THE BOOK OF SCHC

Ivan MARTINEZ, Laurent TOUTAIN

IMT ATLANTIQUE

Published 18 juillet 2022

Table of Contents

Acronyms

AS Application Server
CBOR Concise Binaire Object Representation
CoAP Constrained Application Protocol
IP Internet Protocol
IPv4 Internet Protocol version 4
IPv6 Internet Protocol version 6
LPWAN Low Power Wide Area Network
LNS LoRaWAN Network Server

1. Getting started

OpenSCHC offers an easy way to understand and experiment the compression/decompression mechanism as well as the fragmentation protocol. We start be illustrating compression and fragmentation process defined in RFC 8724. To simplify experiments, OpenSCHC does not require to run on a LPWAN network, experiments can run on a regular network using UDP tunnels.

1.1 Installation

1.1.1 Download code

To download openSCHC, just clone the openschc repository on github.

```
> git clone git@github.com:openschc/openschc.git
```

and use the scapy branch.

```
> git checkout scapy
```

1.1.2 Setting up the environment

For testing SCHC compression and fragmentation d-features, we need to generate some IPv6 packets, that will be caught by openSCHC and processed.

An IPv6 prefix is needed for that traffic. A simple way to get a global IPv6 prefix is to use Hurricane Electric tunnel broker : Hurricane Electric

- Go to that website `https://tunnelbroker.net/`,
- Create an account if needed or just log in,
- Select *Create another Tunnel*,
- enter the IPv4 address of the machine running openSCHC and a location,

Once the tunnel created, you will find some configuration examples, that will help you to configure your machine.

For Ubuntu 20, it will be in the `/etc/netplan` directory : `netplan`

```
> cat /etc/netplan/50-cloud-init.yaml
# This file is generated from information provided by
# the datasource.  Changes to it will not persist across an instance.
# To disable cloud-init's network configuration capabilities, write a file
# /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg with the following:
# network: {config: disabled}
network:
    ethernets:
        ens3:
            dhcp4: true
            match:
                macaddress: fa:16:3e:e9:db:5d
            addresses:
            - "2001:41d0:404:200:0:0:0:3a86/64"
            gateway6: "2001:41d0:0404:0200:0000:0000:0000:0001"
            set-name: ens3
    version: 2
    tunnels:
      he-ipv6:
        mode: sit
        remote: 216.66.87.102
        local: 51.91.121.182
        addresses:
          - "2001:470:1f20:1d2::2/64"
        gateway6: "2001:470:1f20:1d2::1"
```

Type the following comment to validate your new network configuration :

```
>sudo netplan try
Warning: Stopping systemd-networkd.service, but it can still be activated by:
  systemd-networkd.socket
Do you want to keep these settings?


Press ENTER before the timeout to accept the new configuration


Changes will revert in 111 seconds
Configuration accepted.
>sudo netplan apply
```

and your machine interfaces looks the following :

```
> ifconfig
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 51.91.121.182  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 2001:41d0:404:200::3a86  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::f816:3eff:fee9:db5d  prefixlen 64  scopeid 0x20<link>
        ether fa:16:3e:e9:db:5d  txqueuelen 1000  (Ethernet)
        RX packets 9543878  bytes 1758163442 (1.7 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9253880  bytes 1479370777 (1.4 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

he-ipv6: flags=209<UP,POINTOPOINT,RUNNING,NOARP>  mtu 1480
        inet6 fe80::335b:79b6  prefixlen 64  scopeid 0x20<link>
        inet6 2001:470:1f20:1d2::2  prefixlen 64  scopeid 0x0<global>
        sit  txqueuelen 1000  (IPv6-in-IPv4)
        RX packets 782095  bytes 447475094 (447.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 71534  bytes 6022694 (6.0 MB)
        TX errors 5  dropped 0 overruns 0  carrier 5  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 7212  bytes 657864 (657.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7212  bytes 657864 (657.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Note that in this example, the machine is multi-homed ; `ens3` interface is configured with the regular IPv6 address and `he-ipv6` with the Hurricane Electric IPv6 address.

1.2 Network Topology

We have currently configured the network for the code part of SCHC. SCHC sends SCHC packet to another device which will also run SCHC. On that second machine, install the github repository, too.

With openSCHC the following topology (cf. figure 3.4 on page 25) will be used for the rest of the tutorial. SCHC packet will be tunneled over the Internet to reach the other SCHC instance.

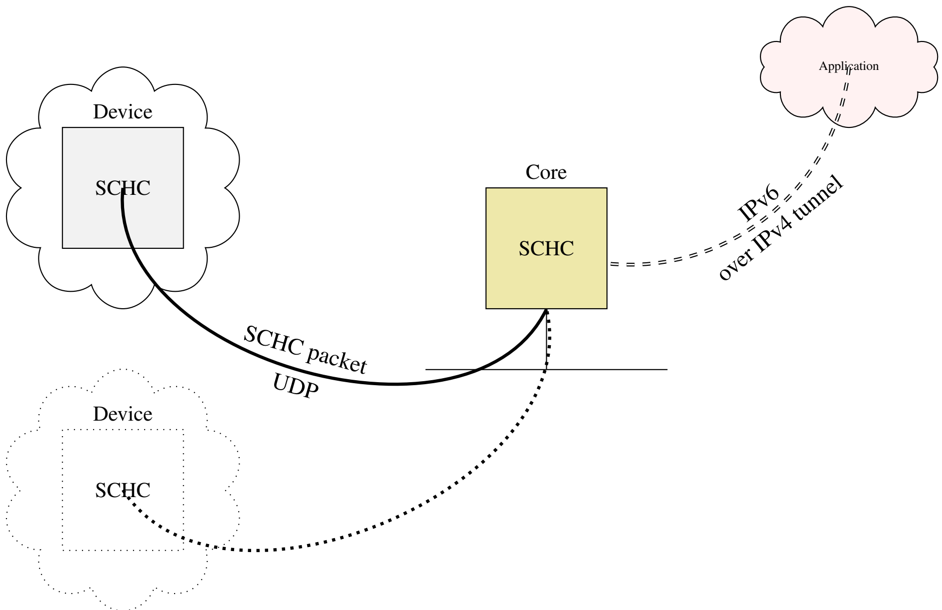

FIGURE 1.1 – Network Topology

In this architecture we see that we will need at least 2 SCHC instances. One located in the core that will act as a router between the IPv6 network and the constrained network, illustrated in our example by an UDP tunnel, but is could also be the LPWAN network.

The others instances of SCHC will be located in the devices. We will define upstream traffic as the communication between the device and the core, and without any surprise, the other direction will be called downstream.

upstream

downstream

Messages exchanged between these entities are called SCHC packets.

SCHC packet

One or several applications will communicate with the devices.

application

2. Rules

SCHC works between two entities. Usually, one is a constrained end-device and the other one is a core equipment acting as a router. To allow both of them to perform the same actions, or more precisely one action and a reverse action, such as compression and decompression or fragmentation and reassembly, a common set of rules must be shared between these two entities.

The distinction between a device and a core allows to differentiate communication going upstream from the device to the core and downstream in the opposite direction. The device usually owns a limited number of rules corresponding to the traffic generated and received by that device. On the contrary, the core instance of SCHC must be aware of all the devices' rules. Even if all the devices have the same behavior, SCHC treats each device individually.

rule ID

All rules are identified by a rule ID. The rule ID is a unique binary sequence for an association between a device and a core. The rule ID length is not specified by the standard and are chosen when the rules are specified. They must be different inside a set of rule. The only constraint is that rule ID cannot overlap.

For instance :

- `0` and `111111` are two valid rules ID
- `01` and `0101` not not valid since they share the same first two bits.

Rules may be represented in binary, but the decimal notation is also used for compactness. In this book, we will represent a rule ID with the `rule ID value/rule ID length`. For instance `3/8` indicates a rule ID stored on 8 bits with value 3.

This notation can be misleading, if `1/2` and `127/8` taken from the example above are overlapping since they both start with `01`, `12/4` and `12/6` are two valuable rules, since when written in binary `1100` and `001100` do not share a common binary sequence.

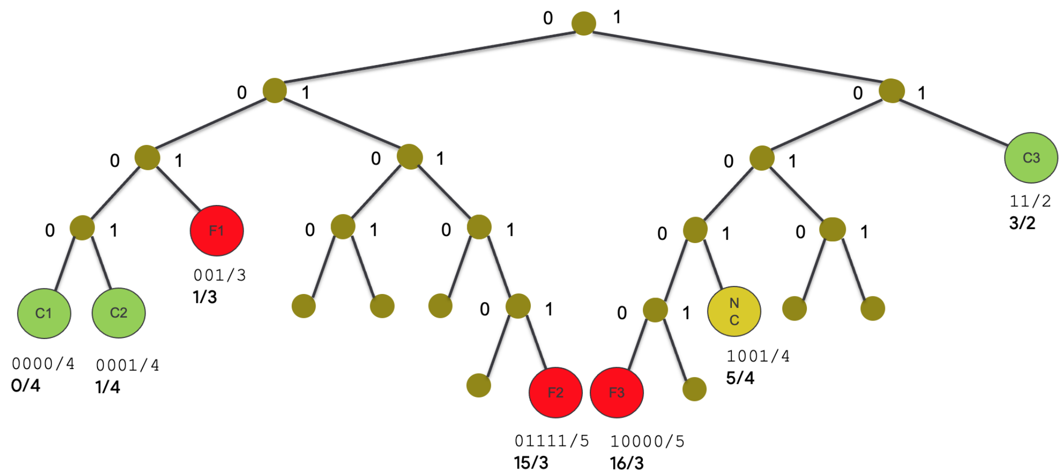

FIGURE 2.1 – Example of binary tree associated to rule IDs

2.1 openSCHC rules

In openSCHC, a rule is defined in a JSON object :

