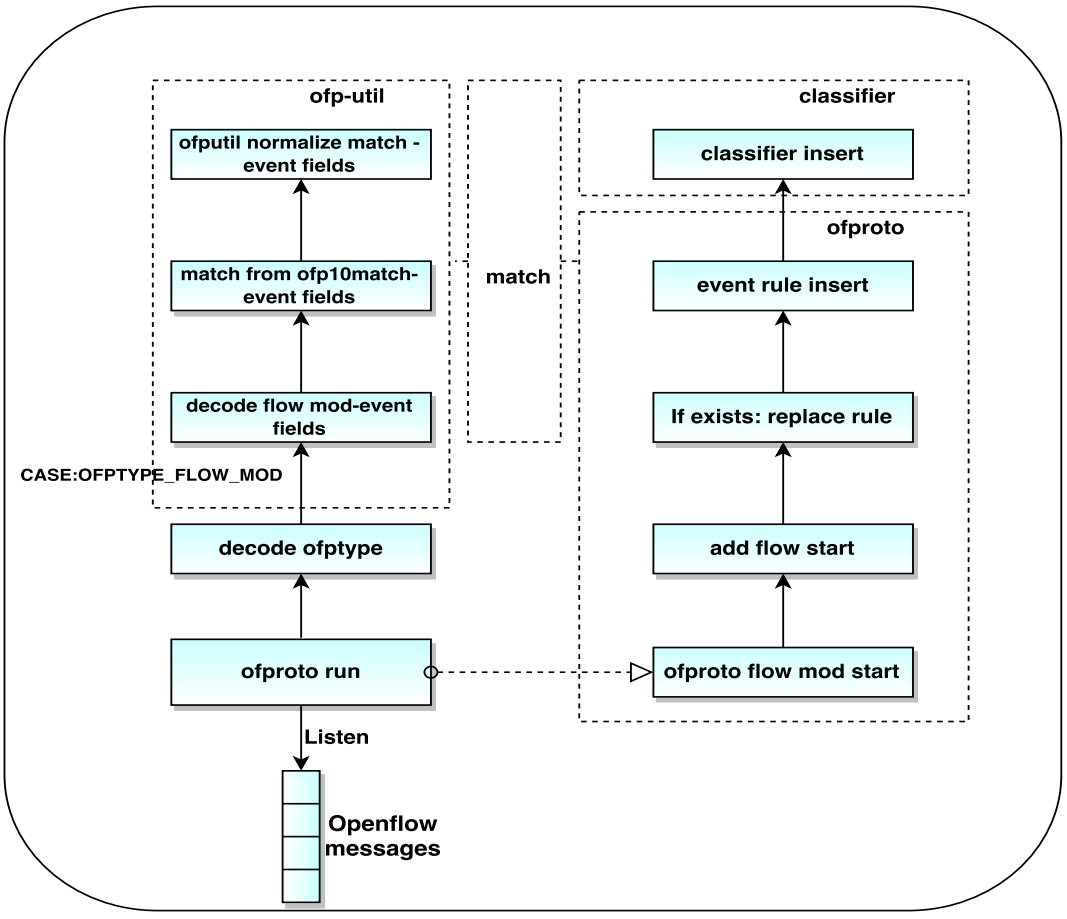


Figure 4.10 shows an Openflow message of type OFPTYPE_FLOW_MOD is parsed to extract the relevant rules and added into the classifier. The ovs-vswitchd daemon actively polls for Openflow messages. On receiving a message, the message type is decoded. If the message is of type 'OFPTYPE_FLOW_MOD' the message 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:

```
1 struct ofp10_match {
2 ..
3 ovs_be64 e_attr1;
4 ovs_be64 e_attr2;
5 ovs_be64 e_type;
6 }
```

Once the *struct op10_match* is created, the relevant event fields are extracted and put into a *struct match* and normalized. The ofpbuf is not directly pulled into *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.

Figure 4.10: Openflow: Event rule creation

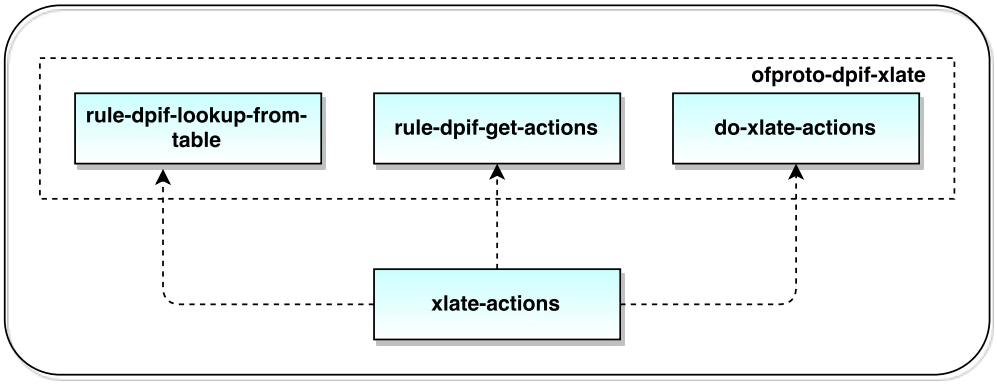


At this point the implementation supports programmable event rules and support form the three event fields specified by the system model. However the event detection mechanism is yet to be implemented. In the next subsection, a closer look into the implementation steps needed to support matching based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

4.6.6 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.

Figure 4.11: ofproto: Rule look-up



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.

```
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;
};
```

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

```
/* 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);
```

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.

4.6.7 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:

- **set_max** : A *set_max:val* action ensures that any event with attribute value higher than *val* is filtered out for a particular event type.
- **set_min** : A *set_min:val* action ensures that any event with attribute value lower than *val* is filtered out for a particular event type.

The *set_max, 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 *set_max* action used for illustration.

```

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. */
};
```

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.

```
case OFPACT_SET_MAX:{
  if(htonll(flow->e_attr1) <= ofpact_get_SET_MAX(a)->attr){
    xlate_normal(ctx);
  }
  break;
}
```

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:

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

The *attr_cmp* action provides the implementation of operations notated in 4.14.

4.6.8 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:

- **mov_max**: A *mov_max:maxval* action filters the event if the attribute is lower than *maxval*. However if the attribute value higher than *val* is encountered, the event is forwarded and the newly seen value becomes the new *maxval*. This action provides implementation support to the operation notated in 4.15. Below the pseudo-code of the *mov_max* action is shown:

```
  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

```

- **set_win,win_max**: The combination of *set_win:win,win_max:maxval* actions performs a similar role done by the *mov_max*. However the *maxval* is incremented only for the specified window *win*, after which it remains constant. To understand this action, let us look at an example for event type 'TEMP', with action *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 *maxval*. And going forward, a 'TEMP' event with value 58 will be filtered. This increment of *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 *maxval*. This action provides implementation support to the operation notated in 4.16. The below pseudo-code glimpses through the implementation.

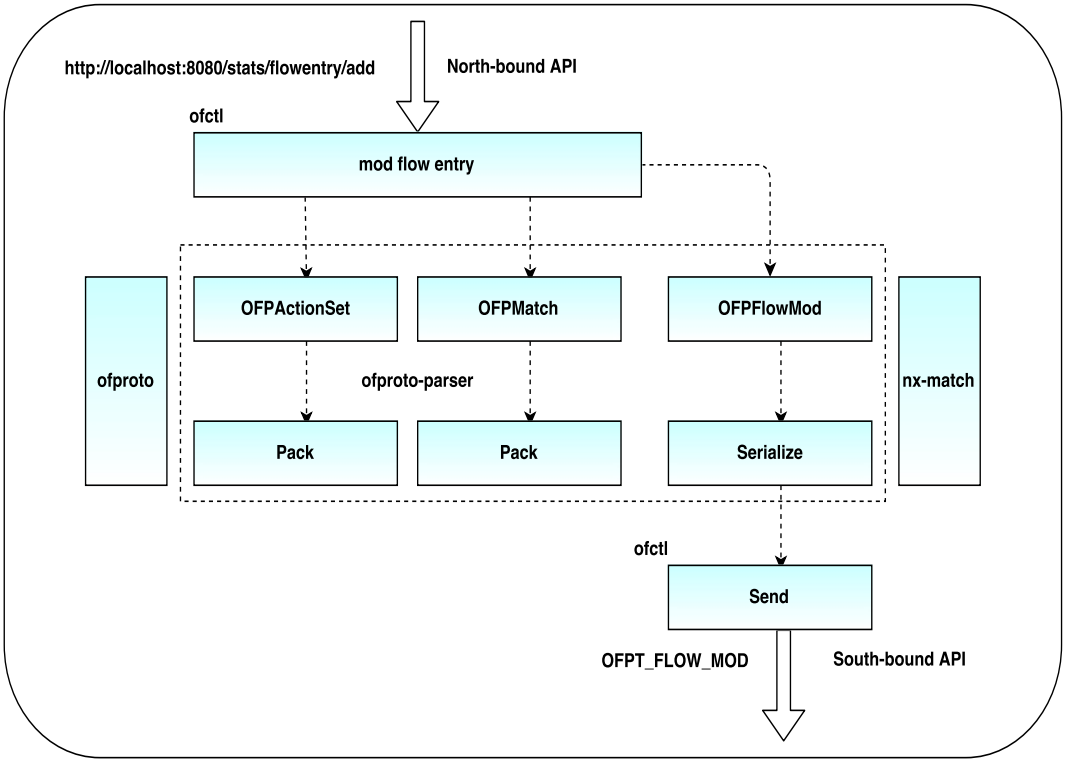
```
case OFPACT_SET_WIN:
window= ofpact_get_SET_WIN(a)->attr
winItem=search_window(type)
if(winItem == NULL):
  insert_window(type, window, 0)
else:
  delete_window(winItem)
  insert_window(type, window, ++(winItem->counter))
break;

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
```

4.6.9 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 *OFPT_FLOW_MOD* message.

Figure 4.12: RYU: North to South - Event Rule creation



The implementation is broken down into four parts:

- **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.

```
1  _OFP_MATCH_PACK_STR = 'IH' + OFP_ETH_ALEN_STR + 's' + OFP_ETH_ALEN_STR + \
2  'sHBxHBB2xIIHHQQQQ'
3
4  OFPAT_SET_WIN_MAX = 12  # window moving max value for attribute1.
5  OFPAT_SET_MOV_MAX = 13  # moving max value for attribute1.
6  OFPAT_SET_MIN = 14      # min value for attribute1.
7  OFPAT_SET_MAX = 15      # moving max value for attribute1.
8  OFPAT_SET_WIN = 16      # window value
9  OFPAT_SET_CMP = 17      # compare mode
10
11 OFPFW_EVNT_ATTR1 = 1 << 22
12 OFPFW_EVNT_ATTR2 = 1 << 23
13 OFPFW_EVNT_TYP = 1 << 24
14
```

```
1  struct ofpact_map of10[] = {
2  ..
3  { OFPACT_SET_WIN_MAX, 12},
4  { OFPACT_SET_MOV_MAX, 13},
5  { 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,
}

```

- **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.

```

@_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)

```

- **ofctl**: The ofctl modul consists of the implementation for the *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.

```
#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))
```

```
7   actions.append(dp.ofproto_parser.OFPActionSetMax(val))  #call OFPActionSet
8
```

- **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 *OFPT_FLOW_MOD* message which is defined in position 14 within the Open vSwitch, and hence is packed similarly in RYU.

```
1     OFPT_FLOW_MOD = 14       # Controller/switch message
2
```

In Open vSwitch:

```
1     enum ofpraw {
2     /* OFPT 1.0 (14): struct ofp10_flow_mod, uint8_t[8][]. */
3     OFPRAW_OFPT10_FLOW_MOD}
4
```

The below snippet, shows the methods added to parse and serialize individual event *set_min* action.

```
1   class OFPActionAttrVal(OFPAction):
2   def __init__(self, val):
3     super(OFPActionAttrVal, self).__init__()
4     self.val = val
5
6   @classmethod
7   def parser(cls, buf, offset):
8     type_, len_, val = struct.unpack_from(
9     ofproto.OFP_ACTION_ATTR_VAL_STR, buf, offset)
10    assert type_ in (..,
11    ofproto.OFPAT_SET_MIN,
12    ..)
13    assert len_ == ofproto.OFP_ACTION_ATTR_VAL_SIZE
14    return cls(val)
15
16  def serialize(self, buf, offset):
17    msg_pack_into(ofproto.OFP_ACTION_ATTR_VAL_STR,
18    buf, offset, self.type, self.len, self.val)
19
20
21  @OFPAction.register_action_type(ofproto.OFPAT_SET_MIN,
22  ofproto.OFP_ACTION_ATTR_VAL_SIZE)
23
```

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.

```
1 uri = path + '/flowentry/{cmd}'
2 mapper.connect('stats', uri,
3 controller=StatsController, action='mod_flow_entry',
4 conditions=dict(method=['POST']))
```

4.7 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.

