D B3.2 – Design of models for IPv4/IPv6 modules

Pascal Moniot
STmicroelectronics
Rue Jules Horovitz
Cidex 19 Grenoble Cedex
pascal.moniot@st.com
The aim of this document is to describe ST Microelectronics’ Internet protocol hardware accelerators models (the NDCL) and associated Firewall software development done in MUSE project. Such modules can be integrated as throughput accelerator in future residential gateways.
TABLE OF CONTENTS

1 INTRODUCTION ......................................................................................................................8

2 SYSTEM VIEW ..........................................................................................................................9
  2.1 The Architecture ..................................................................................................................9

3 NDCL .......................................................................................................................................10
  3.1 Definitions ...........................................................................................................................10
  3.1.1 Multi-dimensional classification ....................................................................................10
  3.1.2 Basic NDCL operations ...................................................................................................12
  3.1.3 Protocol and layer handlers ............................................................................................13
  3.1.4 Ruleset trees ....................................................................................................................13
  3.1.5 Aggregation of 1DCL results ...........................................................................................14
  3.2 NDCL functional overview .................................................................................................16
  3.2.1 Configuration ..................................................................................................................16
  3.2.2 Operating modes .............................................................................................................20
  3.3 Overall architecture .............................................................................................................20
  3.3.1 Interfaces ........................................................................................................................20
  3.3.2 Top view ..........................................................................................................................24
  3.3.3 LOCK_MGT ...................................................................................................................26
  3.3.4 TOPOLOGY ......................................................................................................................28
  3.3.5 TRANSITION ..................................................................................................................34
  3.3.6 RULESET_RESOLVER ..................................................................................................40
  3.3.7 CLASSIFIERS ..................................................................................................................44
  3.4 Software configuration ........................................................................................................47
  3.4.1 Getting NDCL block information ....................................................................................47
  3.4.2 General configuration .......................................................................................................47
  3.5 Interface ...............................................................................................................................47

4 PACKET LOADER ..................................................................................................................47
  4.1 PL operations .........................................................................................................................47
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 PL top view</td>
<td>47</td>
</tr>
<tr>
<td>4.2.1 architecture</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2 Cache area</td>
<td>47</td>
</tr>
<tr>
<td>4.2.3 PDMEMS</td>
<td>47</td>
</tr>
<tr>
<td>4.2.4 PKT_MGR</td>
<td>47</td>
</tr>
<tr>
<td>4.2.5 RR_ELECTOR</td>
<td>47</td>
</tr>
<tr>
<td>4.2.6 DECODER</td>
<td>47</td>
</tr>
<tr>
<td>4.2.7 BUS_INTF</td>
<td>47</td>
</tr>
<tr>
<td>4.2.8 Endianess</td>
<td>47</td>
</tr>
<tr>
<td>4.3 Using PL block</td>
<td>47</td>
</tr>
<tr>
<td>4.3.1 packet storage</td>
<td>47</td>
</tr>
<tr>
<td>4.3.2 Reading data</td>
<td>47</td>
</tr>
<tr>
<td>4.3.3 Writing data</td>
<td>47</td>
</tr>
<tr>
<td>4.3.4 Reading a PDMEM</td>
<td>47</td>
</tr>
<tr>
<td>4.3.5 Updating a PDMEM</td>
<td>47</td>
</tr>
<tr>
<td>4.4 PL API</td>
<td>47</td>
</tr>
<tr>
<td>4.4.1 PL_init</td>
<td>47</td>
</tr>
<tr>
<td>4.4.2 PL_release</td>
<td>47</td>
</tr>
<tr>
<td>4.4.3 PL_remove</td>
<td>47</td>
</tr>
<tr>
<td>4.4.4 PL_flush</td>
<td>47</td>
</tr>
<tr>
<td>4.4.5 PL_get_spec</td>
<td>47</td>
</tr>
<tr>
<td>4.4.6 PL_store</td>
<td>47</td>
</tr>
<tr>
<td>4.4.7 PL_read</td>
<td>47</td>
</tr>
<tr>
<td>4.4.8 PL_write</td>
<td>47</td>
</tr>
<tr>
<td>4.4.9 PL_read_data</td>
<td>47</td>
</tr>
<tr>
<td>4.4.10 PL_write_data</td>
<td>47</td>
</tr>
<tr>
<td>4.5 PL register map</td>
<td>47</td>
</tr>
<tr>
<td>5 THE FIREWALL</td>
<td>47</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>47</td>
</tr>
<tr>
<td>5.1.1 The case of Linux IPv4 protocol stack</td>
<td>47</td>
</tr>
<tr>
<td>5.2 Firewall Architecture</td>
<td>47</td>
</tr>
<tr>
<td>5.2.1 Functional Blocks</td>
<td>47</td>
</tr>
<tr>
<td>5.2.2 Functional overview</td>
<td>47</td>
</tr>
<tr>
<td>5.3 Firewall Security Policy Database (FSPD)</td>
<td>47</td>
</tr>
<tr>
<td>5.3.1 Rules and Ruleset</td>
<td>47</td>
</tr>
<tr>
<td>5.3.2 Physical interfaces security parameters</td>
<td>47</td>
</tr>
<tr>
<td>5.4 Firewall Configuration Tool (FCT)</td>
<td>47</td>
</tr>
<tr>
<td>5.4.1 Managing NetObjx, ServiceObjs and Interfaces</td>
<td>47</td>
</tr>
<tr>
<td>5.4.2 Rules and rulesets management</td>
<td>47</td>
</tr>
<tr>
<td>5.4.3 Translating and installing rulesets into NDCL</td>
<td>47</td>
</tr>
<tr>
<td>5.5 Security Policy Parser (SPP)</td>
<td>47</td>
</tr>
<tr>
<td>5.6 Action Sub-system (AS)</td>
<td>47</td>
</tr>
<tr>
<td>5.7 Log Sub-system (LS)</td>
<td>47</td>
</tr>
<tr>
<td>5.8 Dynamic State Table (DST)</td>
<td>47</td>
</tr>
<tr>
<td>5.9 Connection Tracking Engines (CTE)</td>
<td>47</td>
</tr>
<tr>
<td>5.9.1 UDP-CTE</td>
<td>47</td>
</tr>
<tr>
<td>5.9.2 TCP-CTE</td>
<td>47</td>
</tr>
<tr>
<td>5.10 IP Fragmentation</td>
<td>47</td>
</tr>
<tr>
<td>6 INTEGRATION</td>
<td>47</td>
</tr>
<tr>
<td>6.1 Packet Reception</td>
<td>47</td>
</tr>
<tr>
<td>6.1.1 Classification</td>
<td>47</td>
</tr>
<tr>
<td>6.1.2 Firewall</td>
<td>47</td>
</tr>
<tr>
<td>6.1.3 Execution step</td>
<td>47</td>
</tr>
<tr>
<td>6.2 Locally generated packets</td>
<td>47</td>
</tr>
</tbody>
</table>
LIST OF FIGURES AND TABLES

