Building a NIC with Netcope P4

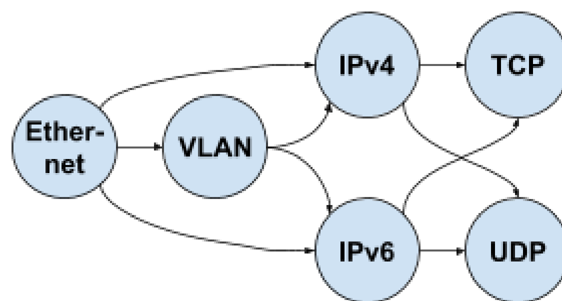
While P4 was primarily meant for network switches, it is very relevant for other related parts of network infrastructure. The benefits of having single means to describe as much of the network as reasonably possible are obvious. Network Interface Cards (NICs) are a nice example of new P4 use cases. From a certain point of view, isn't the NIC just one port of the switch, this time a virtual one?

Let's have a look at how a smart NIC can be promoted to full-featured NIC using pure P4 language. It will be shown how easy it is to implement features like MAC address filtering, packet and byte counting or load balancing using RSS (receive-side scaling) with support of encapsulation protocols. You can download and review the source codes to investigate the example further and understand why P4 language will drive the networks of near future.

We chose several interesting features that are typical for a NIC in order to demonstrate versatility of the P4 language. First, we will have a look at *parser*, a piece of the pipeline that has very interesting role as it defines supported protocol stack.

Parsing

The first strong point of the P4 language is located at the very beginning of the packet processing pipeline. The parser defines which protocols are recognized and can be used in the rest of the pipeline. In this particular case, it extracts destination MAC address for filtering on the input, and IP addresses and TCP/UDP port numbers that are used for RSS. So far, nothing special. But the parser can be configured to support various encapsulation protocols like MPLS, 802.1Q aka VLAN, PPPoE, VXLAN, GRE, etc. and even custom protocols like in the case of in-band network telemetry (INT). In our case, it is demonstrated on the VLAN protocol whose header can be present between Ethernet and IPv4/IPv6 headers. If the VLAN protocol was not supported and a VLAN-tagged packet would come in, it wouldn't be possible to parse the encapsulated IPv4/IPv6 packet and further processing that depends on corresponding header fields like RSS would fail. With P4, if a new protocol is to be used, just rewrite your parser and you are ready to go without modification of the rest of the pipeline. Supported protocol stack of our NIC looks like this:



For illustration of how user-friendly the P4 language is, a snippet of the code that defines a VLAN header and its processing follows:

```
header_type vlan_tag_t {
  fields {
    pcp : 3;
    cfi : 1;
    vid : 12;
    etherType : 16;
```

```
header vlan_tag_t vlan_tag;
parser parse_vlan {
  extract(vlan_tag);
  return select(latest.etherType) {
    ETHERTYPE_IPV4 : parse_ipv4;
    ETHERTYPE_IPV6 : parse_ipv6;
```

``
  }
}
``
	``
	    default : ingress;
	  }
	}
	``

Filtering

Filtering is applied for incoming packets to be checked against a valid MAC address. If the destination MAC address of the packet does not match, the packet is dropped. Simply said the NIC does not work in promiscuous mode. For the sake of clarity, we do not consider multicast or broadcast addresses, but it is a matter of minutes to add support of these. Do you feel like trying this?

In P4, filtering and applying actions is handled by match & action (M&A) tables. If a packet matches a condition of an item, corresponding action is applied. If not, a default action is applied. In our case, we define a M&A table with a single item for the valid MAC address. There are two actions defined for this table. For valid packets, RSS computation is performed. The others are dropped. Let's have a look at the source code snippet to reveal P4's simplicity. On the left side, there is the P4 source code of the single-item M&A table with exact match on Ethernet's destination MAC address field and the two available actions. On the right side, there's a portion of the text configuration file that can be used to populate M&A at run time.

```
table table_mac_filter {
  reads {
    ethernet.dstAddr : exact;
  }
  actions {
    compute_rss;
    _drop;
  }
  max_size: 1;
}
```

```
# drop packets by default
table_mac_filter default action drop

# perform RSS on valid packes
table_mac_filter keys (
  ethernet.dstAddr 0,96,8,159,177,243 )
  action compute_rss
```