5 Evaluation and Results

In this chapter the EVS bridge is evaluated for performance and function. The chapter describes the modes of deployment used for evaluation and the baselines used for measurement and comparison. Wherever eligible a brief discussion of the result is provided with necessary parameters.

5.1 Evaluation Environment

In Table 5.1, the components working together to form the evaluation environment are listed. At the core of the set up is an Open vSwitch built from the tree with release tag 2.6.1 accelerated by Intel Data Plane Development Kit built from the source with release tag 16.11.2. The hypervisor support comes from Linux-KVM with QEMU 2.9.0 for emulation, running on an Intel(R) Xeon x86_64 CPU with 16 cores. The operating system used is Ubunutu xenial 16.04.2 LTS running a 4.4.0.79-generic kernel.

Table 5.1: Evaluation Environment

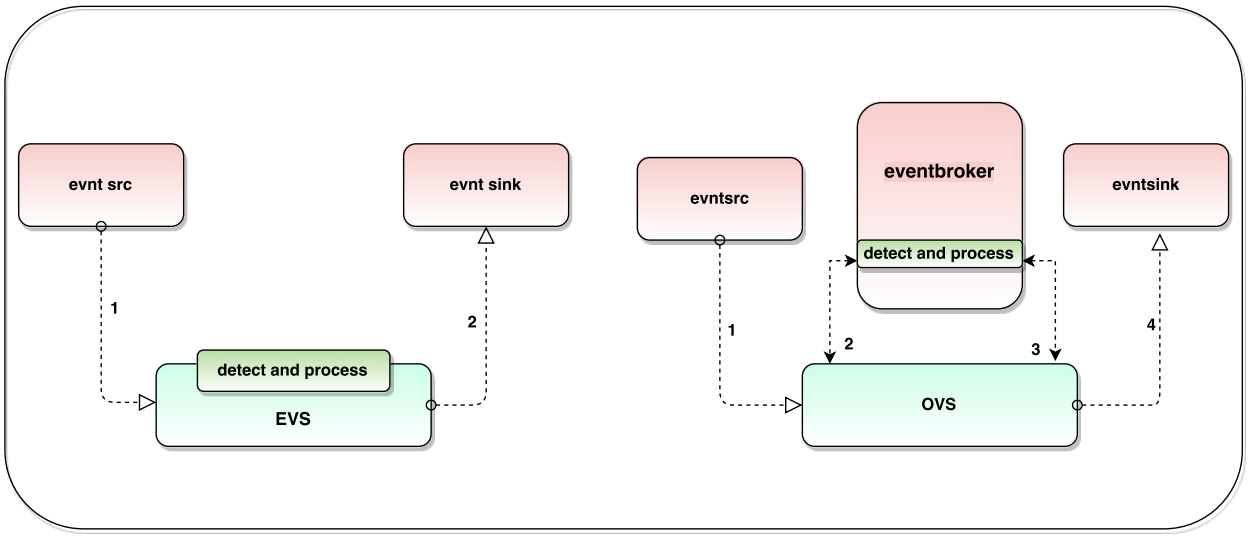
Processor	Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
Architecture	x86_64
CPU(s)	16
Kernel	4.4.0.79-generic
Distribution	Ubuntu xenial 16.04.2 LTS
Open vSwitch	2.6.1
Intel DPDK	16.11.2
Qemu	Qemu Emulator 2.9.0
Ryu	4.0

5.2 System under Test

The system under test during the evaluation is the Event Processing enabled Open vSwitch(EVS) bridge. Firstly the performance of the EVS bridge is measured in terms of link bandwitdh and average round trip time and compared with that of a standard OVS bridge. This is important because an addition of functionality to the OVS bridge should not result in a performace degradation that will make the OVS bridge unusable for the generic switching scenarios it was originally designed for.
Secondly, the EVS bridge is evaluated with respect to the problem statement - described in Chapter 1.2 - that there is a performance gain to be attained when aspects of event processing are offloaded on to the underlying network. To evaluate the performance, an application(evntsrc) is implemented to generate events, and another application(evntsink) is developed to consume the packets and measure latency. Standard packet generators cannot be used for evaluation because of the need to control the format of the packet payload. Performance of the EVS bridge is compared against a custom light-weight userspace application(evntbroker) emulating an event processing engine. The evntbroker application bridged on the standard OVS. The flow-path of data in the two systems is captured in figure 5.1. We can see that EVS bridge replaces the udpbroker and reduces the number of packets in the system. Although the evntbroker does not preform the role of full-fledged complex event processing engine, it is sufficient

to provide a baseline for the event processing operations built in to the EVS bridge. The point-to-point latency between evntsrc and evntsink is measured in both the systems- under two deployment scenarios - and the results discussed.

Figure 5.1: Data Flow in EVS vs OVS



Thirdly, we measure the performance of the EVS bridge for the following additional parameters:

- Measure of latency with increasing size of event types.
- Measure of latency with increasing number of event types.
- Measure of latency with increasing percentage of filtered flows.
- Measure of latency with increasing number of matched event attributes.
- Measure of latency with event attributes detection and redirection.
- Measure of latency with compare operations performed on event attributes.

The Evaluation is performed in two deployment modes:

- Network Namespaces: As discussed in Chapter 3.7, a network namespace is a logically separate network stack within the same host machine. Since each namespace has its own network stack and can be attached with virtual Ethernet or physical ports, they offer a perfect development environment for Open vSwitch. During the evaluation phase, namespaces were bridged on Open vSwitch using TAP devices. The set-up, methodology and results are discussed in detail in section 5.4
- Qemu-KVM Guests: As discussed in Chapter 3.8, Qemu-KVM provide a combination of device emulation and virtual machine monitoring to set up guest virtual machines in a physical host. The evaluation set up involves Open vSwitch accelerated by Intel DPDK as a hypervisor switch bridging QEMU-KVM guests. The set-up, methodology and results are discussed in detail in section 5.5.

5.3 Apparatus for Evaluation

As briefly discussed in the previous chapter 5.2 and illustrated in figure 5.1, the three components needed for the evaluation are:

- Event Source: The EVS bridge implementation handles events only if the packets have a custom tag in the packet. All other packets without the tag are sent through the standard processing flow. Once the tag is detected, the EVS bridge parses the packet payload to derive the event types and event

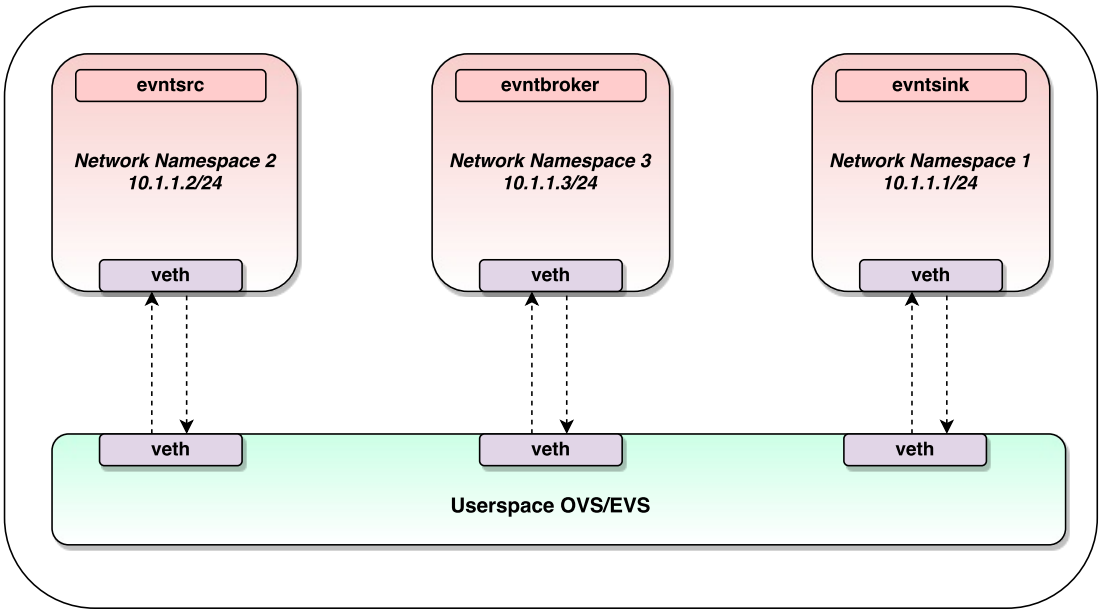
attributes. Hence the packets have to follow the format *[tag,event_type,event_attribute1,event_attribute2]*. To this end, a Java based Event Source generator, *evntsrc*, has been implemented to generate UDP packets in the established format.

- Event Sink: The events are consumed by an event sink which is responsible for measuring the point to point latency from source to sink. The *evntsink* application has been implemented to consume events and measure the latency.
- Event Broker: A custom event broker, *evntbroker*, is implemented to perform the operations built-in to the EVS bridge. The evntbroker provides an alternate flow-path for events to that of the flow-path in the EVS bridge. And the flow-path of data is similar to that of any event processing engine. Hence the flow-path provided by evntbroker is referred to as a baseline for evaluation.

5.4 Evaluation on Network Namespaces

In this section we evaluate the performance of userspace-EVS against userspace-OVS. As described in Chapter 5.3, EVS bridge set-up does not include an evntbroker, whereas the standard OVS set-up includes a evntbroker. This is because EVS bridge is equipped to perform the operations of the evntbroker. Figure 5.2 illustrates the set-up for network namespaces. The namespaces are connected to the vSwitch(EVS or OVS) using a pair of virtual Ethernet interfaces. The evntsrc, evntsink, and evntbroker applications are all running in their own L3 network namespace. More about the set up of veth-pairs, the format and rate of the generated UDP packets, and the API signatures used to deploy relevant event processing rules are discussed in subsection 5.4.1.

Figure 5.2: Namespaces on EVS/OVS



5.4.1 Set-up Methodology

In this subsection the steps taken to set up the Network Namespaces bridged on the two userspace bridges is detailed. To begin the set up, it is assumed that the EVS code base is compiled and the ovs-vswitchd daemon and the ovsdb-server are up and running. In addition the modified RYU controler is also running with OpenFlow v1.0. With this in place, the following steps are taken to set-up the system.