Figure 0-1: conventions in schematics ................................................................. 7
Figure 2.1: System Architecture demonstrated on Mediaref .............................. 9
Figure 3: NDCL input and output ........................................................................ 11
Figure 4: rulesets and constraints ....................................................................... 11
Figure 5: an illustration of NDCL usage ............................................................... 12
Figure 6: NDCL functional organization ............................................................... 13
Figure 7: an example of NDCL organization ....................................................... 13
Figure 8: an example of ruleset tree ................................................................... 14
Figure 9: ruleset table example .......................................................................... 14
Figure 10: an example of class assignment ......................................................... 15
Figure 11: I-R Tree ............................................................................................. 16
Figure 12: Basic layers ....................................................................................... 16
Figure 13: configuration example ....................................................................... 17
Figure 14: an example of class mask ................................................................... 18
Figure 3-13: sequential operating mode .............................................................. 20
Figure 16: interfaces ........................................................................................... 21
Figure 17: special lock requests ......................................................................... 22
Figure 3-16: input packet topology ..................................................................... 23
Figure 19: triggering classification ..................................................................... 24
Figure 20: top view ............................................................................................. 25
Figure 21: lock requests ..................................................................................... 26
Figure 22: TOPOLOGY and LHDL/PHDL- an example ....................................... 29
Figure 23: TOPOLOGY top view ........................................................................ 30
Figure 24: LHDL input registers ....................................................................... 31
Figure 3-23: LHDL .............................................................................................. 32
Figure 26: TRANSITION ...................................................................................... 35
Figure 27: TRANSITION example – instances ..................................................... 38
Figure 28: TRANSITION example – classes ....................................................... 39
Figure 29: TRANSITION example – rulesets ....................................................... 39
Figure 30: TRANSITION example – POLL operation, step 1 ............................ 39
Figure 31: TRANSITION example – POLL operation, step 2 ............................ 40
Figure 32: RULESET_RESOLVER .................................................................. 41
Figure 33: RULESETS instructions .................................................................. 42
Figure 34: an example of ruleset instructions ..................................................... 43
Figure 35: bunches and segments ..................................................................... 44
Figure 36: 1DCL interface .................................................................................. 45
Figure 37: 1DCL START/READY signals ........................................................... 46
Figure 38: empty I-R trees ................................................................................ 47
Figure 39: registers map ..................................................................................... 47
Figure 40: PL interface ...................................................................................... 47
Figure 41: functional overview .......................................................................... 47
Figure 42: cache area structures ........................................................................ 47
Figure 43: registers map ..................................................................................... 47
Figure 44: Linux IP Networking Stack ............................................................... 47
Figure 45: Protocol stack, NDCL and Firewall work-flow ................................ 47
ABBREVIATIONS

1DCL Mono-dimensional classification engine
Bunch Consecutive instructions in RULESETS memory, describing all the transitions starting from a node
Class An aggregation of instances
CLASSID Class identifier
CRLINK Class/Ruleset link
FW Firewall
I-R tree A ruleset tree, where each transition is represented by a class/result doublet
Instance A call to a mono-dimensional classifier for a specific purpose
INSTANCEID Instance identifier
LHDL Layer handler
NDCL Multi-dimensional classification engine
PHDL Protocol handler
Ruleset index NDCL result for a specific ruleset
Ruleset node One node in a ruleset tree
Ruleset transition One transition in the ruleset tree
Ruleset tree A representation of a ruleset as a tree
RULESETS A memory in NDCL, used to describe a ruleset tree
memory
TP Topology analyzer
CONVENTIONS

We use the following conventions for schematics.

- XXX A register called XXX
- XXX A memory called XXX
- XXX The functional block we currently represent
- XXX An external functional block, or a sub-block

- Read: a functional block reads the contents of pointed register or memory.
- Write: a functional block writes the contents of pointed register or memory.
- Trig: a functional block trigs the activity of pointed functional block, with pseudo-signal XXX.

Figure 0-1: conventions in schematics

In this document, we call “pseudo-signal” a piece of information exchanged between two blocks, generally used to trigger operations. By extension\(^1\), we may consider some pseudo-signals as complex sets of pseudo-signals, read and written according to a specific protocol guaranteeing synchronization integrity.

\(^1\) Because we are at functional description level
1 INTRODUCTION

The aim of this document is to describe ST Microelectronics’ Internet protocol hardware accelerators models (the NDCL) and associated Firewall software development done in MUSE project. Such modules can be integrated in future residential gateways as HW accelerator to improve the throughput.

STm is using hardware functional models to verify on a demonstrable environment the functionalities supported by a dedicated HW/SW module in a complex application (e.g. the gateway). This functional demonstration (the emulation) is not a real time demonstration (it processes a limited number of packets per second) meanwhile it is performing enough to allow the connection to a “true” network (using Ethernet in our case study). This step is needed before going into real design where the designer is simulating the functionality of an isolated module.

2 platforms where used for our demonstration:

- STm Mediaref (see [http://cmg.st.com](http://cmg.st.com)), with this platform, we are extracting relevant figures to help the final architect to decide on the HW architecture. This platform uses STm cores (ST20 and ST40) and PCI network interface card (Ethernet)
- UML (User Mode Linux), which allows us to easily duplicate the development environment for testing purpose.

The methodology applied for this development was to first profile a Linux protocol stack on both a PC and a Mediaref. Using the figures extracted from this profiling (provided in a separated document), we decided in a specific HW / SW partitioning to accelerate the processing of IP packet.

Resulting from the selected HW / SW partition, the following component was developed:

- **NDCL**: hardware multi-dimensional classifier. Its purpose is to process packet topology and to classify the packet based on rules defined by other software modules (i.e. router, firewall etc...)
- **PACKET LOADER**: Its role is to improve the system performances cashing packet information in PL local memory to speed-up packet access in SoC design. It includes an API used by SW to trigger classification and to FW to access to classification results.
- **FW**: software block. It filters packets coming from the network according to a set of rules. A dedicated FW is needed to take the best of the NDCL classification module to improve the system performance when used with the NDCL improving the overall packet processing.

Hardware emulated blocks have been designed using Simmk an STm dedicated tools emulating on hardware platform (ST40 / ST20 / ST200 in our case) the exact hardware constraints (e.g. finite state machine; clock synchronization; registers etc...).

The following chapters describe all this modules starting by a system view and followed by module specifications.
2 SYSTEM VIEW

2.1 The Architecture

This section describes the software architecture (of a Linux based implementation) and the interconnections between the different blocks.

It introduced HW / SW functional block description and there integration made with the protocol stack.

The Mediaref integration uses the ST40 running the software part of the application, while all hardware co-processors are emulated on the ST20. Using this partitioning simplifies future re-used of the development at the cost of the additional communication channel between ST40 and ST20 that is handled in the SVI module.

The following figure describes the integration and the block connectivity, block functional description follows.

![System Architecture](image)

**Figure 2.1: System Architecture demonstrated on Mediaref.**

**Note:** ST40 runs Linux version 2.4.17.

Like in a real application, hardware emulated blocks (NDCL and PL) are called using their software drivers through. The block driver allows an appropriate abstraction of the request with respect to the block complexity. The following repartition was decided for the platform:

- **Firewalling**
  - **Firewall**: a Linux kernel module. It is connected to Linux Protocol stack with two Netfilter hooks, NF_IP_LOCAL_OUT and NF_IP_PRE_ROUTING. For each IP packet, FW provides “accept; drop; reject;” verdict to Protocol Stack. If filtering of different than IPv4 packet is needed (ARP, IPv6 etc…), FW uses additional hook not implemented in this platform.
3 NDCL

The NDCL is the core of the all system; this chapter describes NDCL detailed implementation.

3.1 Definitions

This section introduces the problems solved by a NDCL block, and provides some basic definitions for understanding multi-dimensional classification. Those definitions will be used within this document.

