Abstract—In age of Internet of Things(IoT), almost all devices are connected via physical and virtual networks. Virtual switch serves as packet forwarding device in communication between two or more IoT devices. As key part of Software Defined Network(SDN), Open vSwitch (OvS) is an open source virtual switch which is responsible for packet forwarding among network devices according to flow rules given by SDN controller. Performing efficient in-network processing among large scale heterogeneous devices with resource constrains requires light-weight design of processing unit. Due to various additional features and memory consumed by them, not all IoT devices are equipped to handle processing of OvS. This paper describes a light weight version of OvS with all basic functionalities intact and few insignificant functionalities removed to make it compatible with all IoT devices.

I. INTRODUCTION

In a heterogeneous IoT system, we have several IoT devices with different processing capabilities who interact with each other to perform different tasks. Keeping the processing capabilities in mind, we propose a light weight version of OvS virtual switch to perform in-network processing and route packets between different host applications. Host applications can be sensors, actuators or functions running on network nodes. Packets can contain requests to invoke functions, event triggers, notifications or can contain data bits being passed between different host applications.

In upcoming sections, we would be explaining concept of Software Defined Architecture(SDN) and Openflow protocol which forms the framework on which Open vSwitch operates. We would also explain design, core features of Open vSwitch and insignificant functionalities considered to be removed to make it light-weight. In the end, we would be discussing impact of light-weight Open vSwitch on the system.

II. FRAMEWORK

A. Software Defined Network

In traditional network, both control logic and packet forwarding plane resides in network device. Software Defined Network(SDN)[1] is an emerging paradigm that separates networks control logic from routers and switches and allows logical centralization of network control. It separates networks control plane, which decides how to handle network traffic and data plane which forwards traffic according to decisions made by control plane. This separation is realized by well-defined programming interface between switches and SDN controller [1]. It introduces the ability to program the network, create new abstractions in networking, simplify network management and facilitate network evolution.

Fig.1 shows high level view of SDN architecture. SDN[1] is divided into three fundamental components forwarding devices, controller and network applications. Forwarding devices perform actions on data packets as instructed by SDN controller. SDN controller is responsible for installing control commands on forwarding devices and collect status information about packet flows. It also offers global network view to network applications while hides details of underlying hardware in forwarding devices. Network applications enforce desired behavior on network via virtual solutions and network programming languages. This maps the configuration desired by applications into physical configuration of global network view highlighted by SDN controller.

SDN[1] helps in offering simple mechanism to modify network policies via high-level languages and
software components compared to low-level device specific configurations. Controller can automatically react to sudden changes of network state without unnecessarily alerting network application and maintain high-level policies. Also, centralized controller simplifies development of networking functions, services and applications.

B. Openflow

Openflow[2] is a protocol used by SDN[1] controller to communicate with Open vSwitch. It provides ability to program flow tables in switches and routers. Network can be partitioned into two parts research and production and researchers can control their own flow by choosing routes their packets follow, implement new protocols and not disturb production traffic. The datapath of switch operating on Openflow[2] protocol consists of flow table and an action associated with every flow entry. Flow tables are sent from controller to switches via a secure channel. Also, controller can send Openflow[2] messages like add, drop, update to switch to modify its behavior. There can be multiple flow rules for one packet, but the flow with highest priority always takes precedence. Flow rules are created by matching packet type, source and destination MAC address, source and destination IP address, input and output port numbers for a packet or combination of any of the attributes mentioned above.

III. OPEN vSWITCH

Open vSwitch(OvS)[3] is a virtual switch usually used in a forwarding device in a Software Defined Network(SDN)[1]. It uses Openflow[2] protocol for communication with controller, contains flow caching which prevents it from using more hypervisor resources than required, is open-source and can be implemented on multiple platforms. Open vSwitchs environment is usually selected by user who chooses an operating system and hypervisor.