- To begin the set up, the userspace EVS bridge has to be initialized. See the following command:

```
$ ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
```

- The namespaces are set up. In this document, the set up for one namespace is shown.

```
$ ip netns add ns1
```

- Two veth pairs of ports are created.

```
$ ip link add tap1 type veth peer name ovs-tap1
```

- One end of the created pair is attached to the EVS bridge.

```
$ ovs-vsctl add-port br0 ovs-tap1
```

- The other end of the pair is attached to the network namespace.

```
$ ip link set tap1 netns ns1
```

- The interface in the network namespace is statically assigned with an ip and both the ends of the pairs are brought up.

```
$ ip netns exec ns1 ip link set dev tap1 up
$ ip netns exec ns1 ip addr add 10.1.1.1/24 dev tap1
$ ip netns exec ns1 ip link set lo up
$ ip link set dev ovs-tap1 up
```

- The evntsink, evntsrc and evntbroker applications are run on the three namespaces.

```
$ ip netns exec ns1 java evntsink
$ ip netns exec ns2 java evntsrc
$ ip netns exec ns2 java evntbroker
```

- Install the rules onto the EVS bridge to handle the event operations. The API signature for detecting and event of type 'TEST' and redirecting it the evntsink at 10.1.1.1 is shown below:

```
 {
"dpid": 178974088016461,
"table_id": 0,
"priority": 11112,
"flags": 1,
"match":{
```

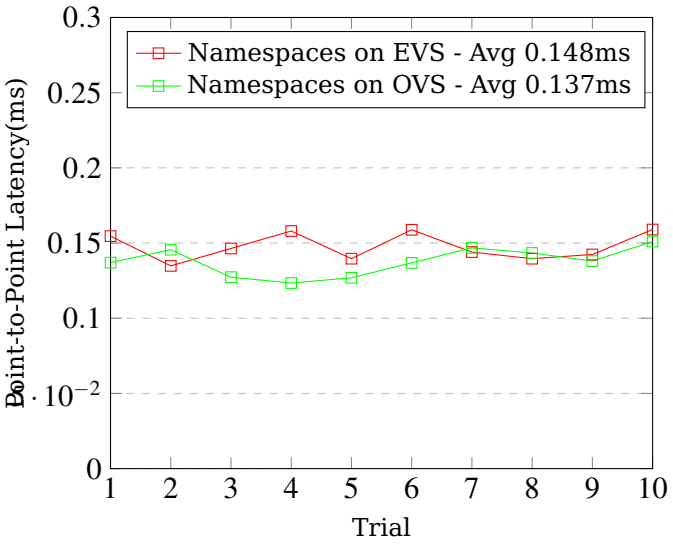
```
"dl_type":0x0800,
"nw_proto":17,
"nw_dst":"10.1.1.1",
"tp_dst":9877,
"e_type":"TEST"
},
"actions":[{
"type":"set_nw_dst",
"nw_dst": 10.1.1.1
},
{
"type":"NORMAL"
}
]
}
http://localhost:8080/stats/flowentry/add
```

The latency measurements are produced at the evntsink application. It is important to understand the methodology for measurement. Network namespaces are purely separate logical networks. They run on the same CPU and use the Hardware. Technologies such as dockers, linux containers combine facilities provided by network namespaces, process namespaces, file system namespace, IPC namespaces among others with linux control groups to provide virtualization containers. But network namespaces simply provide a logical network stack. To measure the latency, the evntsrc application sends the system timestamp in the payload. The payload follows format specified in figure 4.5. On receiving the packet, the eventsink extracts the timestamp from the payload, computes the difference with current system time. The delta value is aggregated for 1000 UDP packets in each run to give one trial reading of the point-to-point latency. The same methodology is used across all experiments.

5.4.2 Performance measurement without event operations

In this subsection, the evaluation of latency between the evntsrc and evntsink connected without performing any event detection and redirection operation is detailed. Here the EVS bridge performs the role of a standard OVS bridge because no rules specific to events are installed. The packets are sent directly from evntsrc to evntsink without the need for evntbroker. The same exercise is repeated for the standard OVS bridge in order to contrast the performance of the two bridges. The observed results of this analysis are summarized in Figure 5.4. As we can see, the point-to-point latency between evntsrc and evntsink on the EVS bridge is only slightly greater than that of the point-to-point latency of the same on the standard OVS bridge. The slight increase in latency can be attributed to the fact that the EVS bridge performs deep packet inspection per packet to parse and de-serialize the payload data.
In addition to point-to-point latency, the bandwidth of the link, and average round trip time (RTT) is measured for both EVS and OVS bridge. To measure the bandwidth of the link, *hperf3* is used and to measure RTT *hping3* is used. As we see in Figure 5.1, EVS bridge is similar to the standard OVS bridge when compared on the parameters of link bandwith and RTT. The bandwidth of the link is low in both cases because namespaces are bridged using virtual interfaces and expectedly have low throughput.

Figure 5.3: Performance of bridged namespaces



	OVS Bridge	EVS Bridge
Bandwidth	154 Mbits/s	149 Mbits/s
Avg RTT	0.183 ms	0.180 ms

5.4.3 Performance measurement with event detection and redirection

In this subsection the performance evaluation of the bridge whilst performing an event detection and redirection operation on a single event type is presented. For example, Let us say an event with type 'TEST' is sent from evntsrc to the evntbroker. This operation is notated as:

$$D(e.t) \quad | \quad stream, \tag{5.1}$$

where D is the detect operation;
| denotes the redirect operation;
and *stream* is the logical stream to which the detected event is redirected to.

Whereas other streaming applications may have named stream support, EVS stream redirection is done based on network addresses of the application consuming the event stream. The controller rule installed in EVS corresponding to this operation is:

```
{
"dpid": 178974088016461,
"table_id": 0,
"priority": 11112,
"flags": 1,
"match":{
"dl_type":0x0800,
"nw_proto":17,
"nw_dst":"10.1.1.1",
"tp_dst":9877,
"e_type":"TEST",
},
"actions":[{
"type":"set_nw_dst",
```

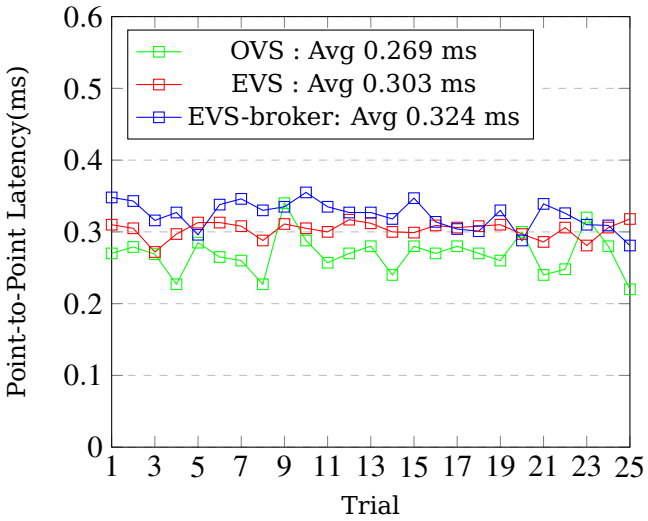
```
"nw_dst": 10.1.1.1
},
{
"type":"NORMAL"
}
]
}
http://localhost:8080/stats/flowentry/add
```

Consider a stream of TEST events where each event has readings from multiple sensors which has to be redirected to another processing engine specifically built for handling these events using an Event Processing language. Such a query is in essence similar to the operation defined in (5.1) and would normally look like:

```
INSERT INTO NEWTESTPSTREAM
SELECT * FROM TESTSTREAM
```

In the EVS bridge, this event of type 'TEST' is detected and redirected to the evntsink using *mod_nw_dst* action rules. Hence the event doesn't go to the evntbroker when EVS bridge is used. As discussed before streams in EVS are identified using network addresses. Named stream support isn't provided. Although it is possible to create custom actions with custom names and make them perform the *mod_nw_dst* actions, in a way mimicking the idea of 'named' streams, this would make a) add more lines of code in the already strong 0.5 million lines of code in OVS, and b) would mean custom action for every stream and thus wouldn't scale.

Figure 5.4: Performance of Event Redirection



The same experiment is repeated on the OVS bridge. But in this case, the event with type 'TEST' is simply forwarded to the evntbroker, which proceeds to detect the event and sends it to the relevant evntsink. As we can see in the graph in figure 5.5, the performance of event redirection at the EVS bridge is lower than the performance of event redirection done on an evntbroker, albeit marginally. Even though the EVS bridge prevents the packet from going into the evntbroker and reduces the number of hops for the event packet, the expected improvement in performance is not seen. This can be explained by two factors. One, the redirection in EVS bridge is performed by *mod_tp_dst* or *mod_nw_dst* or *mod_dl_dst*

depending on the layer. These Openflow actions modify the event packet and also redirect the individual event packet into the datapath interface, which in case of userspace EVS is dpif-netdev. This additional parsing and lookup balances out the benefits of removing the evntbroker. Two, the EVS bridge employs per packet event de serialization and parsing, which adds some overhead.

In addition to the above described experiments, the EVS bridge was set up with an evntbroker and evaluated to get a better sense of the performance. In this set up although the EVS bridge is used, the event detection rules are not installed. Instead the events are forwarded to the broker as if operating in the standard OVS mode. This results show far higher latency than the standard OVS bridge and slightly higher latency compared to the EVS bridge running in event processing mode. In the latter case, EVS in event processing mode has two fewer packet hops resulting in lower latency. In the former case, we clearly see that although the OVS bridge and EVS in evntbroker have equal number of packet hops, the per-packet parsing and de-serialization of events, as described before, adds an overhead. By comparing the three results, we get an idea of how much overhead is added by the per-packet event parsing and de-serialization and how much overhead is added by the context switch and the extra packet hops.

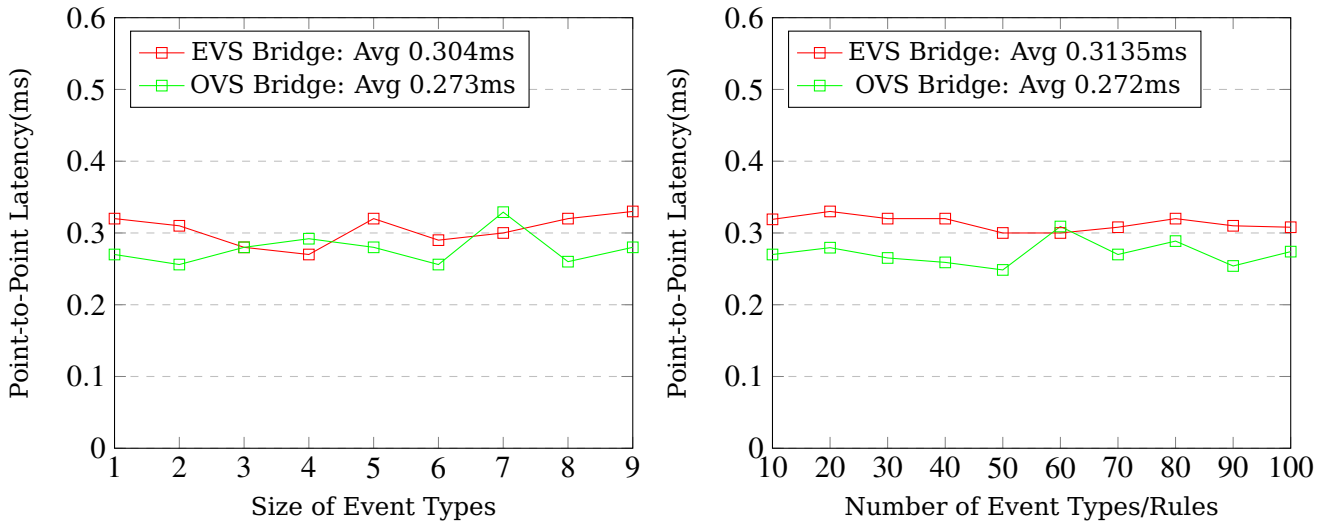
These results provides an interesting insight to the evaluation process in a virtualized environment. In a virtualized environment the costs added by the per-packet processing are going to be similar to the per-packet processing overhead in namespaces, but the costs added by context switch from host to VM , back to host are going to be a lot more expensive. There is more discussion on this topic in section 5.5.