3.1.1 Multi-dimensional classification

Purpose of a NDCL block is to analyze the contents of a packet, and return some identifiers representing some rules each packet belongs to. Rules are defined at application level, as explained in “3.2.1-Configuration”.

For the protocol stack point of view, a rule specifies how to behave for each packet. Several applications ought to associate rules to behaviors, for example:

- Routing: define packet's destination
- Security: recognize encrypted packets
- Firewalling: define filtering rules (accept, drop, etc…)
- QoS: associate packet to a class of service
- Etc…

- Firewall Configurator: It’s a user space process. Using HTTP server and client, it provides a user interface to setup firewall rules.

### Classification

- NDCL: the multi-dimensional hardware emulated classification engine block
- NDCL driver: A Linux kernel module used to access NDCL Hardware emulated block
- NDCL Configurator: A Linux kernel module that configures the NDCL. This operation is complex and needs real time processing to sustain packet classification throughput.
- Classification Manager: a Linux kernel module. It schedules the operations between all blocks and reduces the modifications on Linux Protocol Stack. To handle packet classification, classification manager is connected to Linux Protocol stack netif_rx() function and to Linux Netfilter hook (NF_IP_LOCAL_OUT).

### Packet storage

- Packet Loader: It is Packet Loader hardware emulated block.
- PL driver: A Linux kernel module uses to access PL hardware emulated block

### Block communication

- SVI: it is the software inter-connect driver; an abstraction layer providing platform connectivity. SVI implements a logical communication in the context of the emulation platform, but it maybe used to interface with any hardware bus (e.g. STBus; AMBA; etc…) if implemented.
Each application uses some specific fields to classify packets, which are not necessarily the same from one application to another. Applications represent rules as a set of **constraints** on fields’ values plus an **action**, executed when constraints match. A set of rules is called **ruleset**.

Next figure is an illustration of rules:

![Figure 3: rulesets and constraints](image)

For a NDCL point of view, we only care of the constraint part of each rule. So we call **ruleset** a set of constraints a NDCL block has to match when performing a classification. A NDCL never addresses actions associated to results: this issue is related to the way applications use returned ruleset indexes.
Actually, the meaning of returned indexes can be different. For example, we can have some ruleset indexes directly defining an action, or providing an indirection to some software specific additional process, etc… Mainly, this is due to the fact that returned indexes are programmable by software.

### 3.1.2 Basic NDCL operations

From the rulesets definition, we can deduce several major issues addressed by a NDCL:

- **Packet topology**: a NDCL must know:
  - Which protocol are included in each packet
  - Where each protocol header is located in each packet

- **Protocol specific classifiers (1DCL)**: many fields need some specific processing, such as lookup, hash or tree parsing. For that reason, one NDCL aggregates the results of some more specific 1DCL. Each 1DCL contains its own set of information (**context**) and configuration methods.

As a result, a multi-dimensional classifier performs three major functions:

- **One packet topology analysis**.
- **Several mono-dimensional classifications**, using integrated 1DCL sub-blocks.
- **One ruleset resolution**, responsible for synthesizing 1DCL results and producing rulesets indexes.
3.1.3 Protocol and layer handlers

In this functional specification, 1DCL sub-blocks are grouped according to the protocol they belong to, which are organized according to the layers they belong to. For example, we may have one NDCL block, divided as follows:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Protocol</th>
<th>1DCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>N/A</td>
<td>Physical interface</td>
</tr>
<tr>
<td>3</td>
<td>IPv4</td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOS</td>
</tr>
<tr>
<td></td>
<td>IPv6</td>
<td>Address</td>
</tr>
<tr>
<td>4</td>
<td>TCP</td>
<td>Port</td>
</tr>
<tr>
<td></td>
<td>UDP</td>
<td>Port</td>
</tr>
</tbody>
</table>

Figure 6: an example of NDCL organization

We call protocol handler (PHDL, for short) a group of 1DCL belonging to the same protocol (IPv4, TCP, UDP, etc...), plus the engine that triggers their activities. A PHDL also includes a topology analyzer (TP, for short), which aims at deducing next protocol and next protocol header location according to current header contents.

We call layer handler (LHDL, for short) a set of PHDL belonging to the same protocol stack layer.

3.1.4 Ruleset trees

On the basis of previous definitions, one ruleset can be represented as a tree (called ruleset tree), where each node (called ruleset node) represents one specific location in ruleset table, and each transition is a 1DCL result.

For example, let's consider above figure “NDCL functional organization”. We use two 1DCL, one for IP destination address (layer 3) and one for TCP destination port (layer 4).

Corresponding ruleset tree is shown hereafter:
Notice that a wildcard (\*) is required for every node. A wildcard represents any value that does not match other transitions.

In this representation, we organize ruleset trees by levels (one for IP destination address, one for TCP destination port in our example), called "levels of transitions". Intuitively, one level of transition corresponds to one column in a ruleset table.

### 3.1.5 Aggregation of 1DCL results

Practically, ruleset trees are more complex than previous representation, mainly because the same level of transition may correspond to several 1DCL. For example, let’s consider the following ruleset:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>IP source</th>
<th>IP destination</th>
<th>TCP source</th>
<th>TCP destination</th>
<th>UDP source</th>
<th>UDP destination</th>
<th>ICMP type</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>192.15.6.1</td>
<td>192.15.6.2</td>
<td>8080</td>
<td>8080</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>ICMP/IP</td>
<td>*</td>
<td>129.<em>.</em>.*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>UDP/IP</td>
<td>192.<em>.</em>.*</td>
<td>129.<em>.</em>.*</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>20</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>3</td>
</tr>
</tbody>
</table>

In this example, one 1DCL engine may be used for several purposes (for example, we may call the same TCP port classifier for classifying TCP source and TCP destination ports, using different arguments).

Consequently, we make a distinction between:
• A 1DCL block: an engine performing a mono-dimensional classification.
• A 1DCL instance: one call to a 1DCL block for a specific purpose.

Each 1DCL instance is associated to an instance identifier (instance for short). Each instance corresponds to an identifier, denoted INSTANCEID. An instance is unique in the NDCL, and is used to represent the levels of transitions in ruleset trees. We can handle previous ruleset with a NDCL block implementing the following 1DCL instances:

- An IP source address classifier, which is arbitrarily assigned number #0
- An IP destination address classifier, which is arbitrarily assigned number #1
- A TCP source port classifier, which is arbitrarily assigned number #2
- A TCP destination port classifier, which is arbitrarily assigned number #3
- A UDP source port classifier, which is arbitrarily assigned number #4
- A UDP destination port classifier, which is arbitrarily assigned number #5
- An ICMP type classifier, which is arbitrarily assigned number #6

Moreover, some 1DCL results may be used at the same level of transitions in a ruleset tree. For example, TCP and UDP source ports may belong to the same level. In other words, we can aggregate some results for some specific rulesets.

Previous ruleset table can be viewed as a ruleset tree if and only if we aggregate some 1DCL results altogether, considering that:

- Each aggregation corresponds to one level of transition in ruleset tree
- Two instances belonging to the same aggregation do not belong to the same protocol, but belong to the same layer
- 1DCL aggregations are semantically coherent (they represent some things that may be considered as having a similar meaning in ruleset trees, for example ‘source port’, ‘destination address”, etc… Actually, coherency is not precisely defined… It’s intuitive).

We define a class of results (class for short) an aggregation of instances. Each class corresponds to an identifier, denoted CLASSID.