Counting

In P4 language, there are several means of storing stateful data. These features can be used in applications that process network traffic on a flow-basis. They include packet and byte counters, registers and meters. In our case, we will use an array of several registers. They will serve to count all packets, packets that passed MAC address check, octets of such packets, and dropped packets. It would be certainly possible to extend the statistics with another metrics (e.g. from etherStatsTable of RMON1 MIB). Do you accept the challenge?

In the snippet below, there is a definition of the register array on the left side and the case of increasing the value of the byte counter of received packets on the right side. If you go through the P4 source code, you can come across two references about metadata. This concept of the P4 language is used to perform more complex operations that use data from previous levels of the processing pipeline. In our case, packet metadata contains values read from the register array that are then updated with new values. Intrinsic metadata carry auxiliary information about the packets like its length that is used in our case to increase the value of the counter. The lower part of the table shows how the values of the register array are read from SW.

```
#define COUNTERS_RCVD_BYTES  2
register counters {
       width : 32;
       static : table_mac_filter;
       instance_count : 4;
}
```

```
register_read(packet_metadata.rcvd_bytes,
  counters, COUNTERS_RCVD_BYTES);
add_to_field(packet_metadata.rcvd_bytes,
  intrinsic_metadata.packet_len);
register_write(counters, COUNTERS_RCVD_BYTES,
  packet_metadata.rcvd_bytes);
```

```
$ np4tool core --read --register=counters
    Netcope P4 core
        Core 0:
            Register counters : 1021,202,191000,819
```

Load Balancing with RSS

And finally, probably the most interesting feature. Receive-side scaling is, simply said, load balancing feature across software receive queues (which often map to CPU cores) that can be adjusted by a software application. It has two major parts. The first one is checksum computation whose result is used to address an indirection table. And how can be this behavior described in the P4 language? Since a checksum computation is common task performed upon network data (consider IPv4 and TCP/UDP), P4 features a construct called *field list computation*. It allows to specify the list of the header fields and the checksum function for the computation of the result.

The snippets below shows the definition of the list of the header fields on the left, the definition of the hash function on the right and usage of the field list computation with the action.

```
field_list ipv46_tcp_udp_fields {
  ipv4.protocol;
  ipv4.srcAddr;
  ipv4.dstAddr;
  ipv6.nextHeader;
  ipv6.srcAddr;
  ipv6.dstAddr;
  tcp_udp.srcPort;
  tcp_udp.dstPort;
}
```

```
field_list_calculation
ipv46_tcp_udp_csum {
  input
  {
    ipv46_tcp_udp_fields;
  }
  algorithm : csum16;
  output_width : 4;
}
```

```
// Compute the index to the RSS indirection table
modify_field_with_hash_based_offset(packet_metadata.rss_index, 0,
  ipv46_tcp_udp_csum, 65536);
// Store the index to be passed together with the packet
modify_field(intrinsic_metadata.hash, packet_metadata.rss_index);
```

The RSS indirection table is quite simple as it only maps the hash value to a value specified by the user. The output value is then used to specify the egress port of the packet. The snippets below show the M&A table itself on the left, the action on the right and the configuration performed in software below.

```
table table_rss {
  reads {
    packet_metadata.rss_index : exact;
  }
  actions {
    update_egress_spec;
  }
}
```

```
// Overwrites egress port based
// on the RSS indirection table
action update_egress_spec(spec) {
  modify_field(intrinsic_metadata.
    egress_port, spec);
}
```

```
table_rss default action update_egress_spec params ( spec 0 )
table_rss keys ( packet_metadata.rss_index  0 )
                                    action update_egress_spec params ( spec 128 )
table_rss keys ( packet_metadata.rss_index  8 )
                                    action update_egress_spec params ( spec 128 )
table_rss keys ( packet_metadata.rss_index  1 )
                                    action update_egress_spec params ( spec 129 )
table_rss keys ( packet_metadata.rss_index  9 )
                                    action update_egress_spec params ( spec 129 )
```