5.4.4 Performance measurement with increasing size of event types

In this subsection an evaluation of the bridge is presented with an increasing size of event types. To do the same, the evntsrc generates events with types ranging from 1 to 9 characters. The analysis was done on the EVS bridge and on the OVS bridge. As seen in the left graph plotted in Figure 5.6, the increasing size of an event type string doesn’t result in a linear increase in latency in both the cases. For EVS bridge, this is explained by the fact that Tuple search space classifier of Open vSwitch creates a hash table for every unique combination of inserted flow rule. In the case of this evaluation, one flow rule is added for each run, while increasing the size of event type field in consecutive runs. Additionally, although the event type field matches against a string in the event payload, the match field during flow rule insertion for event type is Hexadecimal value of the string. The rationale behind this approach is that strings within an event payload are serialized into Hexadecimal values on the wire, and since event parsing is part of the packet processing pipeline, keeping the fields as-is avoids the expensive conversion, thus reducing the overhead on the processing pipeline.

In case of the OVS bridge, the broker does a simple string match, and increasing the size of the string makes little to no difference. It would therefore be interesting to see how adding additional event types would impact the performance of the two bridges.

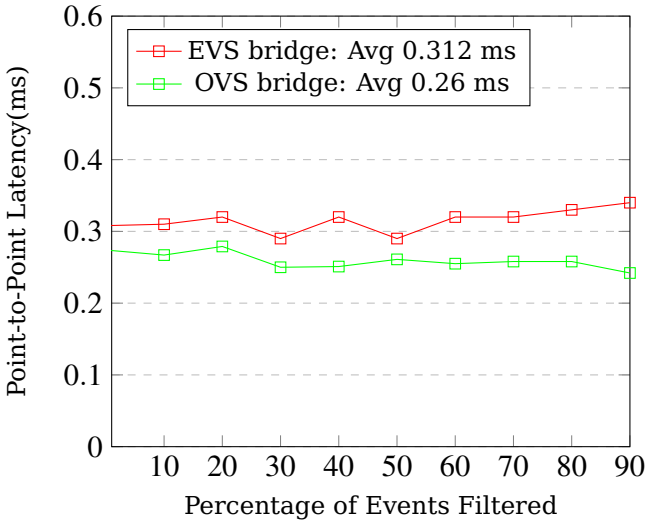
Figure 5.5: Changing Size and Number of Event Types



5.4.5 Performance measurement with increasing number of event types

In addition to size of event types, the number of event types sent to the EVS bridge were also increased to measure the impact on latency. Each event type is handled by one rule in EVS bridge, so to handle 100 event types and redirect them to evntsink the EVS bridge was set up with 100 rules. As see in the right graph in Figure 5.6, an increase in the number of event types/rules does not affect the performance of the EVS bridge. This is because the number of flow rules remain constant, only the actions on each flow rule is modified to increase filtered events. And since the Tuple Space Search Classifier only adds an additional Hashtable when there is an new combination of fields entered as a flow rule, even increasing the number flow rules, as long as they have the same match fields would not result in additional lookups. The same two experiment set ups were repeated for the standard OVS bridge, with an evntbroker responsible for handling and forwarding the event types to the evntsrc. EVS bridge and OVS bridge performed similarly in this regard showcasing the potential of the EVS bridge in real-world scenarios.

Figure 5.6: Performance of Event Filtering



5.4.6 Performance measurement with increasing percentage of filtered event types

In this subsection the performance evaluation of the EVS bridge with an increasing percentage of filtered event types is presented. The evntsrc is configured to generate 100 random event types, addressed to the evntbroker. The EVS bridge initially is configured with flow rules to send through all the events. At this stage the setup is similar to the sub-section 5.4.3 with 100 event types and flow rules. In the next stage 10% of the event types are filtered with an additional 10% added subsequently until all the events are filtered. The same experiment is repeated for the standard OVS bridge. In the case of OVS bridge, the filtering is done in the evntbroker.
As seen in the graph plotted in Figure 5.7, both EVS and OVS perform very similarly when the percentage of event filters is increased. This experiment however does not showcase the benefits of event filtering in EVS. Because the measurements are taken only when the event is delivered to the evntsink. And for the event to be delivered to the evntsink, it must first be redirected by the EVS bridge using the expensive *mod* actions.

5.4.7 Performance measurement of event attributes detection and redirection

In this subsection the performance evaluation of the EVS bridge is presented when event attributes are detected and redirected. The results are contrasted to the measurements produced when similar attributes are detected and redirected using the evntbroker on the OVS bridge. To do the set up for the experiment, rules are installed into the EVS bridge to detect event attributes. When the events with the given attribute or pattern of attributes are detected, the events are redirected to the evntsink. All the other events are filtered out.

Detect operation on one attribute

First, the rule for detecting event type and a single event attribute is considered. This operation is notated by:

$$D(e.t \wedge e.a_1) \quad | \quad stream, \tag{5.2}$$

where D is the detect operation;
$e.t$ is the event type;
$e.a1$ is the first event attribute;
$|$ denotes the redirect operation;
and $stream$ is the logical stream to which the detected event is redirected to.

Whereas other streaming applications may have named stream support, EVS stream redirection is done based on network addresses of the application consuming the event stream. The controller rule installed in EVS corresponding to this operation is:

```
{
"dpid": 178974088016461,
"table_id": 0,
"priority": 11112,
"flags": 1,
"match":{
```

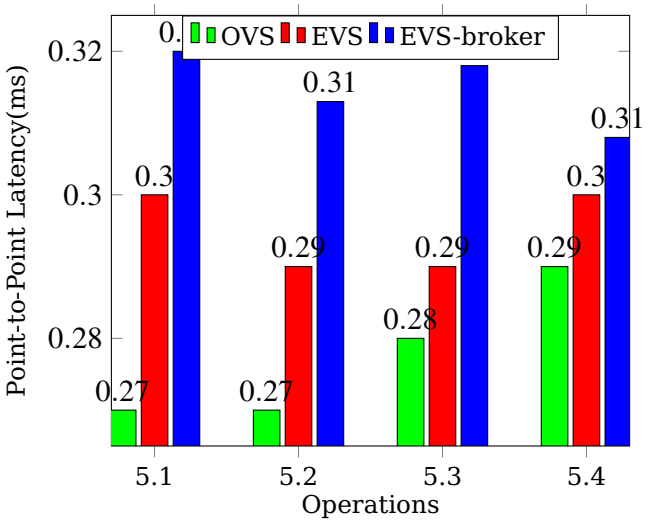
```
"dl_type":0x0800,
"nw_proto":17,
"nw_src":"10.1.1.2"
"nw_dst":"10.1.1.3",
"tp_dst":9877,
"e_type":"TEMP",
"e_attr1":80,
},
"actions":[{
"type":"set_nw_dst",
"nw_dst": 10.1.1.1
},
{
"type":"NORMAL"
}
]
}
http://localhost:8080/stats/flowentry/add
```

This is similar to the INSERT INTO clause provided by many Event Processing Languages, where the events from an event stream are filtered based on a value and forwarded into another event stream. In this case, the new event stream is the the stream of events handed over to the application listening on port 9877. Consider a stream of temperature events where each event has readings from multiple sensors. The operation at (5.2) is semantically similar to the below Event-Processing query:

```
INSERT INTO NEWTEMPSTREAM
SELECT * FROM TEMPSTREAM
WHERE DEVICE_1.VALUE = 80
```

As seen in Figure 5.8, the latency caused by performance of the operation notated in (5.2) in the EVS bridge is marginal higher than the latency observed on the standard OVS bridge by the order of 0.02 ms. When the same operation was evaluated on the EVS bridge operating in standard mode, the latency observed was marginally higher than the OVS bridge by a order 0.03 ms. This is effectively the latency added by per-packet event processing.

Figure 5.7: Performance of Event attribute match operations



Detect operation on two disjunct attributes

The next operation considered for evaluation is notated by:

$$D(e.t \wedge (e.a_1 \vee e.a_2)) \quad | \quad stream, \tag{5.3}$$

where D is the detect operation;
$e.t$ is the event type;
$e.a1$ is the first event attribute;
$e.a2$ is the second event attribute; | denotes the redirect operation;
and *stream* is the logical stream to which the detected event is redirected to.

The controller installs an additional rule - seen below - in addition to the one above, to perform this operation:

```
{
"dpid": 178974088016461,
"table_id": 0,
"priority": 11112,
"flags": 1,
"match":{
"dl_type":0x0800,
"nw_proto":17,
"nw_src":"10.1.1.2"
"nw_dst":"10.1.1.3",
"tp_dst":9877,
"e_type":"TEMP",
"e_attr2":80,
},
"actions":[{
```

```
"type":"set_nw_dst",
"nw_dst": 10.1.1.1
},
{
"type":"NORMAL"
}
]
}
http://localhost:8080/stats/flowentry/add
```

Drawing parallel to an Event Processing Language, the operation at (5.2) is similar to:

```
INSERT INTO NEWTEMPSTREAM
SELECT * FROM TEMPSTREAM
WHERE DEVICE_1.VALUE = 80 OR DEVICE_2.VALUE=80
```

As we seen in Figure 5.8, the execution of this operation follows the trends observed for operations 5.1 ad 5.2. No remarkable overhead was added on the EVS bridge, despite the addition of the rule. The reason for this is clearly explained in section 5.4.6.

Detect operation on two conjunct attributes

The next operation considered for evaluation is notated by:

$$D(e.t \wedge (e.a_1 \wedge e.a_2)) \quad | \quad stream, \tag{5.4}$$

where D is the detect operation;
$e.t$ is the event type;
$e.a1$ is the first event attribute;
$e.a2$ is the second event attribute; $|$ denotes the redirect operation;
and $stream$ is the logical stream to which the detected event is redirected to.

The controller installs one rule - seen below - to perform this operation:

```
{
"dpid": 178974088016461,
"table_id": 0,
"priority": 11112,
"flags": 1,
"match":{
"dl_type":0x0800,
"nw_proto":17,
"nw_src":"10.1.1.2"
"nw_dst":"10.1.1.3",,
"tp_dst":9877,
"e_type":"TEMP",
```

```
"e_attr1":80,
"e_attr2":80,
},
"actions":[{
"type":"set_nw_dst",
"nw_dst": 10.1.1.1
},
{
"type":"NORMAL"
}
]
}
http://localhost:8080/stats/flowentry/add
```

Drawing parallel to an Event Processing Language, the operation at (5.2) is similar to:

```
INSERT INTO NEWTEMPSTREAM
SELECT * FROM TEMPSTREAM
WHERE DEVICE_1.VALUE = 80 AND DEVICE_2.VALUE=80
```

As seen in Figure 5.8, and similar to the case of disjunction operation (5.3), the execution of the conjunction operation in the EVS bridge does not add any remarkable point-to-point latency. And as expected, the EVS-broker has the worst perfomance, because of the added processing and packet hops.

Network Traffic Monitoring for the event detection and redirection operations

For all the operations defined through 5.1 to 5.4, the traffic in the network was monitored both in the case of OVS and EVS bridge. The system was set up to send a flow of 100000 UDP packets from the eventsrc to the evntsink. The traffic characteristics observed on the six interfaces are shown in Table 5.2. It can be observed that the EVS bridge avoids using the interfaces related to namespace 3 completely for the flow. That is a 33% reduction in traffic for single staged event processing.