In previous example, we can –for example- create the following classes:

<table>
<thead>
<tr>
<th>CLASSID</th>
<th>Corresponding 1DCL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0: IP source address</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>#1: IP destination address</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>#2: TCP source port</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#4: UDP source port</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#6: ICMP type classifier</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>#3: TCP destination port</td>
<td>First step in layer 4 analysis</td>
</tr>
<tr>
<td></td>
<td>#5: UDP destination port</td>
<td>Second step in layer 4 analysis</td>
</tr>
</tbody>
</table>

Figure 9: an example of class assignment

A possible ruleset tree for representing previous ruleset table is shown below. In this figure, transitions are represented as I-R, where I is the INSTANCEID, and R is classification result. We call such tree “I-R tree.”

---

2 Arbitrarily means here that it’s hardwired in block design.
Figure 10: I-R Tree

Important: it’s application responsibility to maintain the overall system coherency, in particular the results returned by 1DCL and classes assignment.

### 3.2 NDCL functional overview

This section summarizes the functions covered by a generic NDCL block, using previous section’s definitions.

#### 3.2.1 Configuration

Basically, a NDCL blocks implement $N^3$ 1DCL instances. The list of included sub-blocks defines a specific NDCL. 1DCL are organized in $L$ different layers. Every NDCL implementation supports at least 4 layers:

<table>
<thead>
<tr>
<th>Layer number</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Link$^3$</td>
</tr>
<tr>
<td>3</td>
<td>Network</td>
</tr>
<tr>
<td>4</td>
<td>Transport</td>
</tr>
<tr>
<td>7</td>
<td>Application, i.e. where NDCL block stops classifying</td>
</tr>
</tbody>
</table>

Figure 11: Basic layers

There may be additional layers (for example, one specific layer for RTP, or security tags, etc...). By convention, they are assigned number $+X$ (where $X$ is an integer). Naturally, assignment is implementation specific.

Each 1DCL is associated to one or several classes (up to $P^5$ classes), associated to $M$ different rulesets. Associations are programmable by software.

---

$^3$ $N$ is greater than 0 and is lower than 33.

$^4$ Link layer information is not classified, but a NDCL needs some inputs from link layer to start processing a packet.
From previous definitions we see that:

- A ruleset tree has a maximum depth of \( P \) (one rule can’t exceed \( P \) levels of transition).
- \( N \) is the maximum number of fields that can be used in a packet to express a rule.

By associating instances to classes and classes to rulesets, software defines the types of classifications to be performed. Consequently, a generic NDCL is not specialized for specific applications, and can be configured for many different purposes.

![Figure 12: configuration example](image)

As illustrated in previous figure:

- One instance can be assigned to several classes.
- One class is defined on the basis of one or several instances.
- One ruleset is also defined on the basis of one or several classes.

Configuration itself is obtained by filling some **class masks** and the **Class/Ruleset link (CRLINK) table**.

### 3.2.1.1 Class mask

A class mask is a 32 bits field, where each bit corresponds to a specific class. Each supported instance corresponds to one mask. For each mask:

- If bit \( X \) is asserted, then that instance belongs to class number \( X \)...
- Else, it does not.

---

\( ^5 \) \( P \) is greater than 0 and is lower than 33.
Example:
Let’s consider the example given in paragraph “3.1.5-Aggregation of 1DCL results”. Then we have the following class masks:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Mask (hexa)</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00000001</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>0x00000002</td>
<td>#1</td>
</tr>
<tr>
<td>2</td>
<td>0x00000004</td>
<td>#2</td>
</tr>
<tr>
<td>3</td>
<td>0x00000008</td>
<td>#3</td>
</tr>
<tr>
<td>4</td>
<td>0x00000004</td>
<td>#2</td>
</tr>
<tr>
<td>5</td>
<td>0x00000008</td>
<td>#3</td>
</tr>
<tr>
<td>6</td>
<td>0x00000004</td>
<td>#2</td>
</tr>
</tbody>
</table>

Figure 13: an example of class mask

In the NDCL, class masks are implemented in CLMASK registers.

3.2.1.2 Ruleset mask
Each of the M rulesets corresponds to an index between 0 and M-1 (the very first ruleset is ruleset 0, next is ruleset 1, etc…).
A ruleset mask is a 32 bits field, where each bit corresponds to a specific ruleset index. Each class corresponds to one ruleset mask. For each mask:
- If bit X is asserted, then that class participates the ruleset identified by index X.
- Else it does not.
In the NDCL, ruleset masks are implemented in RMASK registers.

3.2.1.3 CRLINK table
This table defines the list of classes used by every ruleset. This definition includes:
- The list of classes itself
- The order in which classes will have to be taken into account to solve the ruleset

Next table is an example of CRLINK table:

<table>
<thead>
<tr>
<th></th>
<th>R0</th>
<th>R1</th>
<th>...</th>
<th>RM-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>P-1</td>
<td>3</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>P-1</td>
<td>...</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>P-1</td>
<td>-1</td>
<td>-1</td>
<td>...</td>
<td>P-1</td>
</tr>
<tr>
<td>P</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 3-1: Example of CRLINK table

---

Notice that we have P+1 lines in the table. This line is mandatory, and must always be filled with -1.
In this representation:
- Each column represents a ruleset
- Each line represents a sequence number (i.e. an order of apparition of a class)
- Cells represent class identification. Class -1 is reserved for ‘no class’, while another value represents a class number (from 0 to P-1).

The NDCL interprets the CRLINK table as follows:
- For a given ruleset X:
  - The first class to expect a result from is given by CRLINK[0][X]
  - The second class to expect a result from is given by CRLINK[1][X]
  - And so on, until class -1 is found (meaning that subsequent class values different from -1 are not taken into account)

So ‘Table 3-1: Example of CRLINK table’ means that:
- To solve ruleset R0, the NDCL first waits for a result from class 0, and then a result from class P-1.
- To solve ruleset R1, the NDCL first waits for a result from class 2, then from class 3 and finally from class P-1.
- To solve ruleset RM-1, the NDCL first waits for a result from class 0, then 1, then 2… and finally class P-1.

The CRLINK table is represented by a set of CRLINK registers:
- CRLINK_0_0 to CRLINK_0_6 represent column 0 in CRLINK table (i.e. the information associated to ruleset 0).
- CRLINK_X_0 to CRLINK_X_6 represent column X in CRLINK table (i.e. the information associated to ruleset X).

Each set of registers implements a compact representation of the line, knowing that each cell requires 6 bits. For simplicity reason, one cell is never split on two registers. So we get the following representation:
- CRLINK_0_0 contains CRLINK[0][0], CRLINK[1][0], CRLINK[2][0], CRLINK[3][0], CRLINK[4][0]
- CRLINK_0_1 contains CRLINK[5][0], CRLINK[6][0], CRLINK[7][0], CRLINK[8][0], CRLINK[9][0]
- Etc…

Next formula shows how to guess the appropriate CRLINK register and offset in this register to map CRLINK[X][Y]:
\[ Z = \frac{X}{5}, \ W = X \mod 5 \Rightarrow CRLINK[X][Y] = CRLINK_{Y,Z}, \text{ at offset } 6^*W \]

3.2.1.4 I-R tree

Once class and ruleset masks are defined, external applications can create and modify their I-R trees, using a specific memory embedded in NDCL block. Memory structure is described in paragraph “3.3.6.2-RULESETS memory”.
3.2.2 Operating modes

External modules can request a NDCL block to perform:
- A complete classification process for a specified list of rulesets
- One specific 1DCL operation, as explained in above section “Aggregation of 1DCL results”

A NDCL always operate in **sequential mode**. It is analyzes one packet at a time. Topology analysis, 1DCL operations and rulesets resolutions are performed in parallel on the same packet, so that the packet analysis runs as fast as possible.