![Fig. 2. Components and Interfaces of Open vSwitch][3]

In Open vSwitch[3], two components contribute to packet forwarding. First is open-vswitchd, which is a userspace daemon coded in exactly same manner for different operating systems. Second is datapath kernel module, which is written in accordance with host operating system to enhance its performance. The datapath kernel module receives the packet first. If ovs-vswitchd has already instructed datapath kernel module about handling these type of packets, then appropriate action like packet modification, packet drop or packet forwarding to selected port is taken on the packet. Else, packet is delivered to ovs-vswitchd. The userspace daemon decides how to handle the packet, passes the flow back to datapath kernel module and gives instruction to store the flow in cache for handling packets of similar type. The datapath kernel module has 2 layers of caching microflow cache which caches flows per transport connection and megaflow cache which caches flows for traffic heaps beyond individual connections. The main protocol used to pass flow tables is Openflow[2]. It allows controller to add, remove, update and obtain statistics on flows from Open vSwitch. The ovs-vswitchd module receives Openflow tables from SDN controller, matches any packets received from datapath kernel module against these tables, fetches actions and caches the result in kernel datapath. In open-vswitchd, each packet looks up series of Openflow[2] tables and finds the highest priority flow whose conditions are satisfied by packet, and executes its Openflow actions.

Ovsdb-server[4] component stores configuration database of Open vSwitch. Controller connects to ovsdb-server[4] over OVSDB protocol to access configuration database which allows it to create and drop Openflow switches, add or remove ports, configure Quality of Service queues, associate controller and switch, enable and disable Spanning Tree Protocol. Ovsdb-server provides RPC interfaces to one or more Open vSwitch databases. It supports JSON-RPC client connections over TCP/IP sockets. By default, ovsdb-server[4] runs as an active server but, there exists facility to make ovsdb-server run as a backup server. When running as backup server, all transactions that can modify database content are rejected. When ovsdb-server role changes, all client connections are reset and clients are required to reconnect to server.

Below are set of commands provided by OvS:

**ovs-vsctl[5]**: It connects ovsdb-server process which maintains OvS configuration database. Using this, it queries and applies changes to databases depending upon commands. It allows addition and deletion of bridge, port, interface, adding tunneling between interfaces and so on.

**ovs-ofctl[6]**: Its used as a command line tool to monitor and configure Open-vSwitch. It can show current state of Open vSwitch like features, configuration and flow table entries. It supports TCP/IP, Secure Socket Layer and Unix domain server sockets connections to connect to switch.

**ovs-dpctl[7]**: It is used to create, update, delete Open vSwitch datapaths hosted on one or more machines. It works only with datapaths implemented outside ovs-
vswitchd module, like Windows and Linux kernel-based datapaths.

**ovs-appctl[8]**: Its used to query Open vSwitch daemons.

**ovs-vswitchd[9]**: It is an Open vSwitch daemon which manages and controls any number of switches on local machine. It retrieves its configuration from database at startup, sets up datapaths and operates across each bridge as per its configuration files. A single instance of ovs-vswitchd runs at a time and manages as many number of instances as supported by Open vSwitch datapaths.

#### IV. INSIGNIFICANT FUNCTIONS

Below is list of insignificant functions removed from Open vSwitch to make it light-weight. Precise care was taken to ensure that removal of these functions would not disrupt functioning of main functionalities of Open vSwitch described above. Before listing these functionalities, we would like to clarify that criteria for deeming a function insignificant can depend on user to user and project requirements. Our version of Open vSwitch was created to integrate with Datamill project spearheaded by Dr. Peizhao Hu of Rochester Institute of Technology.

**A. Port Mirroring**

Port Mirroring[10] allows packets sent or received on one or more ports to be duplicated on different port. It is mainly used for debugging and checking packet contents. It involves creating dummy interface that will receive packets and adding that dummy interface to bridge in use. Once interface is connected to bridge, we configure it to output mirror to target interface or port. Port Mirroring is part of ovs-vsctl command.

```bash
$ ovs-vsctl add-br br0
$ ovs-vsctl add-port br0 eth0
$ ovs-vsctl add-port br0 mir1
$ ovs-vsctl -- -- --id=@p get port mir1
$ ovs-vsctl -- -- --id=@m create mirror name=m0 select-all=true output-port=@p
$ ovs-vsctl -- -- set bridge br0 mirrors=@m
```

Commands mentioned above create bridge br0 and configure bridge br0 with port eth0. All traffic coming in or going out of eth0 is also mirrored to port mir1. We have set up a check on third command which creates mirror. Subsequently, checks have also been set up on commands which check traffic at mirrors and drop mirrors.