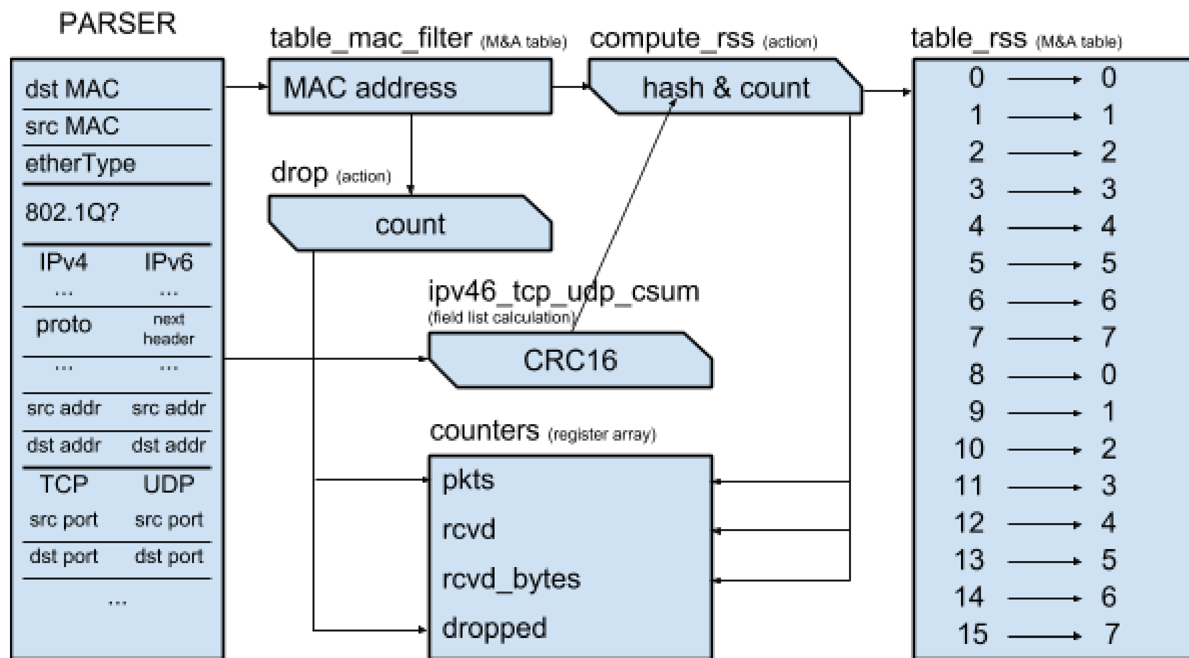
```
...
```

Overall Picture of the P4 Pipeline

Let's put everything together into a single processing pipeline. Recall what topics we have discussed:

Feature	P4 construct
Protocol stack	parser
MAC address filter	M&A table + run time configuration
Counting packets and bytes	register array + run time manipulation
RSS	field list computation, M&A table (indirection table) + run time configuration

The schematics below shows the interconnection of the tasks we have discussed. It shows that the pipeline can be quite complex, M&A tables can be chained like MAC address filter and RSS indirection table, actions can be compound and manipulate with stateful information like counters, compute checksums and perform other tasks.



Complete source codes of the example can be downloaded from our GitHub account. They can be verified using the behavioral model (bmv2) or synthesized in Netcope P4 cloud environment. The cloud environment also features test lab with real hardware where you can test generated firmware with traffic captured in PCAP files.

There is not much more source code to build a P4 NIC than there is presented in this whitepaper. The total number of the lines of the P4 source code (excluding license headers) is 285. Not too bad for a 100Gbps-capable NIC with MAC address filtering, packet and byte counting and RSS. Eh? Read more about using Netcope P4 in whitepapers about in-band network telemetry (INT) and segment routing (SRv6). Feel free to contact Netcope Technologies for more information about the technology.