Next figure is an illustration of sequential mode:

![Sequential mode diagram](image)

**Figure 3-14: sequential operating mode**

3.3 Overall architecture

This section defines the overall architecture (and consequently interface) of a generic NDCL for a functional point of view. It helps understanding how a generic NDCL block works, and how it shall be configured.

3.3.1 Interfaces

Next top view introduces the interfaces of a NDCL block:
Figure 15: interfaces
In this figure:
- Network protocol stack is a hardware or software engine that triggers classifications operations
- Network configuration tools are software blocks responsible for setting up and modifying NDCL configuration
- Packet represents a classified packet

### 3.3.1.1 Lock system

Whichever the requestor (network protocol stack or network configuration tools) is, and whatever the operation is (starting a classification or configuring NDCL), it must first perform a lock request, to avoid conflicting with another requestor.

A NDCL is able to support up to **U lock requests** in parallel, indexed from 0 to U-1. Each lock request is associated to a pair of registers:
- **LOCK**: when an application writes into that register, it posts a lock request.
- **READY**: once lock request is posted, application must poll READY register until it equals 1. This register is a read/reset register, to avoid residual authorizations.

At system level, developers are responsible for assigning a unique lock request identifier to each external block, but one lock request identifier is reserved⁷:

<table>
<thead>
<tr>
<th>Lock request number</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Request for a new classification.</td>
</tr>
</tbody>
</table>

**Figure 16: special lock requests**

For every lock request, except request 0, user is responsible for unlocking the system when its operations are finished, by asserting the UNLOCK register. Since one request is locked at one specific moment, UNLOCK is global to every request (it unlocks the current request number).

### 3.3.1.2 Configuration

A NDCL provides the following configuration registers:
- **CLMASK0, CLMASK1... CLMASKN-1**: class masks (see “3.2.1.1-Class mask” for masks description).
- **RMASK0, RMASK1 ... RMASKP-1**: ruleset masks (see “3.2.1.2-Ruleset mask”).
- **CRLINK0_0... CRLINK0_6 – CRLINK1_0..., CRLINK_P-1_6**: Classes-Rulesets association. (see “3.2.1.3-CRLINK table” for registers description).
- **RINIT0, RINIT1... RINITM-1**: rulesets initial indexes, as described in “RULESETS memory”.
- **RULESETS**: rulesets tables as described in “RULESETS memory”.

**A CONFIGURATION APPLICATION CAN UPDATE THOSE REGISTERS AND MEMORY AT ANY MOMENT, TAKING CARE OF HAVING FIRST REQUESTED A LOCK. ONCE CONFIGURATION IS FINISHED, APPLICATION MUST WRITE INTO THE UNLOCK REGISTER, IN ORDER TO LET NEXT LOCK REQUEST BEING ALLOWED.**

---

⁷ So U is greater than 2.
3.3.1.3 Triggering a classification

As a preliminary to any new classification request, external processes must first lock request number 0.

A NDCL block sees a packet as follows:

A NDCL block needs the following information to start classifying a packet:

- TPMASK: a bitfield specifying the list of rulesets to be fetched:
  - If bit number X is asserted, then ruleset number X will be computed by NDCL block.
  - Else it will not.
- ADDR: classified packet’s identifier.
- OFFSET: offset of layer 3 header in classified packet.
- PHYNO: a physical interface identifier, specifying which interface packet was received on. PHYNO is the very first classified information.
- NXTPROT: layer 2 should provide a protocol identifier, specifying which layer 3 we are dealing with. For example, NXTPROT is the type field in an Ethernet/DIX header, or is supplied by a SNAP in a 802.2/LLC/SNAP encapsulation.

Setting the L2_START signal acts as a trigger for the NDCL.
When performing packet topology, the NDCL block updates the following registers:

- **ROFFSET3**: offset of layer 3 header in packet (which is, in fact OFFSET). **This offset MUST BE longword aligned** (the 2 LSBs are not taken to account).
- **ROFFSET4**: offset of layer 4 header in packet.
- **ROFFSET7**: offset of application layer header in packet.
- **ROFFSET+1, …**: offset of additional layer headers in packet.
- **RPROT3**: identified layer 3 protocol, or –1 if layer 3 protocol is unknown.
- **RPROT4**: identified layer 4 protocol, or –1 if layer 4 protocol is unknown.
- **RPROT7**: identified application protocol, or –1 if protocol is unknown.
- **RPROT+1…**: identifier additional layer header in packet.

An external process may re-use those fields in order to perform some specific operations on non-classified headers fields, without performing a new topology analysis.

When packet is classified, the NDCL block updates the following registers:

- **RSTATUS**: tells that classification operation is completed, and provides classification status.
- **RINDEX0, RINDEX1… RINDEXM-1**: resulting ruleset indexes.

RSTATUS register provides the following information to external blocks:

- **RSR_COMPLETED**: a response telling whether ruleset resolution is completed or not.
- **TOPO_STATUS**: provides a status on last topology/mono-dimensional classification (success or error).

Notice that **register is reset by NDCL every time a new classification starts**.

Next is a typical classification session for the application point of view:

<table>
<thead>
<tr>
<th>Action</th>
<th>NDCL answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock request 0</td>
<td>Assert READY0 when classification is allowed</td>
</tr>
<tr>
<td>Write TPMASK, ADDR, OFFSET, PHYNO and NXTPROT and assert L2_start</td>
<td>Start packet topology, mono-dimensional classifications and ruleset resolution.</td>
</tr>
<tr>
<td>Wait for COMPLETED signal to be asserted: this is Classification Completion signal.</td>
<td>Once classification is over, write all the results (RSTATUS register, Topology ROFFSETx and RPROTx, Rulesets RINDEX registers) into Packet Loader’s PDMEM and asserts COMPLETED signal.</td>
</tr>
<tr>
<td>Read back RSTATUS, ROFFSET3, ROFFSET4, ROFFSET7, ROFFSET+1… RPROT3, RPROT4, RPROT7, RPROT+1… RINDEX0, RINDEXI/… from the Packet Loader PDMEM registers to get topology and classification results.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 18: triggering classification

### 3.3.2 Top view

Next top view summarizes the functional sub-blocks implemented in a NDCL block (some read/write destinations are missing, to simplify schematics):
Figure 19: top view
Sub-block LOCK_MGT is managing lock requests.
Sub-block TOPOLOGY is in charge of:
- Analyzing packet topology
- Scheduling calls to 1DCL according to packet contents
Sub-block PL is an external or internal packet loader, compliant with specification described in bellowing chapter "packet Loader". There is no synchronization control to access PL. This issue is solved internally, due to the sequencing of accesses.
Sub-block CLASSIFIERS contains all the 1DCL implemented in NDCL block.
The TRANSITION sub-block retrieves the results provided by mono-dimensional classifiers and schedules ruleset resolution according to class and rulesets masks. It is also in charge of storing NDCL results into the external Packet Loader, using the appropriate protocol.

RULESET_RESOLVER is performing rulesets resolutions on TRANSITION sub-block requests.