```
{
"RuleID" : 12,
"RuleIDLength" : 4
}
```

There is two different rule formats, one for compression and the other for fragmentation. Compression Rules contains the keyword `Compression` followed by an array, as shown the the minimal example below : Compression

```
{
"RuleID" : 12,
"RuleIDLength" : 4,
"Compression" : []
}
```

Fragmentation rules uses the `Fragmentation` keyword followed by an object giving fragmentation parameters : Fragmentation

```
{
"RuleID" : 12,
"RuleIDLength" : 6,
"Fragmentation" : {
  "FRMode" : "NoAck",
  "FRDirection" : "UP"
}
```

Fragmentation and compression rules share the same rule space, by construction a rule cannot be both compression and fragmentation. An attentive reader may have also noticed that a fragmentation direction is present in the rule description. In fact, compression rules are bi-directionnal and may be used to compress an header by the device and the core, but fragmentation rules are oriented.

NoCompression

Finally a `NoCompression` keyword also exists and is the equivalent to the above example, indicating an uncompressed packet.

```
{
"RuleID" : 666,
"RuleIDLength" : 10,
"NoCompression" : []
}
```

2.2 Rule Manager

Rule Manager

The Rule Manager plays an important role in openSCHC. It has different goals :

- check the rule correctness
- add default parameters, to simplify rule notation
- display rules in a more synthetic format
- find the best rule to compress or fragment a packet
- find a rule from its rule ID.

2.2.1 Rule definition

The above listing gives a simple python program used to manage the two rules we described before.

Listing 2.1 – rm1.py

```python
from gen_rulemanager import *

RM = RuleManager()

rule1100 =   {
  "RuleID" : 12,
  "RuleIDLength" : 4,
  "Compression" : []
}

rule001100 =   {
  "RuleID" : 12,
  "RuleIDLength" : 6,
  "Fragmentation" :  {
    "FRMode" : "noAck",
    "FRDirection" : "UP"
  }
}

RM.Add(dev_info=rule1100)
RM.Add(dev_info=rule001100)

RM.Print()
```

gen_rulemanager This program import the `gen_rulemanager` module defining the `RuleManager` class. RuleManager

gen_rulemanager The `Add` method allows to add a new rule. Add

gen_rulemanager OpenSCHC displays, with the `Print` : Print

```
****************************************
Device: None
/-------------------------\
|Rule 12/4          1100  |
|---------------+---+--+--+------------------------------+-------------+----------------\
\---------------+---+--+--+------------------------------+-------------+----------------/
/-------------------------\
|Rule 12/6        001100  |
!=========================+=============================================================\
!^ Fragmentation mode : noAck    header dtag 2 Window  0 FCN  3                     UP ^!
!^ No Tile size specified                                                              ^!
!^ RCS Algorithm: crc32                                                                ^!
\=======================================================================================/
```

Compression rules contain the field descriptions (here empty) and Fragmentation rules contain the fragmentation parameters. Note that for fragmentation, the rule manager added some default parameters corresponding to no Ack behavior. no Ack

2.2.2 Set of rules

A device should contain a set of rules related to compression and fragmentation. In OpenSCHC, the SoR (Set of Rules) is a JSON array. The following program has the same behavior as the previous one, but an array for rules is provided. Set of Rules

Listing 2.2 – rm2.py

```python
from gen_rulemanager import *

RM = RuleManager()

rule1100 =   {
 "RuleID" : 12,
 "RuleIDLength" : 4,
 "Compression" : []
}

rule001100 =   {
 "RuleID" : 12,
 "RuleIDLength" : 6,
 "Fragmentation" : {
   "FRMode": "noAck",
   "FRDirection" : "UP"
 }
}

RM.Add(dev_info=[rule1100, rule001100])

RM.Print()
```

2.2.3 Device identifier

You may have noticed that, in the previous examples, the device was displayed as None. This can be appropriate when SCHC is instantiated on a device, since there is no ambiguity as to which device the rule set applies to. Conversely, when the SCHC instance is on the core network side, the set of rules must be associated with a device ID.

device identifier

The device identifier is structured as a technology and an identifier in that technology :

- UDP tunnel — UDP tunnel, as in the platform built in Chapter 1.1 on page 6. In that case the technology is `udp` and the identifier is the tunnel IP address end-point on the device side, followed by the port number used on that end point. For instance : `udp:83.199.24.39:8888` [1].
- LoRaWAN — a LoRaWAN device (*not yet implemented*), the technology is `lorawan` and the identifier is the *devEUI* [2].

Add — gen_rulemanager

Rules associated with a Device ID can be directly stored into the rule manager through the `Add` method as follows :

```
RM.Add(device="1234567890", dev_info=[rule1100, rule001100])
```

Alternately, the JSON structure would be the following :

```
{
    "DeviceID": 1234567890,
    "SoR" : [ ..... ]
}
```

2.2.4 JSON file

Rules can also be stored in a JSON file, the Rule Manager`Add` method uses the `file` argument. For instance :

```
rm.Add(file="icmp.json")
```

2.3 Conclusion

We know the basic structure of rules, how to add them to the rule manager, so now, let's try to compression and fragment some traffic.

1. If the device is behind a NAT, the IP address used must be the global address assigned to the NAT.
2. when a JSON structure is manipulated, the DeviceID literal must be expressed in decimal, not hexadecimal

3. Compressing Ping6

3.1 Introduction

In this chapter, we are going to setup the compression of a ping6 request. An ICMPv6 request is composed of an IPv6 header, followed by a ICMPv6 request message.

ICMPv6 request

```
Internet Protocol Version 6, Src: 2a01:cb08:903a:bd00:a8c4:5c6d:c2b5:84be, Dst:
2001:470:1f21:1d2::1
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
        .... 0000 00.. .... .... .... .... .... = Differentiated Services Codepoint: Default (0)
        .... .... ..00 .... .... .... .... .... = Explicit Congestion Notification: Not ECN-Capable Transport (0)
    .... .... .... 1000 0000 1000 0000 0000 = Flow Label: 0x80800
    Payload Length: 16
    Next Header: ICMPv6 (58)
    Hop Limit: 56
    Source: 2a01:cb08:903a:bd00:a8c4:5c6d:c2b5:84be
    Destination: 2001:470:1f21:1d2::1

Internet Control Message Protocol v6
    Type: Echo (ping) request (128)
    Code: 0
    Checksum: 0x3982 [correct]
    [Checksum Status: Good]
    Identifier: 0x70ef
    Sequence: 699