Table 5.2: Network traffic on Namespaces

Interface	OVS Bridge	EVS Bridge
Hypervisor - tap1	Tx-100000	Tx-100000
Namespace1 - tap1	Rx-100000	Rx-100000
Hypervisor - tap2	Rx-100000	Rx-100000
Namespace2 - tap2	Tx-100000	Tx-100000
Hypervisor - tap3	Rx-100000, Tx-100000	0
Namespace3 - tap3	Rx-100000, Tx-100000	0

5.4.8 Performance measurement of compare operations on event attributes

In this subsection, the performance of the comparison operations on the event attributes is presented. This class of operations perform a binary operation on the event attribute value. The evaluation of two

compare operators which have been implemented- *Greater than or equal to (>=), Less than or equal to (<=)* - is detailed.

Greater than or equal to operator

The operation supported by Greater than or equal to operator is notated by:

$$D(e.t \wedge (e.a_1 >= value) \quad | \quad filter, \tag{5.5}$$

where D is the detect operation;
$e.t$ is the event type;
$e.a1$ is the first event attribute;
$value$ is the integer value to be compared with;
$|$ denotes the redirect operation;
and $fliter$ is the denotation for filtering of the event.

```
{
  "dpid": 178974088016461,
  "table_id": 0,
  "priority": 11112,
  "flags": 1,
  "match":{
    "dl_type":0x0800,
    "nw_proto":17,
    "nw_dst":"10.1.1.2",
    "nw_dst":"10.1.1.1",
    "tp_dst":9877,
    "e_type":"TEMP",
  },
  "actions":[{
    "type":"set_max",
    "value": 100
  },
  {
    "type":"NORMAL"
  }
  ]
}
http://localhost:8080/stats/flowentry/add
```

Which in the perspective of event query languages can be represented as:

```
DROP FROM TEMPSTREAM
WHERE DEVICE_1.VALUE >= 100
```

The operation at (5.5) can be used with conjunction operation on event attribute2.

$$D(e.t \wedge ((e.a_1 >= value) \wedge e_2) \quad | \quad filter, \tag{5.6}$$

or with the disjunction operation on attribute2.

$$D(e.t \wedge ((e.a_1 >= value) \vee e_2) \quad | \quad filter, \tag{5.7}$$

The Less than or equal to operator is introduced next and the evaluation discussion of the two operators is presented together at the end of the subsection.

Less than or equal to operator

The operation supported by Greater than or equal to operator is notated by:

$$D(e.t \wedge (e.a_1 <= value) \quad | \quad filter, \tag{5.8}$$

where D is the detect operation;
$e.t$ is the event type;
$e.a1$ is the first event attribute;
$value$ is the integer value to be compared with;
$|$ denotes the redirect operation;
and $fliter$ is the denotation for filtering of the event.

```
{
"dpid": 178974088016461,
"table_id": 0,
"priority": 11112,
"flags": 1,
"match":{
"dl_type":0x0800,
"nw_proto":17,
"nw_dst":"10.1.1.2",
"nw_dst":"10.1.1.1",
"tp_dst":9877,
"e_type":"TEMP",
},
"actions":[{
"type":"set_min",
"value": 100
},
{
"type":"NORMAL"
}
]
}
http://localhost:8080/stats/flowentry/add
```

Which in the perspective of event query languages can be represented as:

```
DROP FROM TEMPSTREAM
WHERE DEVICE_1.VALUE <= 100
```

The operation at (5.8) can be used with conjunction operation on event attribute2.

$$D(e.t \wedge ((e.a_1 <= value) \wedge e_2) \quad | \quad filter, \tag{5.9}$$

or with the disjunction operation on attribute2.

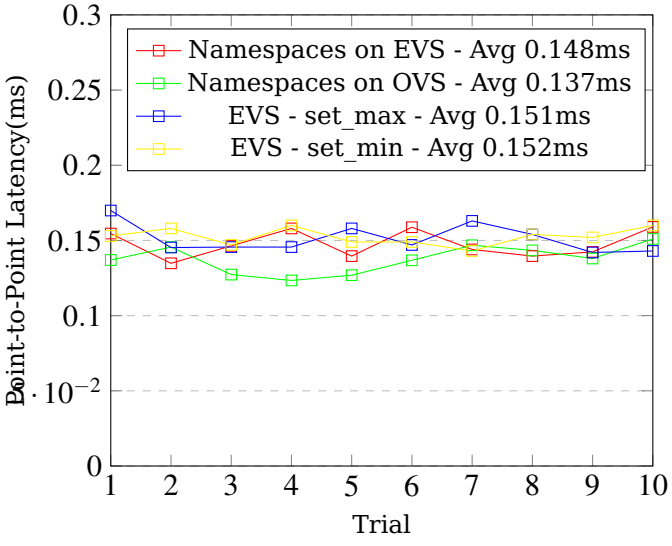
$$D(e.t \wedge ((e.a_1 <= value) \vee e_2) \quad | \quad filter, \tag{5.10}$$

The attr_cmp action defined in section 4.6.7 internal implements the <,>,= operators, the performance observed for the set_min, set_max operators can be generalized for the class of compare operations defined in section 4.6.7. Hence For evaluation purposes only the set_min and set_max operators defined in (5.5) and (5.8) are considered.

Performance of compare Operations

In this subsection, the performance of the compare operations defined in section 4.6.7 is presented. First, the evaluation of rules defined in (5.5) and (5.8) is presented when only positive values are sent to the system. This evaluation is done in order to understand the rule performance is general and to see if the rule in itself creates a bottleneck in performance. For example, for a set_min:100 rule, only events of type "TEMP" and with values greater than 100 are sent. In this case, each event is forwarded by the rule and none of the events are dropped. The same test was repeated for the set_max:100 operator but this time by passing values less than 100. This test is not intended to test the accuracy of the rules, but only to check if the rules with additional operations add an overhead. The performance is plotted in Figure 5.9 and is contrasted with the inter-namespace switching results provided in section 5.4.2, figure 5.4. As it can be seen, the rules for the two operations do not add a significant overhead and are comparable to the performance of the EVS bridge with inter-namespace switching based on event type attribute.

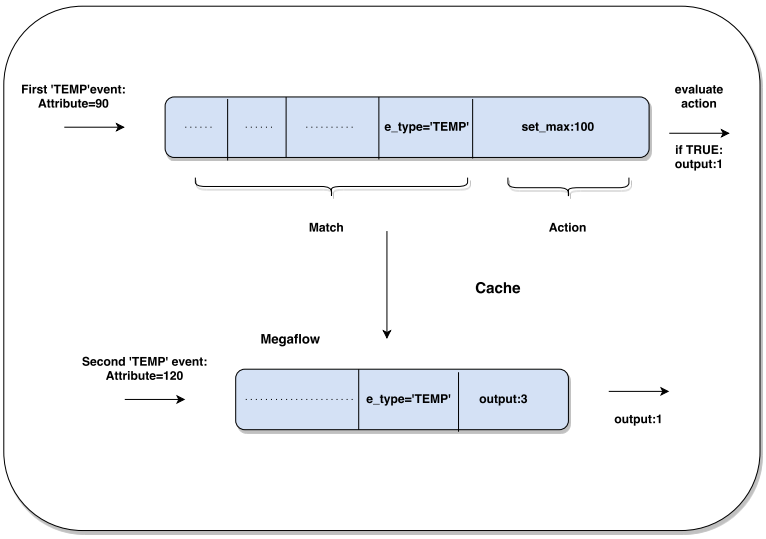
Figure 5.8: Comparator Performance



5.4.9 Evaluating for accuracy - compare operations

In the previous subsection, the performance of the comparision operations were measured only with positive test inputs where none of the events are dropped. But when random values are sent as attributes, the compare operations fail to perform the role they are designed for accurately. This is because of the megaflow cache design of the Open vSwitch. The cross product of all the active flows in the userspace OpenFlow tables are cached in the megaflow table. And as long as the incoming flow matches with an entry in the megaflow table, the learnt action is applied. This makes compare actions tricky to implement. These actions are conditional, i.e if the expression specified by the action evaluates to TRUE, the forwarding action is applied, or else the drop action (or a lack of action) is applied. So when a set_max:100 action is evaluated for the first event, the value for event attribute is extracted; for the ease of discussion let us consider 90 to be this value. Therefore, the set_max action evaluates to TRUE, after which the packet is sent to the learnt port based on the 'nw_dst' address. This flow is now cached in the megaflow table with the learnt output port.

Figure 5.9: Megaflow cache



The subsequent events are now looked-up in the megaflow cache, and the cached forwarding action is applied. Hence once the events start hitting the megaflow table, there is no opportunity for attribute evaluation. Open vswitch implements a re-validator thread which evicts the inactive flows or to update the recently changed flows. The compare operations from this perspective can be viewed as flows which need to be refreshed after every flow hit. By default the cache revalidation happens every 10 seconds. So if an event of type 'TEMP' hits the compare operator rule once in 10 seconds, the operation is evaluated accurately. However this is not ideal. Hence different ways to reduce this flow hit rate are explored. Open vSwitch provides options to configure the revalidation of the cache.

Disabled-Megaflow: The first approach is to disable-megaflow, using the below command.

```
$ ovs-appctl upcall/disable-megaflows
```

After having disabled megaflows, the evaluation to determine the accuracy and performance of the compare operations was performed. The rule installed for this evaluation was:

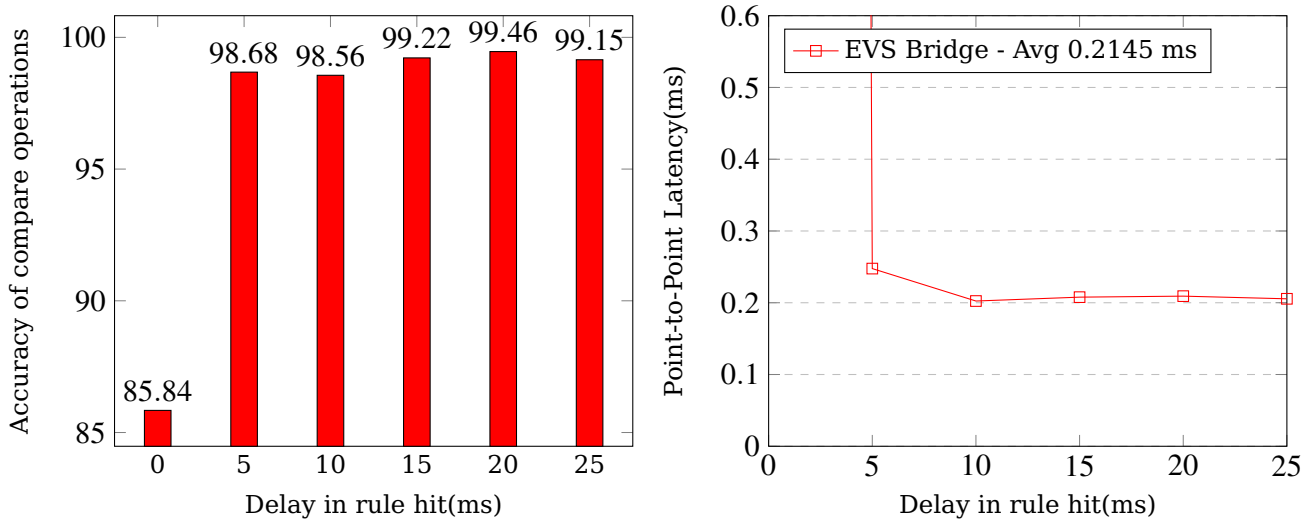
```
{
```

```
"dpid": 178974088016461,
"table_id": 0,
"priority": 11112,
"flags": 1,
"match":{
"dl_type":0x0800,
"nw_proto":17,
"nw_dst":"10.1.1.2",
"nw_dst":"10.1.1.1",
"tp_dst":9877,
"e_type":"TEMP",
},
"actions":[{
"type":"set_max",
"value": 5
},
{
"type":"NORMAL"
}
]
}
http://localhost:8080/stats/flowentry/add
```

The results are monitored such that only values greater than 5 are filtered by the EVS bridge, and values less than 5 are allowed through. To test this, the evntsrc is configured to generate events of type 'TEMP', such that every odd iteration generate a value greater than 5 and every even iteration generates a value less than 5. The evntsink counts any unexpected value greater than 5 received as a miss, and reports at the end of the test.