### 3.3.3 LOCK_MGT

That sub-block parses lock requests posted by writing into LOCKI registers, and serves them following a round-robin scheme.
It uses a special “hidden” (at index U) lock request, set when a ruleset resolution is being performed.
Next table summarizes locks meanings:

<table>
<thead>
<tr>
<th>Lock</th>
<th>Asserted when</th>
<th>De-asserted when</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>User writes into LOCK0</td>
<td>Pseudo-signal UNLOCK0 is asserted</td>
<td>Classification request: sub-blocks TOPOLOGY and CLASSIFIERS are proceeding a topology analysis and some mono-dimensional classifications.</td>
</tr>
<tr>
<td>I &lt; U</td>
<td>User writes into LOCKI</td>
<td>User writes into UNLOCK register</td>
<td>Configuration process: an external application is modifying contexts, in order to update classification contexts.</td>
</tr>
<tr>
<td>U</td>
<td>TRANSITION STARTS RECEIVING MONO-DIMENSIONAL CLASSIFIERS RESULTS (IT IS AUTOMATICALLY ASSERTED AFTER A LOCK0 PHASE)</td>
<td>Pseudo-signal UNLOCKU is asserted</td>
<td>Ruleset resolution: sub-block RULESET_RESOLVER is currently resolving some rulesets.</td>
</tr>
</tbody>
</table>

*Figure 20: lock requests*
An external process can request for lock 0 by writing into LOCK0. LOCK_MGT performs the following actions:

- If a lock request is currently being served:
  - Store lock request 0 into internal registers, in order to serve that request later.
- If no lock request is currently being served:
  - Register lock request 0 as the current request served.
  - Write 1 into READY0, so that external process knows it can access NDCL block.

An external process can request for lock I by writing into LOCKI. LOCK_MGT performs the following actions:

- If another lock request is currently being served:
  - Store I into internal registers, in order to serve I later.
- Else:
  - Register lock request I as current request served.
  - Write 1 into READYI, so that external process knows it can modify NDCL configuration.

TRANSITION sub-block notifies of topology/mono-dimensional classification operations completion by asserting pseudo-signal UNLOCK0. In such a case, LOCK_MGT performs the following actions:

If lock number U is not asserted (so if UNLOCKU is already asserted):
Search for another lock request by reading internal registers. If such request I is found:
Register lock request I as current request served.
Write 1 into READYI, so that requesting process knows it can access NDCL.
Else, LOCK_MGT does not perform anything.

An external process notifies from a configuration completion by writing into UNLOCK register. In such a case, LOCK_MGT searches for the next pending lock request, by checking its internal registers. If such request if found, it serves it:
Register lock request as current request served.
Write 1 into corresponding READY register, so that requesting process knows it can access NDCL.

TRANSITION sub-block notifies from rulesets resolutions completion by asserting pseudo-signal UNLOCKU. LOCK_MGT behavior is similar to previous case.
Important: a NDCL block expects external block to respect the LOCK/READY/UNLOCK paradigm, described hereafter. Some unexpected behaviors may occur if protocol is not respected.

- For lock request 0:
  - Request for lock, by writing into LOCK0 register
  - Wait for READY0 to be asserted, and reset it when read
  - Update registers, in order to write classification request
  - Wait for operation completion, by checking COMPLETED signal, which is asserted when the multi-dimensional classification is over and the results have been written into the Packet Loader

- For lock request I:
  - Request for lock, by writing into LOCKI register
  - Wait for READYI to be asserted, and reset it when read
  - Update configuration
  - Notify configuration completion, by writing into UNLOCK register

Resetting policy: LOCK and UNLOCK registers are reset by the LOCK_MGT when read, while it is up to the external block to reset READY registers, once read.

3.3.4 TOPOLOGY

Topology sub-block covers two major operations in multi-dimensional classification:

- TP: topology resolution: guess which protocols a packet belongs to according to header specific fields values, and guess headers locations.
- SCHED: schedule mono-dimensional classifiers activities, according to identified protocols.

Next figure illustrates entities location in the overall layer/protocol layers organization:
Next figure is a functional overview of TOPOLOGY contents (signals to perform the Packet Loader protocol are omitted to lighten the picture):
Figure 22: TOPOLOGY top view

Note: START_TP2 pseudo-signal is TOPOLOGY trigger, asserted by the protocol stack (or the external application which triggers the classification); it MUST be asserted after writing into ADDR, OFFSET, PHYNO and NXTPROT registers.

In this top view, layer handlers LHDL/i include TP/SCHED blocks. Each layer forwards the following information to next layer:
- Protocol header offset in classified packet
- Next protocol identification, according to hardwired tables.

Next table shows which registers correspond to those values:
### Layer handlers

Each layer handler is a dispatcher to a protocol handler, as summarized hereafter:

<table>
<thead>
<tr>
<th>Layer handler</th>
<th>Protocol header offset</th>
<th>Current protocol identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHDL2(^8)</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>LHDL3</td>
<td>ROFFSET3</td>
<td>RPROT3</td>
</tr>
<tr>
<td>LHDL4</td>
<td>ROFFSET4</td>
<td>RPROT4</td>
</tr>
<tr>
<td>LHDL7</td>
<td>ROFFSET7</td>
<td>RPROT7</td>
</tr>
<tr>
<td>LHDL+X</td>
<td>ROFFSET+X</td>
<td>RPROT+X</td>
</tr>
</tbody>
</table>

For every layer I:
- LHDL/ selects the appropriate TP/SCHED sub-blocks to be activated according to next protocol value.
- TP/SCHED blocks know where to start reading header information according to protocol header offset.

Communication with TRANSITION sub-block is described in paragraph “3.3.5-TRANSITION”.
Communication with CLASSIFIERS sub-block is described in paragraph “3.3.7-CLASSIFIERS”.

#### 3.3.4.1 Layer handlers

Each layer handler is a dispatcher to a protocol handler, as summarized hereafter:

---

\(^8\) If layer handler exists

\(^9\) Protocol header offset and Protocol identification are meaningless for layer 2.
OFFSET and NXTPROT are forwarded to layer 3.
Figure 3-24: LHDL
In this figure, \( J \) is the layer following layer \( I \) (signals to perform the Packet Loader protocol are omitted to lighten the picture).

\text{DISPATCH\_IN} sub-block is a de-multiplexer, selecting the appropriate protocol handler to be called, according to protocol identification (\( \text{RPROTI} \)). It performs the following actions:

- If \( \text{RPROTI} \) value corresponds to a known protocol, trigger corresponding protocol handler, by asserting corresponding \text{START} signal.
- Else:
  - Send an \text{ABORT} signal to \text{TRANSITION} sub-block.
  - Asserts a \text{Layer I Error} signal which triggers Topology main block activity: then Topology raises Layer 3,4,…..7 \text{RESET} signals which trigger the \text{SCHED} part of all the protocol handlers:
    - every protocol handler, if some monodimensional classification is ongoing, set the appropriate \text{1DCL-RESET} signal, to avoid NDCL to wait for long classification when an error has occurred (this is also required for a correct synchronization). In this case the 1DCLs return 0xFF as \text{RESULT} and 0x00 as \text{STATUS} (as in a “no result match” condition)
  - Write –1 into \( \text{RPROT}_J \), in order to stop topology process (i.e.: we spread out the error to other topological blocks).

Rest of the block is a series of protocol handlers, described in next paragraph.