    Data (8 bytes)
        Data: 609ab0db000aecb7
        [Length: 8]

0000  60 08 08 00 00 10 3a 38 2a 01 cb 08 90 3a bd 00   ‘.....:8*....:..
0010  a8 c4 5c 6d c2 b5 84 be 20 01 04 70 1f 21 01 d2   ..\m.... ..p.!..
0020  00 00 00 00 00 00 00 01 80 00 39 82 70 ef 02 bb   ..........9.p...
0030  60 9a b0 db 00 0a ec b7
```

The following compression rule can be applied to that traffic.

Listing 3.1 – fig-rule-ping

```
{
    "DeviceID" : "udp:83.199.24.39:8888",
    "SoR" : [
        {
            "RuleID": 6,
            "RuleIDLength": 3,
            "Compression": [
                {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.TC",  "TV": 0, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
```

```
            {"FID": "IPV6.LEN",          "MO": "ignore","CDA": "compute-length"},
            {"FID": "IPV6.NXT", "TV": 58, "MO": "equal", "CDA": "not-sent"},
            {"FID": "IPV6.HOP_LMT", "TV" : 64,"MO": "ignore","CDA": "not-sent"},
            {"FID": "IPV6.DEV_PREFIX","TV": "2001:470:1F21:1D2::/64",
                                       "MO": "equal","CDA": "not-sent"},
            {"FID": "IPV6.DEV_IID", "TV": "::1","MO": "equal","CDA": "not-sent"},
            {"FID": "IPV6.APP_PREFIX",        "MO": "ignore","CDA": "value-sent"},
            {"FID": "IPV6.APP_IID",         "MO": "ignore","CDA": "value-sent"},
            {"FID": "ICMPV6.TYPE",  "DI": "DW", "TV": 128,"MO": "equal","CDA": "not-sent"},
            {"FID": "ICMPV6.TYPE",  "DI": "UP", "TV": 129,"MO": "equal","CDA": "not-sent"},
            {"FID": "ICMPV6.CODE",  "TV": 0,  "MO": "equal","CDA": "not-sent"},
            {"FID": "ICMPV6.CKSUM", "TV": 0, "MO": "ignore","CDA": "compute-checksum"},
            {"FID": "ICMPV6.IDENT", "TV": 0,"MO": "ignore","CDA": "value-sent"},
            {"FID": "ICMPV6.SEQNO", "TV": 0,"MO": "ignore","CDA": "value-sent"}

           ]
    } ] }
```

3.2 Understanding compression rules

A rule contains a list of fields description :

Field ID
FID

— starting with a Field ID. openSCHC uses a string structured with the protocol name (IPV6 and ICMPV6 in the rule above). Field ID uses keyword FID.

Target Value

— A Target Value (TV) may be present and contains the value the rule expected to find in the field.

Direction Indicator

— A Direction Indicator (DI) can be associated to a field. In the above example, ICMPV6.TYPE is repeated twice for uplink and downlink. When the direction is not specified, openSCHC considers the field as bi-directionnal. ICMPV6.TYPE

Matching Operator

— A Matching Operator (MO) specifies the comparison between the target Value (TV) and the Field Value (FV) present in the packet to be compressed. RFC 8724 defines 4 MO :
  - ignore — ignore : the FV is ignored therefore, the result of the field comparison is always true.
  - equal — equal : the result of the comparison is true is the value contained in the TV is equal to the value contained if the FV. OpenSCHC allows integer and strings for the comparison.
  - MSB — MSB : is true only if the first bits of the TV are equal to the first bits of the FV. The comparison length in bits is specified in the rule with the keywork MO.val. For string comparison, openSCHC impose a multiple of 8 bits for the length.
  - match-mapping — match-mapping : The TV contains an array of values. The comparison is true, if the FV appears somewhere in the list.

  A rule is selected if all the comparison are true and the packet contains no other fields.

Compression
Decompression Actions

— When a rule is selected, the Compression Decompression Actions (CDA) are applied to the packet fields. RFC 8724 defines several CDAs, among them :
  - not-sent — not-sent : the FV is not sent, the decompressor will use the value stored in the rule to recover the value.
  - value-sent — value-sent : the value is explicitely sent as a residue. The decompressor will use the residue to recover the value.
  - LSB — LSB : the least significant bits are sent as residue. The decompressor will combine the TV and the residue to recover the field. LSB can only be used with a MSB MO. No argument are needed for LSB. The LSB size is defined from the field size and the MSB size.
  - mapping-sent — mapping-sent : the index in the array is sent as residue. The decompressor will use the idenx to recover the value. This CDA can only be used with a match-mapping MO.
  - compute-* — compute-* : the FV is not sent, the decompressor will use a specific algorithm to recover the value, such as compute the length or compute a checksum.

3.3 Compressing an ICMPv6 echo request packet

IPV6.VERSION IPV6.NXT In the above example, the `IPV6.VERSION` field is not sent, the value 6 is stored is the rule. The `IPV6.NXT` follows the same behavior, since ICMPv6 is expected, the `TV` is 58.

IPV6.FL IPV6.HOP_LMT Note that `IPV6.FL` and `IPV6.HOP_LMT` use the ignore/not-sent combination. During the rule selection, the `FL` is not taken into account, but during decompression the value stored in the rule is used. The reason is that IPv6 Flow Label (`IPV6.FL`) can be different from 0, in the capture the value is `0x80800`. ignore/not-sent

IPV6.HOP_LMT For IPv6 Hop Limit (`IPV6.HOP_LMT`) the value cannot be forecast, depends of how many routers are crossed and may vary during from one packet to the other. The rule ignores the value and generate a hop limit of 64. To be valid, this implies that the device will not forward anymore the packet. If the device acts as a router the Hop Limit field must be sent.

The address fields compression is more interesting, there is not source or destination address in the SCHC rule. SCHC defines the device and the application addresses [1]. The rule becomes independent of the direction. OpenSCHC splits the addresses into a 64 bit long prefix and a 64 bit long IID. addresses IID

In our scenario the device address is not sent, since the value is known by the device. On the opposite, we allow any host to ping the device [2]. The destination address is fully sent to the device.

At the ICMPv6 level, the type field is elided regarding the direction. The Echo value (128) is expected downlink ; form the application to the device, and the Echo reply (129) on the other direction. In both cases the ICMPv6 code is expected to be 0.

ICMPv6.IDENT ICMPV6.SEQNO The identifier (`ICMPv6.IDENT`) and sequence number (`ICMPV6.SEQNO`) are not compressed and send as residue.

IPV6.LENGTH ICMPV6.CHECKSUM The other fields `IPV6.LENGTH` and `ICMPV6.CHECKSUM` are not sent and computed by the other side.

The SCHC packet has the following format :

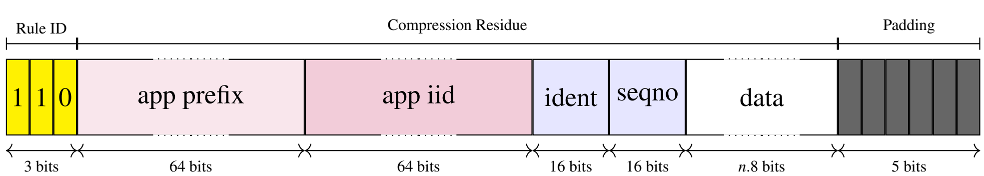

FIGURE 3.1 – ICMPv6 Compression Residue

The rule ID takes 3 bits, followed by the compression residues, the ICMPv6 payload and some padding bits. Since the Rule ID length is 3 bit long and the rest is byte aligned, 5 bits are needed to residue

1. This will apply also for UDP port number
2. This behavior has to be prohibited in a LPWAN network, for security reasons, since in this scenario we are using an UDP tunnel, there is no security issue.

align the SCHC packet on a L2-word (i.e. 1 byte).

3.4 Compression process

In OpenSCHC, SCHC Compresssion/Decompression (CD) and Frgamentation/Reassembly (FR) processes are done by the SCHC Machine.

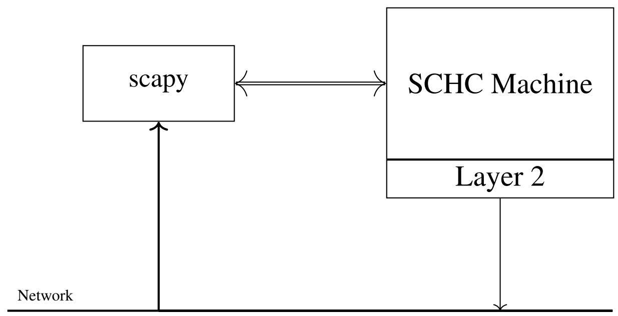

FIGURE 3.2 – Ping gateway architecture

Scapy Figure 3.2 gives the architecture of the ping gateway. Scapy *sniff* function gets traffic on the network. This traffic is composed of IPv6 packets that could be compressed and tunneled SCHC packets that are decompressed.

3.4.1 The code

The scapy module calls the SCHC Machine, in charge of CD-FR processes. Layer 2 allows to send decompressed packet or a SCHC packet on the network.

Main program

Listing 3.2 – ping_core1.py