**B. Interface Rate Limiting**

Interface Rate Limiting[11] allows us to control rate of packets received and dispersed from an interface. It helps in determining Quality of Service for a packet. It utilizes Kernels traffic control frameworks ingress policing. Ingress policing rate is maximum rate(kbps) at which interface can send packets beyond policing rate. Interface rate limiting is part of ovs-vsctl command.

```bash
$ ovs-vsctl set interface tap0 ingress_policing_rate=1000
$ ovs-vsctl set interface tap0 ingress_policing_burst=100
$ ovs-vsctl list interface tap0
```

First two commands listed above enforce ingress policing rate and ingress policing burst on interface tap0 while third command lists limits applied to interface tap0. We have set up check on first two commands of ovs-vsctl to avoid user set ingress policing rate and ingress policing burst.

**C. VXLAN Tunnel Endpoint**

The main intention behind VXLAN Tunnel Endpoint (VTEP)[12] is to create overlay network connecting Virtual Networks and Physical Servers and make them function as if they are in same L2 network. In Open vSwitch, we use SDN controller to manipulate OVSDB database on VTEP depending on its type. If VTEP is hypervisor, then controller will make changes in Open vSwitch database, else it will make changes in vtep hardware database.

In server 1,

```bash
$ ovs-vsctl add-port s1 vtep -- -- set interface vtep_type=vxlan option:remote_ip=192.168.2.20 option:key=flow_ofport_request=10
```

In server 2,

```bash
$ ovsdb-tool create /etc/openvswitch/vtep.db /usr/share/openvswitch/vtep.ovsschema
$ ovs-appctl -ovsdb-server ovsdb-server/add-db /etc/openvswitch/vtep.db
$ vtep-ctl add-ps s2
$ vtep-ctl set Physical_Switch s2 tunnel_ips=192.168.2.20
$ vtep-ctl list Physical_Switch
```

Above commands create VTEP connection between server 1 and server 2. The first command creates vxlan port on s1 and points it to hardware vtep at 192.168.2.20. Second and third commands load hardware vtep database in server 2. Next 2 commands register switch s2 to hardware vtep database and indicate switch tunnel IP where VXLAN tunnels will be terminated. We have set up necessary checks for commands ovs-vsctl, ovs-appctl, ovs-ofctl, ovsdb-tool and vtep-ctl which are required for operations of VTEP in Open vSwitch.

**D. PCAP and TCPDUMP**

Ovs-pcap reads pcap file name at command line and prints each packets contents as sequence of hex digits on a line of its own.
$ ovs-pcap filename

The above command describes the syntax for calling pcap method in Open vSwitch. It also has features of printing version information to console and printing help message to console about how to operate pcap command.

Ovs-tcpundump[13] command creates temporary mirror port in userspace daemon and executes linux tcpdump command to listen against those ports. When command instance is destroyed, the temporary mirror ports created are destroyed as well. It doesnt allow multiple mirror ports to listen to same port.

$ ovs-tcpundump –i p1 --mirror-to mp

The above command describes the syntax to call Open vSwitch Tcpdump command. It creates temporary port mp and mirrors packet from port p1 to mp. Checks have put up in python files of pcap and tcpdump to ensure these commands are never executed.

E. Statistics

Open vSwitch maintains per CPU packet processing statistics for a datapath, flow statistics and port statistics stored during its operations. These statistics are regularly sent to controller. Per CPU packet processing statistics contains number of received packets for which matching flow was found and not found in the flow table. The total of these two is total number of packets received by datapath. It also contains total number of packets whose flow was not found and which could not have sent to userspace daemon, usually due to overflow in datapath queues. Flow and port statistics are obtained from ovs-dpctl command