\text{LAYER HANDLER 2} is also responsible of forwarding the information that a new classification is beginning by asserting \text{START} signal, exchanged between \text{TOPOLOGY} and \text{TRANSITION}. 
3.3.4.2 Protocol handlers

A protocol handler implements a TP and a SCHED sub-blocks. Practically, those functional sub-blocks are mixed in a single sequence of simple instructions, consisting in:

- Loading information from packet header, using PL sub-block: header’s offset is found in ROFFSET_I, fields offsets are protocol specific well-known values\(^{10}\)
- Deduce packet’s topology\(^{11}\):
  - Location of next layer header (ROFFSET_J)
  - Next layer protocol identification (RPROT_J)
- Start next layer analysis, by asserting START_TP
- Call mono-dimensional classifiers
- Wait for mono-dimensional classifiers completion (this is reported by the reset of the START signal) and, optionally, re-use some 1DCL for other fields (if more instances belong to that 1DCL). It is protocol-handler specific to keep track of the 1DCLs activity.
- When all the instances associated to that protocol have finished, assert a signal that tells the TOPOLOGY block that the 1DCL process for that layer is over. When all the 1DCL instances have finished, TRANSITION assert the INSTDONE signal.
- As explained above, Protocol Handlers are responsible of asserting internal Layer Errors (if a Topological Error has occurred) and to react to Layer Reset Signals: in this case a Protocol Handler (in particular, its SCHED part) must reset the monodimensional classifiers that are classifying some of its fields.

Notice that there is no specific constraint in the order of operations: sequence shall be defined to optimize system performances. On the other hand, there is a global constraint concerning the synchronization of all layer handlers, due to the fact that protocol handlers access the PL using the same channel: **a protocol handler is not allowed to trigger next layer until it has loaded all the fields required to perform header analysis (topology and classification).**

3.3.4.3 Ruleset Resolution Completed

When TRANSITION block detects the completion of the Ruleset Resolution, if the mono-dimensional classification is over, it raises the RR_COMPLETED signal. When this occurs, TOPOLOGY resets the ongoing classifications by means of the RESET signals like described in [3.3.4.1], and asserts the INSTDONE signal, so TRANSITION can complete the classification operations and write the results into Packet Loader.

3.3.5 TRANSITION

Purpose of transition sub-block is to trigger ruleset resolutions according to TOPOLOGY results, taking into account:

- Class masks, associating instances to classes.
- Rulesets/Class links.
- TPMASK, which provides the list of rulesets to be resolved.

3.3.5.1 Functional overview

Next figure gives an overview of TRANSITION functionalities:

\(^{10}\) Notice that TP implementation is different for every protocol (it’s not a generic engine).

\(^{11}\) This is a protocol specific process, hardwired for each protocol.
Figure 25: TRANSITION
Some connections between SYNC0 and all the output registers are omitted to lighten the picture: as SYNC0 is responsible of writing into PL, it will read all topological and ruleset resolution output registers. Other signals and connections are not drawn, to enlight the picture, too.

TRANSITION sub-block synchronizes the activity between TOPOLOGY sub-block and RULESET_RESOLVER sub-block.

It synthesizes the mono-dimensional classification operations initiated by TOPOLOGY:

- Collects mono-dimensional classifiers’ results
- Initiates corresponding ruleset resolutions according to:
  - TMASK
  - Class masks registers CLMASK/
  - Ruleset mask registers RMASK/
  - Class/Ruleset link table CRLINK

3.3.5.2 **SYNC0**

**SUB-BLOCK SYNC0 HANDLES TOPOLOGY REQUESTS AND COLLECTS 1DCL RESULTS.**

When the TOPOLOGY ABORT signal is asserted:

- Unlock topology resolution, by asserting pseudo-signal UNLOCK0
- Unlock ruleset resolution, by asserting pseudo-signal UNLOCKU
- Report topology error into RSTATUS
- If the functionality is enabled, stores all output registers\(^{12}\) of NDCL into Packet Loader’s PDMEM memory, using the appropriate protocol.
- Report the completion of all the operations by asserting COMPLETED signal, which is a trigger for the NDCL classification manager.

**Note:** when a TOPOLOGY ABORT event occurs, it may happen that some of the rulesets have already been resolved, or even all of them, and in the latter case the RSR_COMPLETED flag in RSTATUS will be set. So an external application MUST check, first of all, the TOPO_STATUS field, and proceed in retrieving the results only if an error has not occurred. In case of a TOPO_ABORT, the rulesets results depend on the time the error occurred and on the number of rulesets to be soved, so they should not be trusted.\(^{13}\)

---

\(^{12}\) Or the needed subset of them. This is implementation specific. Note that this must be standardized as all the other blocks of the system should be able to correctly retrieve these informations.

\(^{13}\) Optionally, if supported by the implementation, the result of a ruleset based only on the physical interface can be trusted also in this case, for metering and security purposes.
When both the TOPOLOGY END and TOPOLOGY 1DCL END signals are asserted:
- Unlock topology resolution, by asserting pseudo-signal UNLOCK0
Note: the END signal is referred only to the topological analysis, the INSTDONE signal is asserted when all the instances have finished their job.

When the TOPOLOGY START signal is asserted:
- Initialize local data
- Reset RSTATUS register of NDCL
- Enables the reception of 1DCLs results (this is required because when triggering a mono-dimensional classification from an external block, SYNC0 is not responsible of collecting the results)

When RR_COMPLETED signal from SYNC1 is asserted
- Unlock ruleset resolution, by asserting pseudo-signal UNLOCKU
- Report Ruleset Resolution completion into RSTATUS register

When both Topological part and Ruleset Resolution part are completed:
- If the functionality is enabled, stores all output registers\(^{14}\) of NDCL into Packet Loader’s PDMEM memory, using the appropriate protocol.
- Report the completion of all the operations by asserting COMPLETED signal, which is a trigger for the NDCL classification manager.

When one 1DCL asserts one of its READY signals, meaning that instance I has finished:
- Collect the result
- Store the results into local registers InstValI
- Assert the local signal COLL_INST_I to trigger POLL activity
- De-assert the READY signal

3.3.5.3 POLL

Sub-block POLL is triggered when one instance has finished.
When COLL_INST_I is asserted, POLL:
- Retrieves for the given instance I the classes it is associated to (this information is stored into CLMASKI register)
- For each class C, retrieves the associated rulesets (this information is stored into RMASKC register)
- For each ruleset and each class, updates some local logic to keep track of the reception of that instance for the pair (Class, Ruleset)\(^{15}\), and forwards the result availability information to SYNC1 block by means of CLASSID, RULESETID registers and WRITE signal
- When this double loop ends, waits for another instance result, if any.

When all the instance results have been collected and forwarded to SYNC1, if the ruleset resolution is not completed (i.e. RR_COMPLETED signal is not asserted) POLL asserts the POLL_COMPLETED signal to tell SYNC1 that it won’t receive results any more.

3.3.5.4 SYNC1

Sub-block SYNC1 interfaces with RULESET_RESOLVER, to trigger ruleset resolution operations.

---
\(^{14}\) Or the needed subset of them. This is implementation specific. Note that this must be standardized as all the other blocks of the system should be able to correctly retrieve these informations.