```
import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

from scapy.all import *

import gen_rulemanager as RM
from protocol import SCHCProtocol
from scapy_connection import *
from gen_utils import dprint, sanitize_value

import pprint
import binascii
import socket
import ipaddress
```

This program `ping_core1.py` shows how to compress ICMPv6 packets and send them on a UDP tunnel to a device. It starts with the modules importation from scapy, openSCHC and some regular python modules. Note the importation of the `scapy_connection` module, located on the working directory. This module implements the scheduler and the methods to send packets. ping_core1.py

scapy_connection

```python
# Create a Rule Manager and upload the rules.
rm = RM.RuleManager()
rm.Add(file="icmp1.json")
rm.Add(file="icmp2.json")
rm.Print()
```

The compression process starts with the creation of a Rule Manager `rm` to include the rules from file `icmp.json` (cf. Listing 3.3), which contains the compression rules we saw below and a NoAck fragmentation rule.

Listing 3.3 – rule icmp1.json

```json
{
    "DeviceID" : "udp:83.199.24.39:8888",
    "SoR" : [
         {
            "RuleID": 6,
            "RuleIDLength": 3,
            "Compression": [
                {"FID": "IPV6.VER", "TV": 6, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.TC",  "TV": 0, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.FL",  "TV": 0, "MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.LEN",          "MO": "ignore","CDA": "compute-length"},
                {"FID": "IPV6.NXT", "TV": 58, "MO": "equal", "CDA": "not-sent"},
                {"FID": "IPV6.HOP_LMT", "TV" : 255,"MO": "ignore","CDA": "not-sent"},
                {"FID": "IPV6.DEV_PREFIX","TV": "2001:470:1F21:1D2::/64",
                                               "MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.DEV_IID", "TV": "::1","MO": "equal","CDA": "not-sent"},
                {"FID": "IPV6.APP_PREFIX",          "MO": "ignore","CDA": "value-sent"},
                {"FID": "IPV6.APP_IID",          "MO": "ignore","CDA": "value-sent"},
                {"FID": "ICMPV6.TYPE",  "DI": "DW", "TV": 128,"MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.TYPE",  "DI": "UP", "TV": 129,"MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.CODE",  "TV": 0,  "MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.CKSUM", "TV": 0, "MO": "ignore","CDA": "compute-checksum"},
                {"FID": "ICMPV6.IDENT", "TV": 0,"MO": "ignore","CDA": "value-sent"},
                {"FID": "ICMPV6.SEQNO", "TV": 0,"MO": "ignore","CDA": "value-sent"}

            ]
         },{
                "RuleID" : 12,
                "RuleIDLength" : 11,
                "Fragmentation" : {
                        "FRMode": "NoAck",
                        "FRDirection": "DW"
                }
        }
    ]
  }
```

IPV6.DEV_PREFIX

This rule must be adapted to your environment. The `DeviceID` must reflect the IPv4 public address of your device and the Target Value of `IPV6.DEV_PREFIX` must be set to a IPv6 prefix of your domain, such as the ones allocated by Hurricane Electric.

```python
# Start SCHC Machine
POSITION = T_POSITION_CORE

socket_port = 0x5C4C
tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tunnel.bind(("0.0.0.0", socket_port))

lower_layer = ScapyLowerLayer(position=POSITION, socket=tunnel, other_end=None)
system = ScapySystem()
scheduler = system.get_scheduler()
```

The position of the SCHC instance has to be specified. In our example, we define our role as a core. Then, we create the UDP tunnel to receive SCHC packets from devices. The default port number is `0x5C4C`.

A `lower_layer` is created to be used by the SCHC machine when SCHC packet will have to be scheduler scapy_connection

sent, and a `system` is created to manage events. The latter includes a scheduler. Both are defined in the `scapy_connection` module.

From that `system` a reference to the `scheduler` is extracted to be used in the `processPkt` function as a global variable.

```
schc_machine = SCHCProtocol(
    system=system,           # define the scheduler
    layer2=lower_layer,      # how to send messages
    role=POSITION,           # DEVICE or CORE
    verbose = True)
schc_machine.set_rulemanager(rm)
```

SCHC machine

The previously created classes are regrouped arounf a SCHC machine which will process packets for compression and fragmentation. A `schc_machine` instance is created and the previously created rule manager is associated to it.

```
sniff(prn=processPkt, iface="ens3") # scappy cannot read multiple interfaces --> use sudo
```

Finally, the scapy `sniff` function is called. This last line never returns. The `processPkt` function is called each time a packet is received on interface `ens3`[3].

Frame processing

```
def processPkt(pkt):
    """ called when scapy receives a packet, since this function takes only one argument,
    schc_machine and scheduler must be specified as a global variable.
    """

    scheduler.run(session=schc_machine)
```

run scapy_scheduler

The `processPkt` function starts by calling the SCHC machine through the `run` method. This function must be called regularly, which is the case when some traffic occurs in the network, so it is important not to filter too much incoming traffic.

The functions looks for two types of packets :

— SCHC tunneled packets coming from devices to the UDP port we defined (`0x5C4C` in our case),

— IPv6 tunneled packets from Hurricane Electric easily recognisable through the use of IP proto 41.

```
    # look for a tunneled SCHC pkt
    if pkt.getlayer(Ether) != None: #HE tunnel do not have Ethernet
        e_type = pkt.getlayer(Ether).type
        if e_type == 0x0800:
            ip_proto = pkt.getlayer(IP).proto
            if ip_proto == 17:
                udp_dport = pkt.getlayer(UDP).dport
                if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
                    print ("tunneled SCHC msg")
                    schc_pkt, addr = tunnel.recvfrom(2000)
                    other_end = "udp:"+addr[0]+":"+str(addr[1])
```

3. Scapy allowed to listen simultaneously to several interfaced, for example `["ens3", "he-ipv6"]`, but since this feature returns sometime some errors, we prefer to listen only to interface `"ens3"` which will carried tunneled IPv6 packets from Hurricane Electrics.

```
                    print("other end =", other_end)
                    r = schc_machine.schc_recv(other_end, schc_pkt)
                    print (r)
```

First, a test done to ensure that the frame has an Ethernet encapsulation [4]. Then we filter the frames to keep only those with :

— an Ethertype equal to 0x800 indicating an IPv4 packet,
— an IPv4 protocol number equal to 17 indicating an UDP message,
— and a destination port of `0x5C4C` indicating a SCHC packet.

If all these conditions are met, we are sure that the socket has received a packet from a device, we will see in the following step how to process it.

Let's focus on the incoming Ping request.