$ ovs-dpctl –s show

Output:
lookups: hit:188 missed:159 lost:0
flows: 4 masks: hit:729 total:3 hit/pkt:2.10
port 0: ovs-system (internal)
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 aborted:0 carrier:0
collisions:0 RX bytes:0 TX bytes:0
port 1: s1-eth2
RX packets:12 errors:0 dropped:0 overruns:0 frame:0
TX packets:17 errors:0 dropped:0 aborted:0 carrier:0
collisions:0 RX bytes:928 TX bytes:1323
port 2: s1-eth1
RX packets:12 errors:0 dropped:0 overruns:0 frame:0
TX packets:19 errors:0 dropped:0 aborted:0

carrier:0
 collisions:0 RX bytes:928 TX bytes:1491
port 3: s1 (internal)
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:13 abortion:0
carrier:0
 collisions:0 RX bytes:0 TX bytes:0

The above command prints summary of configured datapaths which includes datapath numbers and list of ports connected to each datapath along with packet and byte counters for each port. Datapath numbers contain flow statistics and mega flow mask statistics. The lookups row displays statistics related to flow lookups caused by processing incoming packets in datapath. Hits displays number of packets matching existing flows while missed displays number of packets requiring userspace processing. Lost displays number of packets lost before reaching userspace daemon. Flows row displays number of flows in datapath. If one or more datapaths are specified, then information on only those datapaths is specified. Else, information about all configured datapaths is specified. We have put up checks to prevent statistics update and statistics display to user.

V. RESULTS

We created 2 versions of Open vSwitch for our analysis Medium and Tiny version. We considered original Open vSwitch as Large version with all functionalities intact. The difference between medium and tiny version was that medium version had per CPU packet processing statistics, flow and port statistics intact. This version was created in case we need to see packet processing statistics in our network. We tested all 3 versions of Open vSwitch by creating a virtual network using mininet software on Ubuntu operating system with 64-bit processor. We considered minimum and maximum memory required, % CPU used and total size of software after removing insignificant functionalities as parameters for our analysis.

Fig. 3. Comparison of three OvS versions in terms of memory used
Fig. 3 shows the amount of memory consumed by Open vSwitch Large, Medium, and Tiny versions. To get these numbers, we ran all functions present in Open vSwitch and kept track of the change in memory by using the `top` command in a Linux machine. For testing VTEP functionality, two instances of virtual machines were created, and the memory consumed by the instance containing the VTEP database was recorded. On average, OvS Large version consumed around 2.1MB of memory while OvS Medium and Tiny versions consumed 1.9MB of memory. Thus, we were successful in reducing the maximum memory required by Open vSwitch by 9%.

Fig. 4 shows the %CPU used by Open vSwitch on a Linux machine. On an average, Large version of OvS uses minimum 0.9% CPU while Medium and Tiny versions use minimum 0.7% and 0.6% respectively. Also, Large version of OvS requires maximum 2.3% CPU while both Medium and Tiny versions require maximum 1.3% CPU. Thus, we were able to use the %CPU used by Open vSwitch in a Linux machine with 64-bit processor from 1.6% to 1%. Also, the original size of Open vSwitch was 33.5MB. After removal of code corresponding to insignificant functionalities, size of Open vSwitch became 33.3MB. Thus, we were able to reduce the size of Open vSwitch by 0.9%.

VI. CHALLENGES

Although detailed documentation about design and working of main functionalities of OvS is readily available, documentation of insignificant functionalities is hard to find and time consuming. Also, as the source code of OvS which is written in C language is not well documented, debugging the code to find functions and their dependencies takes time. The code for storing statistics was embedded within code of main functionalities. Care had to be taken to affect running of those functionalities and pass empty return statement from functionalities which were removed whenever required.

VII. CONCLUSION

Although we tested this software only on Mininet, we have made sure not to tamper with code required for initial setup of the switch on any operating system. Hence, the light-weight switch can be deployed on any Software Defined Network with forwarding devices running on any operating systems. Using light-weight Open vSwitch can help in reducing runtime for in-network processing in a Software Defined Network by making more memory available in a processor.

VIII. ACKNOWLEDGEMENTS

I sincerely thank Professor Peizhao Hu for providing this opportunity to research and learn about SDN, Open-vSwitch and IoT systems to develop a light-weight Open vSwitch which has immense scope for using in network with limited processing power and resource constraints.

REFERENCES