Another additional parameter that is set up is the frequency of the rule hit. As discussed before, by default Open vSwitch evicts idle megaflow rule every 10 seconds. With the megaflow disabled, this frequency of rule hit was configured by increasing the sleep time of the evntsrc application. This was done in order to measure the accuracy when the frequency is high. This also gives a good indicator of the frequency of rule hit that can be tolerated whilst having the highest accuracy. The results are plotted in Figure 5.11

Figure 5.10: Accuracy and latency measure of compare operations with disabled megaflows



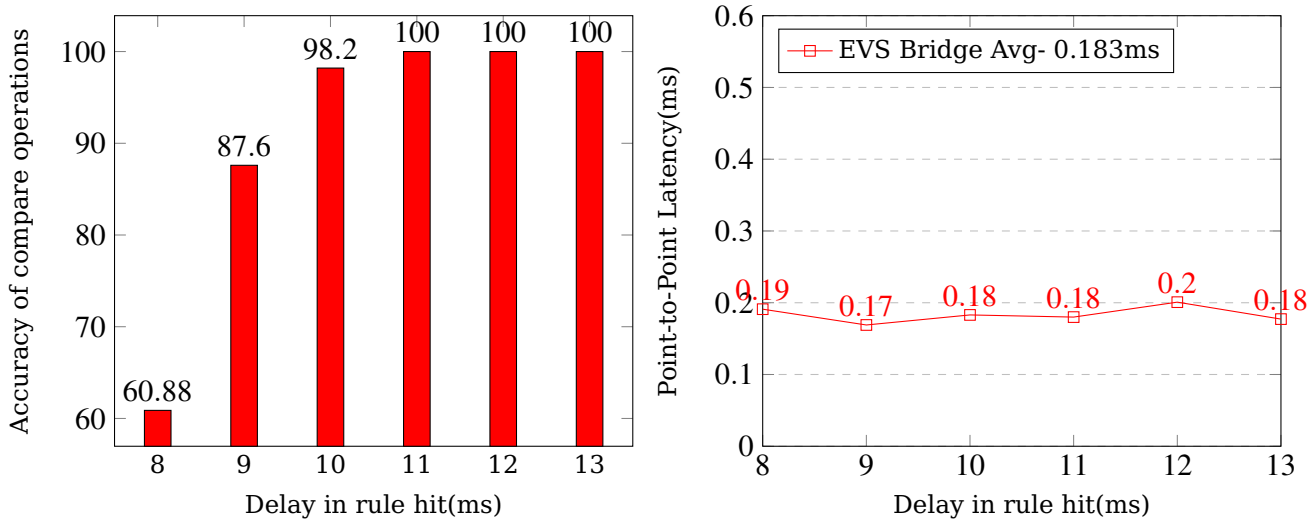
As it can be observed from Figure 5.11, when the compare rule is hit without any delay, the accuracy is at a mere 85.84% and the latency is measured to be 29.27 ms. But as delay is introduced to the rule hit parameter, the accuracy and latency normalizes to 99% with 0.21 ms latency. It is to be noted that even the 0.21 ms latency is quite high compared to 0.151 ms and 0.152 ms latency achieved (plotted in Figure 5.9) for set_min and set_max operators with enabled megaflow. This is because in the graph plotted in Figure 5.9, accuracy is not a considered parameter for evaluation and thereby with enabled megaflow caching, the lookup of the rules is much faster and the compare operation is not evaluated for each event.

Configure Max-idle time: The second approach is to configure the maximum idle time for each flow using:

```
1 $ ovs-vsctl --no-wait set Open_vSwitch . other_config:max-idle=1
```

Unlike disabling megaflows entirely, this configuration ensures that any flow rule which is idle for more than 1 ms is evicted. The results are plotted in Figure 5.12. As it can be seen, the flow rule accuracy reached 100% when the delay is 10 ms. This is because the re-validator needs a few milliseconds to evict the idle rules. In this set up the average point-to-point latency hovered around 0.183 ms, which is higher than the 0.151 ms reported in Figure 5.9, but lower than the 0.21 ms observed and plotted in Figure 5.11.

Figure 5.11: Accuracy and latency measure of compare operations with flow max idle time=1

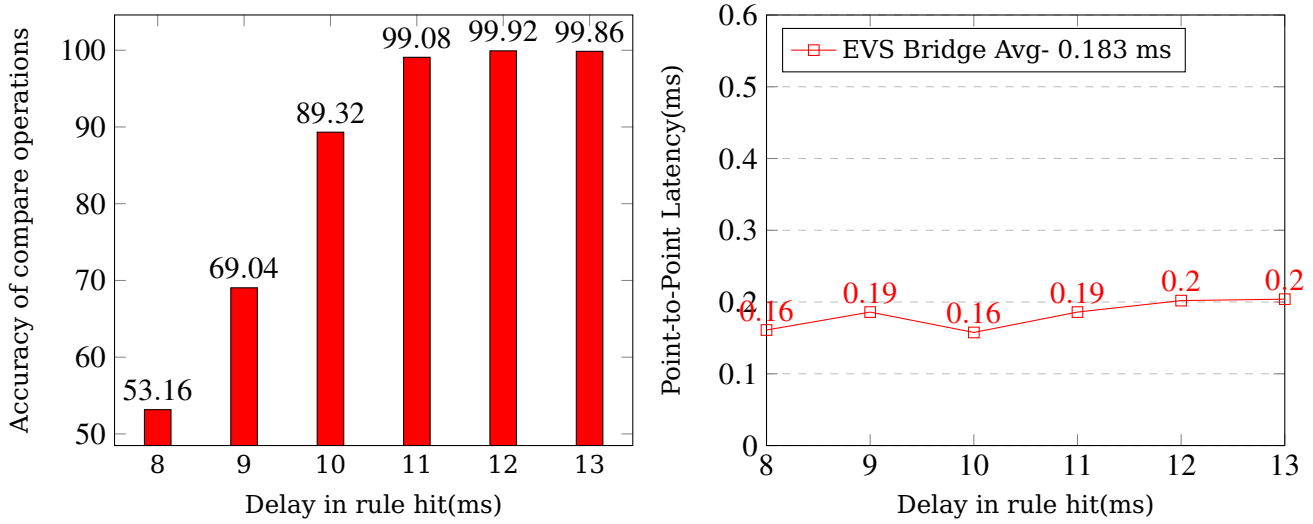


The results from the above experiment show that even though the max-idle time is set at 1 ms, the accuracy is at the peak if the flow hit frequency is around 10 ms. To monitor the change in accuracy and find the best rule hit delay with lowest point-to-point latency, the experiment was repeated with a max-idle time set as 5 ms.

```
1 $ ovs-vsctl --no-wait set Open_vSwitch . other_config:max-idle=5
```

The results shown in Figure 5.13 shows that when the idle flows are evicted every 5 ms, the ideal rule delay hit is observed at 12 ms where the accuracy is 99.96% with an average point-to-point latency of 0.2ms.

Figure 5.12: Accuracy and latency measure of compare operations with flow max idle time=5



Analysing the plots at Figure 5.13,5.12 and 5.11 with different maximum idle time and disabled megaflows, it is concluded that the best combination of accuracy, point-to-point latency and support for high frequency rule hit is achieved when the max-idle time for the flows is set at 1 ms. With this configuration a 100% accuracy with an average 0.18ms latency was observed for compare operations in the conducted trial runs when the flow rule was configured to be hit every 11ms.

5.5 Evaluation with DPDK

In this section the performance evaluation of the EVS bridge is presented with KVM guest virtual machines running the evntsrc ,evntbroker and evntsink applications. This section aims to emulate a real-world datacenter deployment with applications running in virtual machines. The EVS bridge in this section is accelerated by the Intel DPDK library, and the performance of this bridge is compared against a standard OVS bridge accelerated with DPDK. The evaluation in EVS DPDK is performed to measure the latency across two parameters:

- The impact of VM-context switch on latency.
- The impact of megaflow disabling in DPDK on the event actions.

5.5.1 System Set Up

In this subsection the system set-up for KVM guests bridged on EVS-DPDK via the dpdkvhostuser ports is presented. The configuration set up needed for such a system is described in detail. It is assumed that the system requirements for DPDK are met.
The DPDK source code is compiled with the relevant system architecture and environment.

```
export DPDK_DIR= /~/dpdk-stable-16.11.2
cd $DPDK_DIR
export DPDK_TARGET=x86_64-native-linuxapp-gcc
export DPDK_BUILD=$DPDK_DIR/$DPDK_TARGET
make install T=$DPDK_TARGET DESTDIR=install
```

The EVS source code is compiled with the DPDK target directory.

```
  cd $EVS_DIR
  ./configure --enable-coverage --enable-Werror --with-dpdk=$DPDK_BUILD && make && make install

```

The vswitchd daemon and the ovsdb-server are started in a standard manner. After which DPDK configurations are passed on to the ovsdb-server. The first command allocates memory on the two sockets of the system which is reserved for DPDK hugepages, whereas the second command gives the control of dpdkvhostuser sockets to KVM.

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="4096,4096"
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-extra=--vhost-owner \
libvirt-qemu:kvm --vhost-perm 0666
```

After which the EVS bridge is set up with three dpdkvhostuser ports.

```
ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
ovs-vsctl add-port br0 vhost-user1 -- set Interface vhost-user1 type=dpdkvhostuser
ovs-vsctl add-port br0 vhost-user2 -- set Interface vhost-user2 type=dpdkvhostuser
ovs-vsctl add-port br0 vhost-user3 -- set Interface vhost-user3 type=dpdkvhostuser
```

The DPDK hugepages allocations are made by running the dpdk-setup scripts. Minimum of 1G hugepage support is required by OVS.