```
            elif ip_proto==41:
                schc_machine.schc_send(bytes(pkt)[34:])
```

If the IP protocol value is 41, then we have a tunneled IPv6 packet over IPv4. The first 34 bytes corresponding to the Ethernet and IPv4 headers are removed, and the resulting IPv6 packet is send
SCHCProtocol to the SCHC machine for compression with the `schc_send` method. schc_send

This function will trigger the emission of SCHC packet :

— if the compressed SCHC packet enters in one frame, it is directly sent,
— otherwise a fragmentation rule is applied and several SCHC packets can be sent.

The function returns :

— `None` if a compression rule and optionally a fragmentation rule are found.
— False if a rule is missing.

The SCHC packet will be sent using `scapy_connection` functions. `scapy_connection`

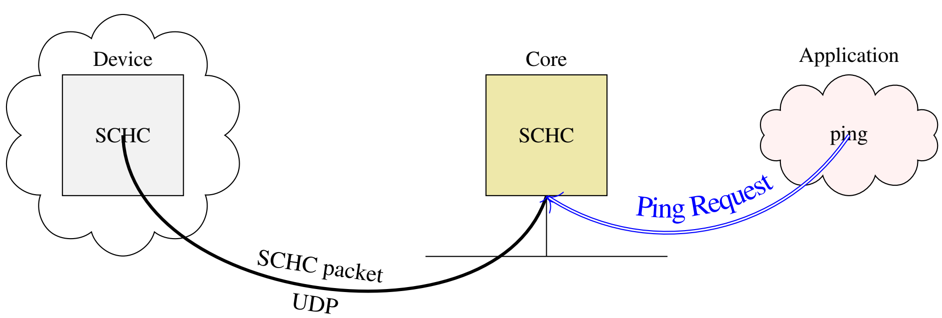

FIGURE 3.3 – SCHC core processing a Ping Request from an Application

3.4.2 The execution

Start the program, in sudo mode :

```
$ sudo python3.9 ping_core1.py
```

4. That wouldbe the case, if the Hurrican Electric interface was directly listened by scapy.

The program will display the rules and cursor spins, indicating the SCHC machine is running [5].

Start pinging the device, from any host in the Internet with IPv6 connectivity. to the IPv6 address of the device defined in the rule, the `-c 1` limits the number of ping messages.

```
$ ping6 2001:470:1f21:1d2::1 -c 1
```

Of course there is no answer to the ping, the openSCHC core instance displays some messages. They can be avoided, if the `verbose` argument is set to False or not specified when creating the `schc_machine` object.

```
schc recv-from-l3 None None b'`\x05\x0c\x00\x00\x10:8*\x01\xcb\x08\x90:\xbd\x00I\xe0\xa3\xec\x01Vv\x9c \x01\x04p...'
schc parser {('IPV6.VER', 1): [6, 4], ('IPV6.TC', 1): [0, 8], ('IPV6.FL', 1): [330752, 20]...}
schc compression rule {'RuleID': 6, 'RuleIDLength': 3, 'Compression': [{'FID': 'IPV6.VER', 'FL': 4, ...}
schc compression result b'\xc5\x40\x39\x61\x12\x07\x57\xa0\x09\x3c\x14\x7d\x80\x2a\xce...'/227
schc fragmentation not needed size=227
```

This trace gives the openSCHC compression process divided into several steps :

— Parse the packet ; from a sequence of byte received on the network, create a list of fields containing the field identification and their associated value.

— Find a valid compression rule ; ask the rule manager to find a rule matching the parsed packet. The rule selection will also provide the device ID.

— Apply the compression rule.

— Send the SCHC packet to the device ID.

In more details :

— The first line (`recv-from-l3`) dumps the original IPv6 packet received by the compressor, corresponding in hexadecimal to :

```
60050c0000103a382a01cb08903abd0049e0a3ec0156769c200104701f2101d20000000000000001800051fb48b20000609f882600060ed2
```

— The second line (`parser`) shows three elements returned by the parser. The first one is the list of fields, the second one is the data following the list of fields and the third one is a status code. The parsed headers are displayed figure 3.4 :

Listing 3.4 – Parsed IPv6/ICMPv6 header fields

```
{('ICMPV6.CKSUM', 1): [20987, 16],
 ('ICMPV6.CODE', 1): [0, 8],
 ('ICMPV6.IDENT', 1): [18610, 16],
 ('ICMPV6.SEQNO', 1): [0, 16],
 ('ICMPV6.TYPE', 1): [128, 8],
 ('IPV6.APP_IID', 1): [b'I\xe0\xa3\xec\x01Vv\x9c', 64],
 ('IPV6.APP_PREFIX', 1): [b'*\x01\xcb\x08\x90:\xbd\x00', 64],
 ('IPV6.DEV_IID', 1): [b'\x00\x00\x00\x00\x00\x00\x00\x01', 64],
 ('IPV6.DEV_PREFIX', 1): [b' \x01\x04p\x1f!\x01\xd2', 64],
 ('IPV6.FL', 1): [330752, 20],
 ('IPV6.HOP_LMT', 1): [56, 8, 'fixed'],
 ('IPV6.LEN', 1): [16, 16, 'fixed'],
 ('IPV6.NXT', 1): [58, 8, 'fixed'],
 ('IPV6.TC', 1): [0, 8],
 ('IPV6.VER', 1): [6, 4]}
```

As shown in figure 3.4, a header description is a dictionnary where keys are tuple Field ID and position [6], and the value is the tuple containing field value and field size in bits.

5. The number of received packets determines the spinning speed. It takes 10 packets to change the cursor appearance.
6. In this example, position is always 1 since no field is repeated several time.

The next return element is the data, i.e. the bytes following the parsed header :

```
b'`\x9f\x88&\x00\x06\x0e\xd2'
```

and the error code is None since the parser recognize all the fields.
This no surprise, the rule 6/3 matches.

- The third line `compression result` gives the SCHC packet. Note the `/227`[7] at the end, indicating the length in bits. Converted in hexadecimal, we have :

```
b'c5403961120757a0093c147d802aced3891640000c13f104c000c1da40'
```

- Finally, no fragmentation is required, so the SCHC packet is directly sent on the tunnel. A frame capture of UDP frame with port 0x5C4C gives :

```
>sudo tcpdump -nXi ens3 udp port 0x5C4C
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
16:41:48.609729 IP 51.91.121.182.23628 > 83.199.24.39.8888: UDP, length 29
        0x0000:  4500 0039 ac95 4000 4011 751f 335b 79b6  E..9..@.@.u.3[y.
        0x0010:  53c7 1827 5c4c 22b8 0025 1936[c540 3961  S..'\L"..%.6.@9a
        0x0020:  1207 57a0 093c 147d 802a ced3 8e5f c000  ..W..<.}.*..._..
        0x0030:  0c13 fbb5 8000 d951 80]                    .......Q.
```

3.5 The Decompression Process

Now that we send SCHC packet to the device, lets process the SCHC packet on this side.

3.5.1 Decompression

Let's do the same operation on the device side. The code is almost the same, as the core SCHC. It is important to note that the rules are excactly the same as the one we used in the core SCHC.

Listing 3.5 – ping_device1.py

```python
POSITION = T_POSITION_DEVICE

from requests import get

ip = get('https://api.ipify.org').text

socket_port = 8888
tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tunnel.bind(("0.0.0.0", socket_port))

device_id = 'udp:'+ip+":"+str(socket_port)
print ("device_id is", device_id)
```

The position is changed to `T_POSITION_DEVICE`. Since the `device_id`, in our example, is based on a public IP address, the call to `https://api.ipify.org` returns the IPv4 public address of the device. We create a UDP socket for port 8888.

```python
lower_layer = ScapyLowerLayer(position=POSITION, socket=tunnel, other_end=None)
system = ScapySystem()
```

7. 227 % 8 = 3. Since the all the residues are byte alligned, the 3 represent the rule ID length. 5 bits of padding will have to be addeed.

```
scheduler = system.get_scheduler()
schc_machine = SCHCProtocol(
    system=system,            # define the scheduler
    layer2=lower_layer,       # how to send messages
    role=POSITION,            # DEVICE or CORE
    verbose = True)
schc_machine.set_rulemanager(rm)

sniff(prn=processPkt, iface="en0") # scappy cannot read multiple interfaces
```

The `lower_layer`, `system` and `schc_machine` are declared the same way as in the core SCHC. The interface name is adapted to its name on the device side.