```
cd $DPDK_DIR/tools
./dpdk-setup.sh
Option:20
```

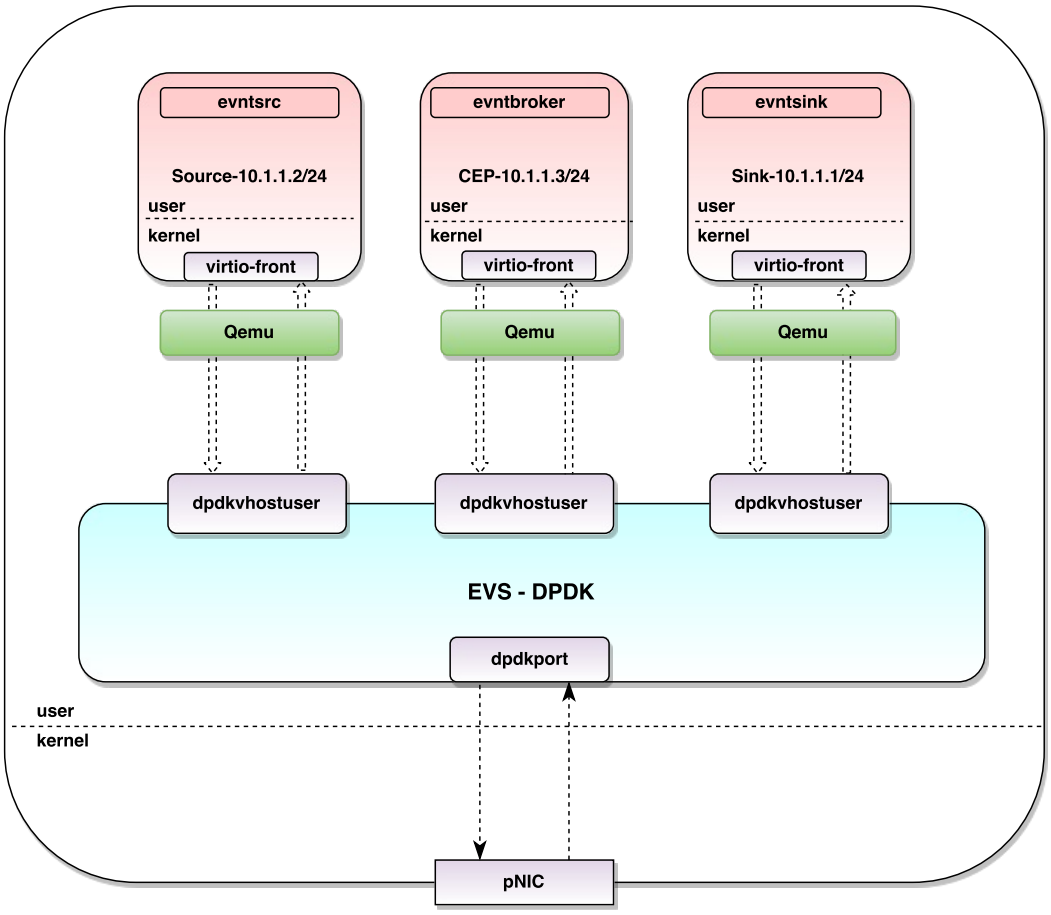
Next, three guest virtual machines are created. Here set up for one machine - bridge - is shown:

```
qemu-img create -f qcow2 /~/bridge.qcow2 20G
qemu-system-x86_64 -hda /home/advith/bridge.qcow2 -cdrom /home/advith/ubuntu-16.04.2-desktop-amd64.iso \
 -boot d -enable-kvm -m 4096
```

The three guests are brought up and attached to the dpdkvhostuser ports with appropriate memory mappings to ensure fast guest to host communication.

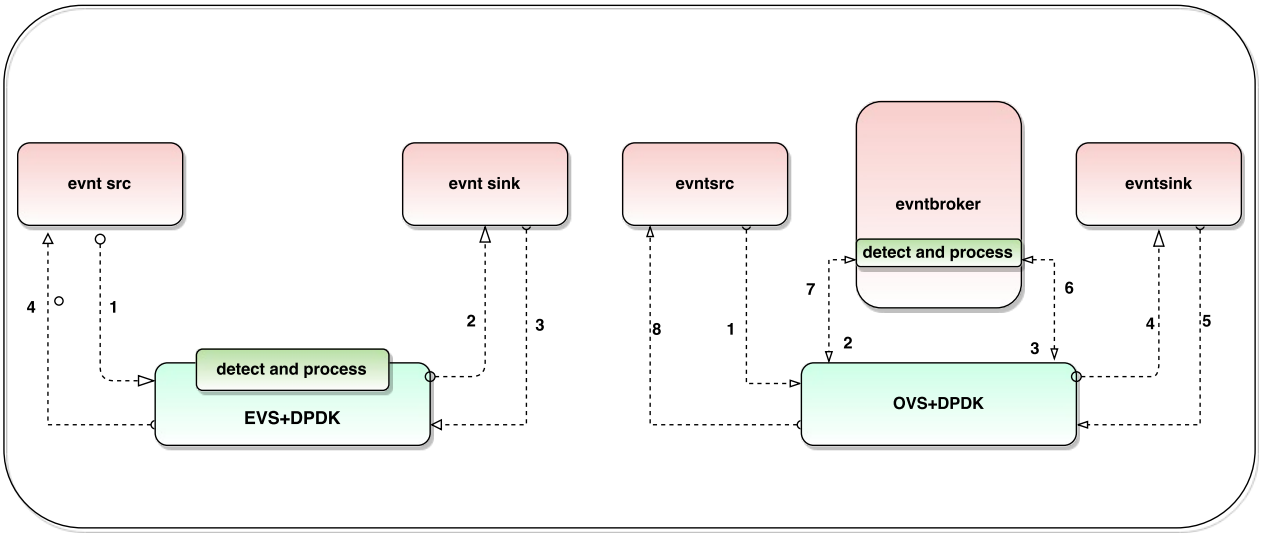
```
qemu-system-x86_64 -m 4096 -hda /~/bridge.qcow2 -boot c -enable-kvm -no-reboot -net none \
-chardev socket,id=char3,path=/usr/local/var/run/openvswitch/vhost-user3 \
-netdev type=vhost-user,id=mynet3,chardev=char3,vhostforce \
-device virtio-net-pci,mac=00:00:00:00:00:03,netdev=mynet3 \
-object memory-backend-file,id=mem,size=4096M,mem-path=/dev/hugepages,share=on -numa node,memdev=mem -mem-
    prealloc
```

Figure 5.13: KVM guests on EVS/OVS DPDK



The set up is pictorially represented in Figure 5.14. More about the DPDK-vhost implementation is discussed in section 2.3.1. Evaluating on guest virtual machines has it's own challenges caused by clock drift. Although all the three VM's set up use the kvm_clock as the current clock source, when the system is tested with a data flow set up illustrated in Figure 5.1, the latency results are observed to be erratic. Each time the guests are restarted - which is necessary when changing from OVS to EVS - the observed latency between evntsrc and evntsink is different, sometimes even negative. This indicates that the guests clocks are drifting despite the use of kvm_clock. Although, Ubuntu specifically recommends that NTP servers shouldn't be set up within guests, the guests were set up with NTP servers to check for synchronization. However the NTP servers in guests do not solve the issue of clock drift. Given this scenario, the Evaluation on Namespaces is done by measuring the round-trip latency of the events generated by the evntsrc application. The data flow set up for the evaluation is illustrated in Figure 5.15. In case of the EVS-DPDK set up, the evntsrc application in KVM guest-1 sends the events with a counter and a timestamp to the evntsink on KVM guest-2 via the EVS bridge. The evntsink application receives the event and retransmits with the same event to the evntsrc again via the EVS bridge. The evntsrc application on receiving the response, finds the delta between the timestamp in the event and the current timestamp to get the round-trip latency via the EVS bridge. This flow of data avoids the clock drift issue by ensuring that timestamps used to calculate the latency are generated within the same guest machine. In case of OVS, an evntbroker application on KVM guest-3 acts as the broker for events.

Figure 5.14: Data Flow in EVS vs OVS DPDK



The evaluation on KVM guests emulates a real-world data centre deployment. The motivation for this evaluation is to repeat the evaluation done on network namespaces, but rather find insights to the questions that are left open after evaluation on network namespaces, namely:

- The impact of the additional context-switch between the hypervisor and guests on latency.
- The impact of high throughput on the compare operations which in current form rely on cache re-validation.

5.5.2 Guest-to-Guest Measurements

Having established the key questions, the initial observations on guest-to-guest communication on the EVS bridge and OVS bridge in terms of throughput, RTT and average packet processing cycles per packet are detailed in the Table 5.3. The EVS bridge introduces a three-fold increase in packet processing cycles

per packet because of the additional processing needed to parse and de-serialize the events within the packet. But the burden is introduced only on tagged packets as is confirmed by the readings of per packet processing on iperf3 packets on both the bridges shown.

Table 5.3: OVS vs EVS DPDK comparision

Parameter	OVS Bridge	EVS Bridge
Throughput	6.40 Gbps	6.60 Gbps
hping3 RTT	4.0 ms	4.9 ms
Tagged packets - Avg processing cycles per packet	6075.91	20472.32
iperf3 - Avg processing cycles per packet	2828.63	2811.79

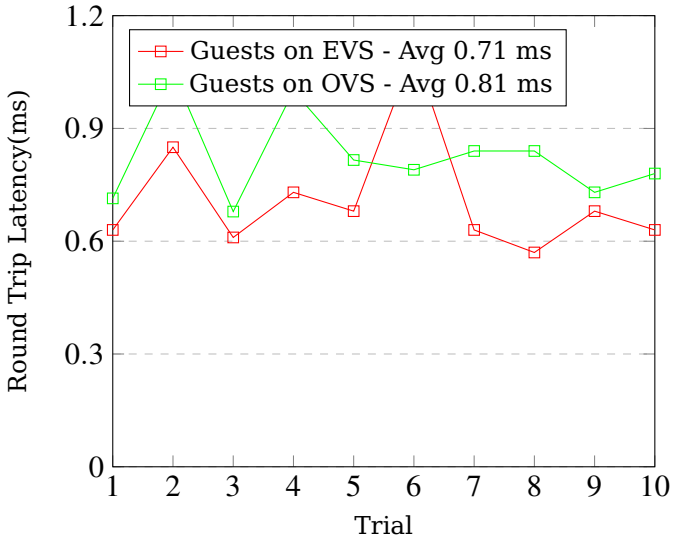
5.5.3 Performance measurement with event detection redirection

In this subsection, the performance evaluation of event detection and event redirection operation - 5.1 - is presented when performed on a hypervisor EVS-DPDK. This subsection is a counterpart of the evaluation performed on namespaces detailed in section 5.4.3. One key question left open by the evaluation on namespaces is whether the added context switch overhead that exists case of a hypervisor-guest deployment have an impact on the performance comparison of event detection and redirection operations performed on EVS against the same operations performed on a evntbroker in another guest. The flow of data for the evaluation is illustrated in Figure 5.15, and as described previously Round-trip latency is used instead of point-to-point latency. The event direction in case of a virtualized L2 network is done using the mod_dl_dst action unlike the mod_nw_dst action in namespaces. A similar rule is installed to handle the redirection from 10.1.1.1 to 10.1.1.3.

```
{
"dpid": 178974088016461,
"table_id": 0,
"priority": 11112,
"flags": 1,
"match":{
"dl_type":0x0800,
"nw_proto":17,
"nw_src":"10.1.1.2",
"nw_dst":"10.1.1.3",
"tp_dst":9877,
"e_type":"TEMP",
},
"actions":[{
"type":"set_nw_dst",
"nw_dst": 10.1.1.1
},
{
"type":"set_dl_dst",
"nw_dst": 00:00:00:00:00:01
```

```
},
{
"type":"NORMAL"
}
]
}
http://localhost:8080/stats/flowentry/add
```

Figure 5.15: Performance of event redirection in bridged virtual machines



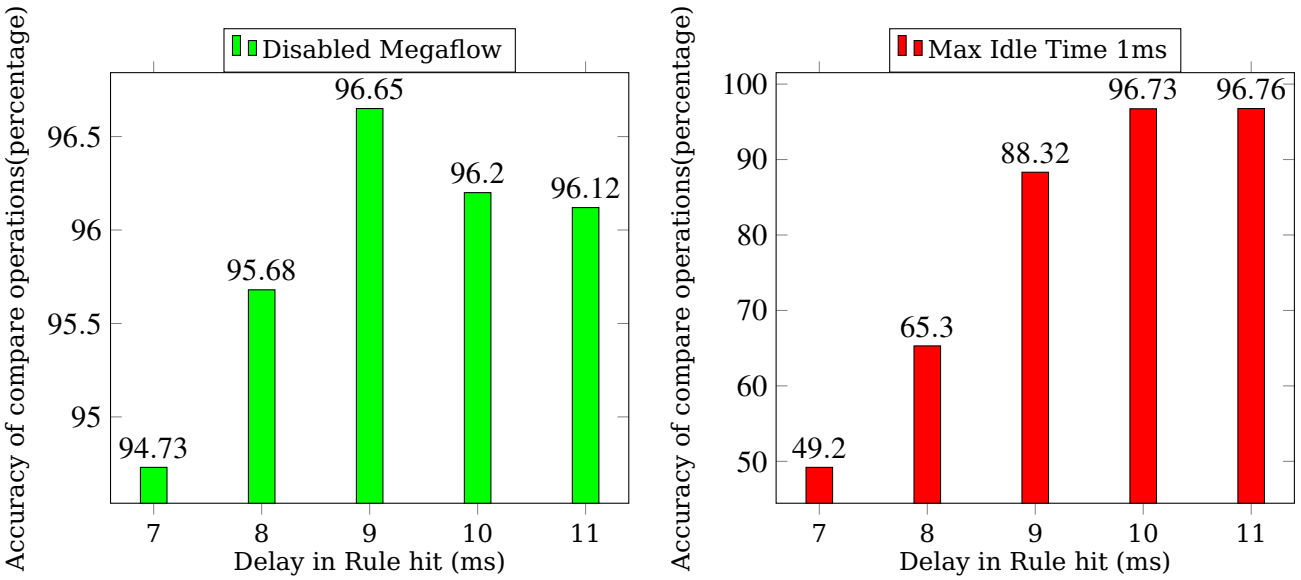
The results show that when event redirection is performed on EVS in a virtualization environment, removing the context switch from hypervisor to guest and back to hypervisor results in a lower round trip latency when compared to sending the events to the evntbroker for redirection. Although the improvement is marginal, this opens up avenues for exploration, especially to reduce the average processing cycle per packet.

5.5.4 Testing for accuracy of compare operations

In this subsection the accuracy of the compare operations - 5.7 and 5.8 - is evaluated in a similar fashion detailed in 5.4.9. The aim of the evaluation is to see the impact of high bandwidth on the ideal rule hit rate for compare operations. Two approaches are considered for the evaluation:

- Disabled Megaflows
- Max-idle time - 1 ms

Figure 5.16: Accuracy of operations in EVS DPDK



As plotted in the graph in figure 5.17, with disabled megaflows the EVS supports an accuracy of 94% to 97% when a flow rule is hit every 7ms, whereas with a configure idle time of 1 ms, an accuracy of 96% is acheived only at a rule hit rate of 10 ms. This can be contrasted with the results in figure 5.11 and 5.12 where higher levels of accuracy were hit. To understand this difference more work is done on the cache revalidation process in Open vSwitch. A future work may involve changing the re-validator to evict event processing rules at a higher rate compare to the other OpenFlow rules.

5.5.5 Evaluation of processing cycles needed for event operations

The event processing pipeline in EVS is expensive in general, as show in table 5.3. Each tagged packet in this pipeline averages three times the average processing cycle. In this subsection, the evaluation of processing cycles needed for event operations is presented.

Table 5.4: Processing cycles for event operations

Parameter	**Average processing cycles per packet**
Tagged packets without detection	20472.9
Tagged packets with event type detection	20039.28
Tagged packets with event type and one attribute detection	20322.76
Tagged packets with event type and two attributes detection	18623.9
Tagged packets with compare operations with disabled megaflow	83275.34

The processing cycles need per packet does not seem to vary as detection operations are performed on event attributes. However, when compare operations are performed with disbled megaflows, as they should be to reach an acceptable level of accuracy, the number of processing cycles needed nearly quadruple. This is because of the cache miss and additional Openflow pipeline look up per packet also observed indirectly as the increased point to point latency for these operations in figure 5.11 and 5.12.

6 Conclusion and Future Work

In this chapter the conclusions derived from the implementation and evaluation of the In-Network Event Processing framework are presented. The future work that can be undertaken to build up on the implementation and the conceivable improvements are additionally discussed.

6.1 Future Work

In the current implementation, event processing actions do not take advantage of the megaflow cache implementation of the Open vSwitch. Instead each event has to be looked up the OpenFlow processing pipeline and event actions applied for accurate results. To achieve the same, the megaflow cache eviction rate is increased which results in poor performance of the Open vSwitch bridge. This is not ideal because this affects all the systems bridged using Open vSwitch and not just the implemented event processing pipeline. To avoid this problem a sophisticated re-validator thread may be developed to evict only event based rules from the cache and allow other rules to remain cached.
The implemented event processing within Open vSwitch results in a significant increase in processing cycles per packet. This is because for each packet, the application layer is accessed, event attributes are extracted and de-serialized for further processing. This adds significant cycles per packet. A future work may address this drawback to make the event extraction process much leaner than what it is currently.
Furthermore, the current implementation focuses on the UDP transport protocol. A future work may extend the support to other protocols.

6.2 Conclusion

Network virtualization and Software defined networking offer boundless possibilities for provisioning chained network functions on demand with the aid of software-based solutions and programmable network control planes. As part of research conducted in the thesis, an exercise in programming the network control plane with application context and enabling the data plane to process application logic is presented within the context of a complex event processing ecosystem. To achieve the goals of the research the following contributions have been made:

- An event processing framework is implemented within the highly adopted Open vSwitch.
- The vSwitch is enabled to perform logical and stateful operations based on user logic configured as event rules.
- A framework to remotely offload event rules onto the network control plane via http is implemented using the RYU controller.
- A thorough evaluation of the implementation against several parameters is detailed and discussed.

The results of the evaluation shows that the benefits of detecting and redirecting events at the network and playing the role of a in-network event broker are compelling. Evaluation of this model shows a reduction in point-to-point latency between the producers and consumers of events and significant reduction in network traffic and processing for single staged processing systems by avoiding the utilization and context switch to a broker application. These results when extrapolated to multi-staged processing systems can potentially avoid multiple context switches and thereby improve the latency significantly and reduce burden on the network. However the results also show in an increased number of processing cycles per packet; which when viewed along with added benefits can be considered as a modest price to pay. When

higher level logical and stateful operations are performed on event attributes, the benefits are less apparent in the current implementation as a result of disabled caching. Although a reduction in network traffic, prevention of context switch to a broker and consequent avoidance of broker processing are observed, the point-to-point latency increases because of reliance on the OpenFlow processing pipeline instead of the cache. In addition to the impact on the event processing pipeline, this also adversely impacts the generic performance of the Open vSwitch. A possible solution to this problem is presented in section 6.1. Overall the thesis elaborates on the potential of offloading aspects of event processing onto the underlying network. Although stateful event operations are implemented, the benefits are not apparent because of the current caching limitations. Nonetheless, whilst performing the role of event broker the benefits become more apparent. This provides network operators with promising avenues to explore models of complementing existing complex event processing ecosystems with highly tuned application-aware custom network solutions.

Bibliography

[ABQ13] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218. ACM, 2013.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[BFÇ12] Nathan Backman, Rodrigo Fonseca, and UÇğur Çetintemel. Managing parallelism for stream processing in the cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, page 1. ACM, 2012.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[BRE+15] Alexander Beifuß, Daniel Raumer, Paul Emmerich, Torsten M Runge, Florian Wohlfart, Bernd E Wolfinger, and Georg Carle. A study of networking software induced latency. In *Networked Systems (NetSys), 2015 International Conference and Workshops on*, pages 1–8. IEEE, 2015.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, May 2000.

[CFP+07] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

[CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[CM13] Gianpaolo Cugola and Alessandro Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, February 2013.

[CQ09] Sharma Chakravarthy and Jiang Qingchun. *Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing*, volume 36. Springer Science & Business Media, 2009.

[CV14] Andreas Chatzistergiou and Stratis D Viglas. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1579–1588. ACM, 2014.

[CYY16] Laizhong Cui, F Richard Yu, and Qiao Yan. When big data meets software-defined networking: Sdn for big data and big data for sdn. *IEEE Network*, 30(1):58–65, 2016.

[DPD] Intel DPDK. Intel data plane development kit.

[EMIH15] Amr El-Mougy, Mohamed Ibnkahla, and Lobna Hegazy. Software-defined wireless network architectures for the internet-of-things. In *Local Computer Networks Conference Workshops (LCN Workshops), 2015 IEEE 40th*, pages 804–811. IEEE, 2015.

[FGHN15] Olivier Flauzac, Carlos González, Abdelhak Hachani, and Florent Nolot. Sdn based architecture for iot and improvement of the security. In *Advanced Information Networking and Applications Workshops (WAINA), 2015 IEEE 29th International Conference on*, pages 688–693. IEEE, 2015.

[Fou12] The Open Networking Foundation. Openflow switch specification, Jun 2012.

[GEW+15] Sebastian GallenmuÌĹller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet io. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 29–38. IEEE, 2015.

[HGJL15] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[HP15] Joel Halpern and Carlos Pignataro. Service function chaining (sfc) architecture. Technical report, 2015.

[HRW15] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. Netvm: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, 2015.

[HSB09] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 1:1–1:15, New York, NY, USA, 2009. ACM.

[JKM+13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[Kaf14] Apache Kafka. A high-throughput, distributed messaging system. *URL: kafka. apache. org as of*, 5(1), 2014.

[KG15] S. P. T. Krishnan and Jose L. Ugia Gonzalez. *Google Cloud Pub/Sub*, pages 277–292. Apress, Berkeley, CA, 2015.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[KN17] Jin-Hong Kim and Jung-Chan Na. A study on one-way communication using pf_ring zc. In *Advanced Communication Technology (ICACT), 2017 19th International Conference on*, pages 301–304. IEEE, 2017.

[LF98] David C Luckham and Brian Frasca. Complex event processing in distributed systems. *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford*, 28, 1998.

[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.

[MAR+14] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[MHM+14] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, and TV Lakshman. Application-aware data plane processing in sdn. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2014.

[NWL13] Fengfeng Ning, Chuliang Weng, and Yuan Luo. Virtualization i/o optimization based on shared memory. In *Big Data, 2013 IEEE International Conference on*, pages 70–77. IEEE, 2013.

[OFA] OFArchive. Of archive. `http://archive.openflow.org/`. 2016-02-08.

[PLS+06] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 49–49. IEEE, 2006.

[PPK+15] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *NSDI*, pages 117–130, 2015.

[QDG+14] Zhijing Qin, Grit Denker, Carlo Giannelli, Paolo Bellavista, and Nalini Venkatasubramanian. A software defined networking architecture for the internet-of-things. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9. IEEE, 2014.

[RDR10] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *ICCCN*, pages 1–6, 2010.

[Riz12] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[RL12] Luigi Rizzo and Giuseppe Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[Rus08] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[Sch14] Dominik Scholz. A look at intelâĂŹs dataplane development kit. *Network*, 115, 2014.

[SGY+09] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, pages 1–13, 2009.

[SOP+15] Radu Stoenescu, Vladimir Olteanu, Matei Popovici, Mohamed Ahmed, Joao Martins, Roberto Bifulco, Filipe Manco, Felipe Huici, Georgios Smaragdakis, Mark Handley, and Costin Raiciu. In-net: In-network processing for the masses. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 23:1–23:15, New York, NY, USA, 2015. ACM.

[sRe15] Esper Query Language sReference. Esper reference release 5.3.0, 2015.

[Sup14] OpenStack Superuser. Openstack user survey insights 2014, 2014.

[Sup17] OpenStack Superuser. Openstack user survey insights 2017, 2017.

[TLGD15] Nguyen B Truong, Gyu Myoung Lee, and Yacine Ghamri-Doudane. Software defined networking-based vehicular adhoc network with fog computing. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 1202–1207. IEEE, 2015.

[XMR16] Yiming Xu, V Mahendran, and Sridhar Radhakrishnan. Towards sdn-based fog computing: Mqtt broker virtualization for effective and reliable delivery. In *Communication Systems and Networks (COMSNETS), 2016 8th International Conference on*, pages 1–6. IEEE, 2016.

[ZOTW06] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 54–71. Springer, 2006.

[ZWRH14] Wei Zhang, Timothy Wood, KK Ramakrishnan, and Jinho Hwang. Smartswitch: Blurring the line between network infrastructure & cloud applications. In *HotCloud*, 2014.