```
            if ip_proto == 17:
                udp_dport = pkt.getlayer(UDP).dport
                if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
                    print ("tunneled SCHC msg")
                    schc_pkt, addr = tunnel.recvfrom(2000)
                    r = schc_machine.schc_recv(device_id=device_id, schc_packet=schc_pkt)
                    print (r)
```

It's time now, to look how a SCHC packet is processed. If the IP protocol is 17, the upper layer is UDP. We check the destination port (here 8888) to see if the message is for us. If it is the case, then we can get officially the message from the socket, to clear the buffers. We read the socket and get the message (`schc_pkt`) and the device address (`addr`) which send the SCHC packet.

schc_recv

Then we call the function `schc_recv` with the device ID we computed before and the SCHC packet we just received. This function returns : SCHCProtocol

- `None`. This is the normal behavior when receiving a fragment. Only the last fragment will return the full messages. Intermediary fragments will return this value.
- a byte array with the uncompressed packet.
- `False` when a error occurs.

Since we are just doing compression, the `schc_recv` function returns the full packet. You may notice that some fields have changed. That's the case for IPV6.FL, PV6.HOP_LMT. IPV6.FL PV6.HOP_LMT

3.5.2 Device optimization

In the previous program, we started to reconstruct the packet. It can be then processed normally or even forwarded to another destination. Nevertheless, we can optimize the behavior and process directly the SCHC packet. This simplify the protocol stack implementation in a very constrained device.

Listing 3.6 – ping_device2.py

```
import gen_rulemanager as RM
from protocol import SCHCProtocol
from scapy_connection import *
from gen_utils import dprint, sanitize_value
from gen_bitarray import *
```

We have to identify the rule in the SCHC packet, we just receive. We need to transform the Byte Array, into a bit array. Therefore, the `gen_bitarray` module is imported [8]. `gen_bitarray`

Listing 3.7 – ping_device2.py

```
                if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
                    print ("tunneled␣SCHC␣msg")
                    schc_pkt, addr = tunnel.recvfrom(2000)
                    schc_bbuf = BitBuffer(schc_pkt)
                    rule = rm.FindRuleFromSCHCpacket(schc=schc_bbuf, device=device_id)
                    if rule[T_RULEID] == 6 and rule[T_RULEIDLENGTH]== 3:
                        print ("ping")
                        tunnel.sendto(schc_pkt, addr)

                    else:
```

gen_bitarray

The SCHC packet is first transformed into a bit array with the `BitBuffer` constructor. BitBuffer

gen_rulemanager

We can call the function `FindRuleFromSCHCpacket` with the device ID and the SCHC packet. If a rule ID is found, the function will return it [9]. FindRuleFromSCHCpacket

We check the rule, to see if it correspond to the ping dedicated rule 6/3. We need to test the value and the length, since two different rules may have the same value (remember chapter 2 on page 9).

If the rule in the SCHC packet corresponds, we directly answer to the core. We use a trick in the rule definition linked to direction (cf. rule 3.3 on page 18).

Listing 3.8 – extract from rule 6/3 for ping traffic

```
{
    "DeviceID" : "udp:83.199.24.39:8888",
    "SoR" : [
        {
            "RuleID": 6,
            "RuleIDLength": 3,
            "Compression": [
...
                {"FID": "ICMPV6.TYPE",  "DI": "DW", "TV": 128,"MO": "equal","CDA": "not-sent"},
                {"FID": "ICMPV6.TYPE",  "DI": "UP", "TV": 129,"MO": "equal","CDA": "not-sent"},
...
               ]
```

ICMPV6.TYPE

As you have may notice, the ICMPV6.TYPE is defined twice : one for downlink direction, (i.e. from the network to the device) and one in the uplink direction. For the former, the code for an ICMPv6 Echo Request (128) is set and for the latter, it corresponds to the code of an ICMPv6 Echo Reply (129).

So if we just echo the SCHC packet, it will be as a response, and application address, sequence number and identifier it carries will be included in the response. Therefore we just need to send the packet to the core SCHC we got the address when we read the socket.

We can optimize the process on the device. The rule 6/3 (cf. rule figure 3.3 on page 18) has been associated to the ping traffic. For the downlink, the type is an Echo Request (128) and for the uplink the type is an Echo Reply (129). The other fields remain unchanged in both directions.

Let's continue with this hack and see how the core should process the SCHC packet.

8. Other importations will be used when IPv6 packet will be manipulated.
9. We are very optimistic in this example. If no rule is found, the function will return None and the program will crash.

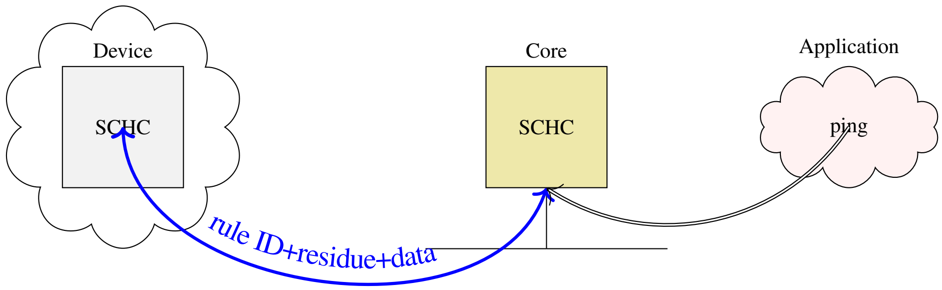

FIGURE 3.4 – SCHC core processing a Ping Request from an Application

3.6 Generating packets

Now, the SCHC core instance receives SCHC packets from the device through the UDP tunnel. We can decompress the SCHC message and send it to the Internet.

```
if ip_proto == 17:
    udp_dport = pkt.getlayer(UDP).dport
    if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
        print ("tunneled␣SCHC␣msg")
        schc_pkt, addr = tunnel.recvfrom(2000)
        other_end = "udp:"+addr[0]+":"+str(addr[1])
        print("other␣end␣=", other_end)
        uncomp_pkt = schc_machine.schc_recv(device_id=other_end,
                             schc_packet=schc_pkt)
        if uncomp_pkt != None:
            uncomp_pkt[1].show()
            send(uncomp_pkt[1], iface="he-ipv6")
elif ip_proto==41:
    schc_machine.schc_send(bytes(pkt)[34:])
```

The processing is almost the same as for the device, if a UDP message with destination port corresponding to the tunnel (i.e. `0x5C4C`) is detected in the packet, then the socket is read. It differs in the way the device is identified. Here, the sender address returned by the socket is used to build the device ID (stored in `other_end` variable). device ID

schc_recv The call to the SCHC machine `schc_recv` method may return the uncompressed packet,in the scapy format [10]. SCHCProtocol

show If the packet is returned, the `show` function display it and then it is sent on the Hurricane Electric interface. scapy

10. If the SCHC packet was a fragment, it will have been buffered until the last fragment is received. In that case None would have been returned.

4. Fragmenting Ping6

In this chapter, in addition to compressing a ping6 request we will fragment it. For that, we will create a ping6 request that exceeds the maximum L2 MTU allowed for the device :

```
ping6 -c 1 -s 200 2001:470:1f21:1d2::2
```

```
##[ IPv6 ]##
    version   = 6
    tc        = 0
    fl        = 63154
    plen      = 58
    nh        = ICMPv6
    hlim      = 52
    src       = 2001:41d0:302:2200::13b3
    dst       = 2001:470:1f21:1d2::2
##[ ICMPv6 Echo Request ]##
       type      = Echo Request
       code      = 0
       cksum     = 0xbc64
       id        = 0x260
       seq       = 0x1
       data      = '\\xa5\\xf3\x1cb\x00\x00\x00\x00\x9d\r\x00\x00\x00\x00\x00\x10\x1 1\x12\x13\x14\x15\x16\x17\x18
                    \x19\x1a\x1b\x1c\x1d\x1e\x1f!"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`
                    abcdefghijklmnopqrstuvwxyz{|}~\x7f\\x80\\x81\\x82\\x83\\x84\\x85\\x86\\x87\\x88\\x89\\x8a\\x8b
                    \\x8c\\x8d\\x8e\\x8f\\x90\\x91\\x92\\x93\\x94\\x95\\x96\\x97\\x98\\x99\\x9a\\x9b\\x9c\\x9d\\x9e
                    \\x9f\\xa0\\xa1\\xa2\\xa3\\xa4\\xa5\\xa6\\xa7\\xa8\\xa9\\xaa\\xab\\xac\\xad\\xae\\xaf\\xb0\\xb1
                    \\xb2\\xb3\\xb4\\xb5\\xb6\\xb7\\xb8\\xb9\\xba\\xbb\\xbc\\xbd\\xbe\\xbf\\xc0\\xc1\\xc2\\xc3\\xc4
                    \\xc5\\xc6\\xc7''

    0000  FA 16 3E E9 DB 5D A2 C8 13 C9 D8 BC 08 00 45 00   ..>..]........E.
    0010  01 0C 79 DB 40 00 F0 29 33 33 D8 42 57 66 33 5B   ..y.@..)33.BWf3[
    0020  79 B6 60 00 F6 B2 00 D0 3A 34 20 01 41 D0 03 02   y.`.....:4 .A...
    0030  22 00 00 00 00 00 00 00 13 B3 20 01 04 70 1F 21   "......... ..p.!
    0040  01 D2 00 00 00 00 00 00 00 02 80 00 88 08 02 6B   ...............k
    0050  00 01 41 00 1D 62 00 00 00 00 14 3B 07 00 00 00   ..A..b.....;....
    0060  00 00 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D   ................
    0070  1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D   .. !"#$%&'()*+,-
    0080  2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D   ./0123456789:;<=
    0090  3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D   >?@ABCDEFGHIJKLM
    00a0  4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D   NOPQRSTUVWXYZ[\]
    00b0  5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D   ^_`abcdefghijklm
    00c0  6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D   nopqrstuvwxyz{|}
    00d0  7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D   ~...............
    00e0  8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D   ................
    00f0  9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD   ................
    0100  AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD   ................
    0110  BE BF C0 C1 C2 C3 C4 C5 C6 C7                     ..........
```

As one may notice, in this case the size of the `data` field of the ICMPv6 Echo Request is expanded in order to make the total package size larger. In this way, when the query arrives at the core it has to be first compressed and then fragmented so that it conforms to the MTU accepted by the device.

The following fragmentation rules are then added to the SoR and can be applied to that traffic :

Listing 4.1 – Fragmentation Rules in rule icmp2.json

```
          },{
                 "RuleID" : 12,
                 "RuleIDLength" : 11,
                 "Fragmentation" : {
                         "FRMode": "NoAck",
                         "FRDirection": "DW"
          }
          },{
                 "RuleID" : 13,
                 "RuleIDLength" : 11,
                 "Fragmentation" : {
                 "FRMode": "NoAck",
                 "FRDirection": "UP"
                 }
     }
]
}
```

4.1 Understanding fragmentation rules

In addition to the rule ID and the rule ID length a fragmentation rule contains two parameters : the mode and the direction.

In order to support reliability, variable L2 MTUs and unidirectional links, the RFC 8724 defines three different fragmentation modes : (i) No-Ack, designed for limited and variable MTU sizes under the assumption that there is no out-of-sequence delivery, (ii) Ack-on-Error for variable MTU and out-of-order delivery using sporadic ACK messages and (iii) Ack-allways for invariable MTUs and no out-of-sequence delivery. In the following we will go deeper into details of these three modes.

As for the direction, it is necessary to stay the behaviour of the SCHC action based on where the traffic is originated. In our example, we will use the rule `12/11` for fragmentation on Downlink and rule `13/11` for fragmentation on Uplink. Therefore, if the traffic goes from the core to the device (Downlink), we use rule `12/11` for fragmenting the traffic at the core side and reassembling at the device side. On the contrary, rule `13/11` is used for fragmenting the traffic going from the device to the co and for the reassembly processes at the core side.

As stated in RFC 8724, in OpenSCHC, SCHC Fragmentation is always done after compression [1]. Then, as shown in Figure 4.1 the whole process goes as following :

1. It shall rather be noted that, the no-compression rule is also permitted. Therefore, if one is willing to use only SCHC Fragmentation, two rules should be defined : (i) no-compression rule and (ii) the desired fragmentation rule.

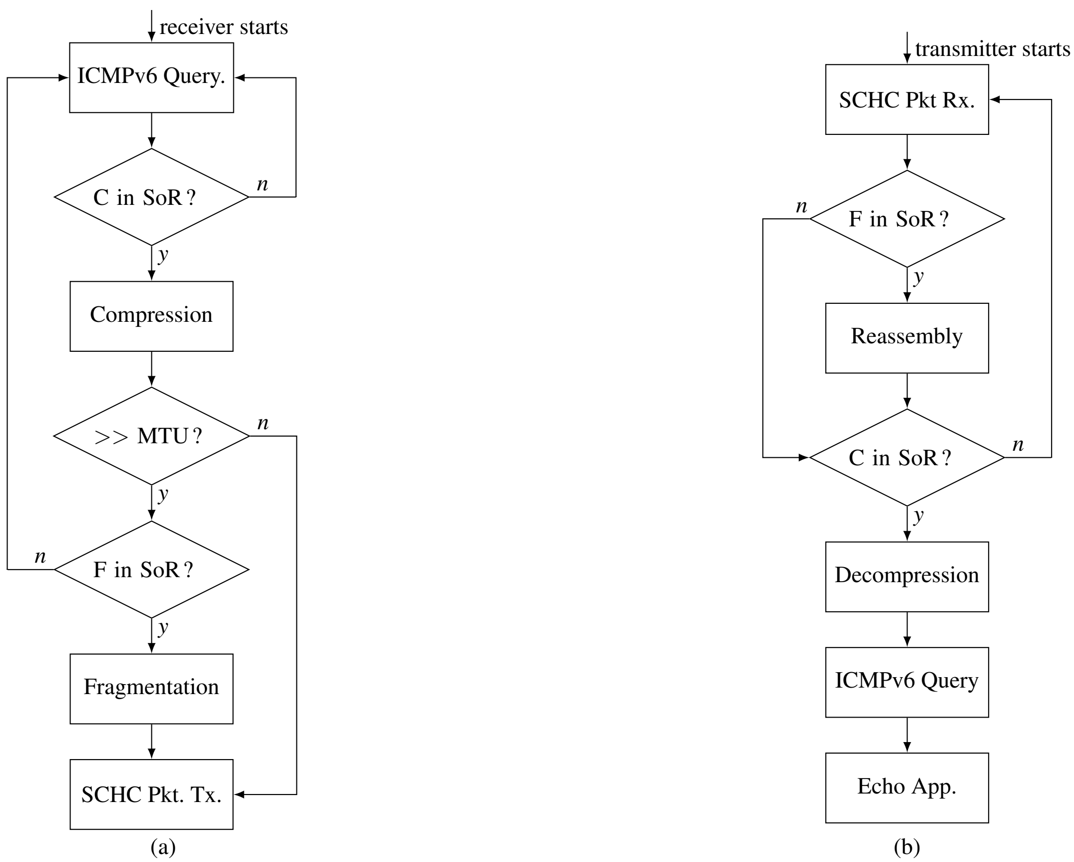

FIGURE 4.1 – ICMPv6 Query Reception when MTU exceeds L2 MTU : (a) Receiver behaviour, (b) Transmitter behaviour.

— ICMPv6 Echo Request Query Reception at core (Figure.4.1(a))
  1. An user send a ICMPv6 Echo request exceeding the maximum MTU allowed.
  2. The request arrives at the core side. It verifies if there is a compression rule on its SoR applicable to this specific traffic.
  3. If it exists, the core applies the compression rule. Then, it verifies if the resulting packet surpasses the L2 MTU size.
  4. If yes, the core verifies if there is a fragmentation rule on dowlink.
  5. If there is one, the core applies the fragmentation using the mode defined in the rule.
  6. The core starts to send fragments to the device.

— ICMPv6 Echo Request Query Reception at device (Figure.**??**(b)) :
  1. The device receives SCHC packets containing the fragments.
  2. The device verifies if there is a fragmentation rule and start the reassembly process.
  3. Once the reassembly process is finished, the device search for a compression ruls applicable to this specific traffic.
  4. The device applies the compression rule to decompress the packet.
  5. The device retrieves the ICMPv6 Echo Reply Query.
  6. At the application level, the device creates a ICMPv6 Echo Reply.

— ICMPv6 Echo Reply Query Transmission at device (Figure.**??**(a)) :
  1. The device verifies if there is a Compression rule for the ICMPv6 Echo Reply.
  2. If yes, the device compress the packet.
  3. The device verifies if the resulting packet surpasses the L2 MTU size.
  4. If yes, it looks for a Fragmentation rule in its SoR applicable to this traffic.
  5. If there is one, the device applies the fragmentation using the mode as defined in the rule.
  6. The device start to send fragments to the device.

— ICMPv6 Echo Reply Query Reception at core (Figure.**??**(b)) :
  1. The core receives SCHC packets containing the fragments.
  2. The core verifies if there is fragmentation rule in its SoR, if yes, it starts the reassembly process.
  3. Once finished, the core search for compression rules applicable to this specific traffic
  4. If there is a compression rue, it decompress the packet.
  5. The core forwards the ICMPv6 Echo Reply to the user.

4.2 Fragmentation in No-ACK mode

This fragmentation mode is designed for no out-of sequence delivery and admits variable L2 MTU. In No-ACK mode, there is no communication from the fragment receiver to the fragment sender. The sender transmits all the SCHC Fragments without expecting any acknowledgement. Therefore, there is no need for bidirectional links.

4.2.1 SCHC Fragments Format

In No-ACK, there are two kinds of SCHC Fragments : (i) `All-0` fragments presented in Figure.4.2, and the last fragment called `All-1` depicted in Figure.4.3 An all-0 fragment, is composed of the following fields [2] :

2. The RFC 8724 also defines the `DTag` (Datagram Tag) and the `W` (Window) fields. The former is used for differentiating SCHC F/R messages belonging to different SCHC Packets, for our example this field is not present, and the latter, representing the Window size used in Ack-on-Error and Ack-Allways modes

— `RuleID`
— `FCN` : It is used to differentiate All-0 and All-1 fragments.
— `Fragment Payload` : Corresponds to the payload. Its size is aligned to the remaining space from to fit the L2 MTU.

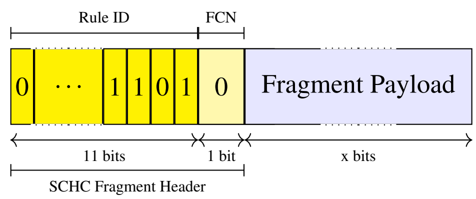

FIGURE 4.2 – All-0 SCHC Fragment in No-Ack mode

The second type of fragment is the `All-1`, it corresponds to the last fragment. As shown in Figure **??**, contrary to the `All-0`, it also contains the RCS field, and it can also contains padding as needed in order to fit the L2 word size.

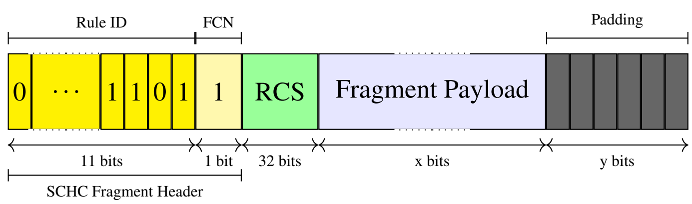

FIGURE 4.3 – All-1 SCHC Fragment in No-Ack mode

In OpenSCHC the Reassembly Check Sequence (RCS) field corresponds to the result of using the CRC32 algorithm, and as recommended by the RFC 8724 it is computed on the full SCHC packet (after reassembly) concatenated with the padding bits.

4.2.2 Fragmentation/Reassembly Process

In this mode, since there are no fragment acknowledgments, the sender creates as many fragments as needed based on the size of the compressed SCHC packet and the L2 MTU. Figure 4.4 presents an example where n fragments are needed. In this case, the transmitter creates $n-1$ `All-0` fragments and one `All-1` with the corresponding RCS field and padding if needed.

4.2.3 The code

In this section, we will present the code necessary to process an ICMPv6 echo request that exceeds the L2 MTU size. Two blocks of code are required : core and device programs. For the downlink, they are in charge of :

— core program : receiving the IPv6 packet, compressing it, fragmenting it and sending the SCHC fragments.
— device program : reassembling and decompressing SCHC packets and, creating the echo reply packet.

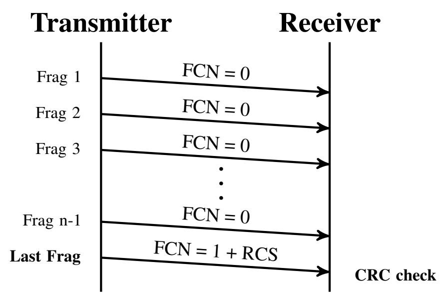

FIGURE 4.4 – All-1 SCHC Fragment in No-Ack mode

For the uplink, they are in charge of :

— device program : compressing and fragmenting the IPv6 Echo Reply packet, and sending the SCHC fragments to the core.
— core program : reassembling and decompressing SCHC packets and, forwarding the Echo reply IPv6 Packet to the user.

Core program

For this example we will extend the code used in Section 3.4.1. As in the previous case the Compression and Fragmentation process starts with the creation of a Rule Manager «`rm`» used to add the rules from the file `icmp2.json` (cf. 4.1 on page 27), which includes the Rule 3 for compressing and Rules 12 and 13 for fragmenting.

As presented in Listing **??** on page ??, the process follows the same logic as in the compression example. The `processPkt` function filters SCHC packets coming from devices and IPv6 tunneled packets from Hurricane Electric.

As in the C/D only example, the `processPkt` calls the SCHC Machine and then looks at the received packets. There can be two kinds off packets : (i) IPv6 tunneled packets filtered by looking at the IP proto 41 and passed to the SCHC Machine removing the first 34 bytes (Ethernet and IPv4 headers), and (ii) SCHC Packets filtered by looking at the UDP socket port `0x5C4C`.

Listing 4.2 – ping_core2.py

```
import sys
# insert at 1, 0 is the script path (or '' in REPL)
sys.path.insert(1, '../../src/')

from scapy.all import *
import gen_rulemanager as RM
from protocol import SCHCProtocol
from scapy_connection import *
from gen_utils import dprint, sanitize_value
from scapy.layers.inet import IP
import pprint
import binascii
import socket
import ipaddress

```

```
# Create a Rule Manager and upload the rules.
rm = RM.RuleManager()
rm.Add(file="icmp2.json")
rm.Print()

def processPkt(pkt):
    """ called when scapy receives a packet, since this function takes only one argument,
    schc_machine and scheduler must be specified as a global variable.
    """
    scheduler.run(session=schc_machine)
    # look for a tunneled SCHC pkt
    if pkt.getlayer(Ether) != None: #HE tunnel do not have Ethernet
        e_type = pkt.getlayer(Ether).type
        if e_type == 0x0800:
            ip_proto = pkt.getlayer(IP).proto
            if ip_proto == 17:
                udp_dport = pkt.getlayer(UDP).dport
                if udp_dport == socket_port: # tunnel SCHC msg to be decompressed
                    print ("tunneled␣SCHC␣msg")
                    schc_pkt, addr = tunnel.recvfrom(2000)
                    other_end = "udp:"+addr[0]+":"+str(addr[1])
                    print("other␣end␣=", other_end)
                    uncomp_pkt = schc_machine.schc_recv(device_id=other_end,
                                          schc_packet=schc_pkt)
                    if uncomp_pkt != None:
                        uncomp_pkt[1].show()
                        send(uncomp_pkt[1], iface="he-ipv6")
            elif ip_proto==41:
                schc_machine.schc_send(bytes(pkt)[34:])

# Start SCHC Machine
POSITION = T_POSITION_CORE

socket_port = 0x5C4C
tunnel = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tunnel.bind(("0.0.0.0", socket_port))

lower_layer = ScapyLowerLayer(position=POSITION, socket=tunnel, other_end=None)
system = ScapySystem()
scheduler = system.get_scheduler()
schc_machine = SCHCProtocol(
    system=system,            # define the scheduler
    layer2=lower_layer,       # how to send messages
    role=POSITION,            # DEVICE or CORE
    verbose = True)
schc_machine.set_rulemanager(rm)

sniff(prn=processPkt, iface="ens3") # scappy cannot read multiple interfaces
```

5. Answers to the questions

Index