\(^{15}\) A pair (Class_x, Ruleset_y) uniquely identifies an instance, that can obviously be linked to other pairs (Class_i, Ruleset_j).
SYNC1 waits for POLL requests to trigger ruleset resolution, by means of the WRITE signal. SYNC1 receives for every instance some results (more precisely, one class ‘C’, and one ruleset ‘R’) in an unexpected order. Each pair (C, R) identifies uniquely an instance (I), and its result value (V): these information are stored into local logic and into the InstVal registers. For each pair (C, R), SYNC1:

- For given ruleset R, it checks whether an entry CRLINK[X][R] exists or not, and if all classes in CRLINK[X][R] (X’ < X) were already received or not.
  
  o If this condition is verified, SYNC1:
    
    - Posts a ruleset resolution request to RULESET_RESOLVER, using the protocol described in paragraph “0-
      
      - RULESET_RESOLVER”.
    
    - Then it checks if next result (i.e. with a class C'' = CRLINK[X’'][R], X'' > X) was received or not, and loops back on that process
  
  o Else, SYNC1 does nothing; the results received and not yet used are already stored into a local registers, so that value will be handled later.

SYNC1 is also responsible of reporting the Ruleset Resolution completion to SYNC0 block using the RR_COMPLETED signal.

The RR_COMPLETED signal triggers also TOPOLOGY activity, as described in [3.3.4.3]. This is useful when the Ruleset Resolution for the incoming packet is quickly completed (i.e. default branches with exit codes in the firsts classes), before the completion of the mono-dimensional classification phase.

When the POLL_COMPLETED signal is asserted, SYNC1:

- If the ruleset resolution is not over and it is not waiting for some result from the Ruleset Resolver Block:
  
  o for every not yet resolved Rulesets R it writes the reset value of 0x0000FFFF (-1 on 16 bits) into the output RINDEX
  
  o asserts the RR_COMPLETED signal, so SYNC0 can conclude as well the multi-dimensional resolution.

The above described case can occur when the configuration creates a “hole” at some layer, so if a ruleset is waiting for results belonging to, for example, UDP protocol, but the currently classified packet is a TCP packet, that ruleset won’t be solved, and the TRANSITION block needs to truncate it.

3.3.5.5 An example of TRANSITION block behavior

This paragraph illustrates the functions solved by CMAPPER, SYNC0 and SYNC1.

We consider the following NDCL configuration (hardwired):

<table>
<thead>
<tr>
<th>Instance</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>Ipv4 source address</td>
</tr>
<tr>
<td>I1</td>
<td>Ipv4 destination address</td>
</tr>
<tr>
<td>I2</td>
<td>TCP source port</td>
</tr>
<tr>
<td>I3</td>
<td>UDP source port</td>
</tr>
<tr>
<td>I4</td>
<td>TCP destination port</td>
</tr>
<tr>
<td>I5</td>
<td>UDP destination port</td>
</tr>
</tbody>
</table>

Figure 26: TRANSITION example – instances
Instances are mapped to the following classes (set with class masks):

<table>
<thead>
<tr>
<th>Class</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>I0 = IPv4 source address</td>
</tr>
<tr>
<td>C1</td>
<td>I1 = IPv4 destination address</td>
</tr>
<tr>
<td>C2</td>
<td>I2 + I3 = TCP or UDP source port</td>
</tr>
<tr>
<td>C3</td>
<td>I4 + I5 = TCP or UDP destination port</td>
</tr>
</tbody>
</table>

Figure 27: TRANSITION example – classes

Classes are used by two rulesets (set with rulesets masks):

<table>
<thead>
<tr>
<th>Ruleset</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>C0 + C2 + C3 = IPv4 source + TCP/UDP source + TCP/UDP destination</td>
</tr>
<tr>
<td>R1</td>
<td>C1 + C3 = IPv4 destination + TCP/UDP destination</td>
</tr>
</tbody>
</table>

Figure 28: TRANSITION example – rulesets

Now, we consider that this NDCL is called with a TPMASK equal to R0 plus R1 (i.e. solve both R0 and R1).
Every time TRANSITION sub-block receives a request for collecting a 1DCL result, associated to an instance, it first uses the local precalculated maps to guess which classes are associated to that instance:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>C0</td>
</tr>
<tr>
<td>I1</td>
<td>C1</td>
</tr>
<tr>
<td>I2</td>
<td>C2</td>
</tr>
<tr>
<td>I3</td>
<td>C2</td>
</tr>
<tr>
<td>I4</td>
<td>C3</td>
</tr>
<tr>
<td>I5</td>
<td>C3</td>
</tr>
</tbody>
</table>

Figure 29: TRANSITION example – POLL operation, step 1

In parallel, SYNC0 tries to collect 1DCL result. Then the POLL sub-block, using the precalculated data, retrieves the rulesets associated to each class.
Then the received result is forwarded to SYNC1, on the shape of a (Class, Ruleset) pair.

Now, since I-R trees are hierarchy based on classes, the order 1DCL results are forwarded to RULESET_RESOLVER depends on current level of resolution in I-R trees. For example, for R1, we must first resolve ruleset by providing the result associated to C1 and then the result associated to C3.

This means that when receiving a 1DCL result, SYNC1 either keeps that result in memory (if previous classes are not solved) or trigger a ruleset resolution.

Next is an example of SYNC1 operation, based on a sequence of inputs:

**INPUT = I0** (Ipv4 source address)
- ACTION for R0 : Trigger ruleset resolution with current result.
- ACTION for R1 : none.

**INPUT = I4** (TCP destination port)
- ACTION for R0 : First wait for C2 result => store result into r_I4_R0.
- ACTION for R1 : First wait for C1 result => store result into r_I4_R1.

**INPUT = I2** (TCP source port)
- ACTION for R0 : Trigger ruleset resolution with current result.
  - Trigger ruleset resolution with r_I4_R0.
- ACTION for R1 : none.

**INPUT = I1** (Ipv4 destination address)
- ACTION for R0 : none.
- ACTION for R1 : Trigger ruleset resolution with current result.
  - Trigger ruleset resolution with R_I4_R1.

### 3.3.6 RULESET_RESOLVER

Purpose of RULESET_RESOLVER is to resolve one ruleset on TRANSITION sub-block request.
3.3.6.1 Functional description

Next figure is a functional view of that sub-block:

![Functional View](image)

Ruleset resolution is a step by step process: every time TRANSITION sub-block gets some new inputs from TOPOLOGY sub-block, it may request RULESET_RESOLVER to check that result in order to get new node in I-R tree. I-R trees are all stored in a single memory, called RULESETS.

To start a ruleset resolution step, TRANSITION sub-block:
- Writes classifier result into IDATA
- Writes the instance identifier that provided this result into INSTANCE
- Provides the ruleset identifier to be solved into RULESETID
- Asserts pseudo-signal SOLVE
- Waits for pseudo-signal OK to be asserted
- Check resolution result by reading STATUS. STATUS value can be:
  - CONTINUE: we have reached a new node in the I-R tree.
  - FINISH: we have reached a terminal node in the I-R tree. This means that ruleset was fetched (result is put into RINDEX).

Notice that this process is asynchronous with:
- RULESET block.
- Other TRANSITION sub-block activities.

In other words, topology analysis and transition work in parallel with rulesets resolutions.
Before starting any series of ruleset resolutions (i.e. at every new multi-dimensional
classification process), TRANSITION asserts the RESET pseudo-signal, so that
RULESET_RESOLVER resets every ROFFS/ register to RINIT/.
Ruleset resolution is performed by LOOKUP, using the following sets of registers:
- ROFFS/: location of current ruleset node for I-R tree ‘I’ in RULESETS memory. This
  register initially equals RINIT/.
- RINIT/: location of ruleset root for I-R tree ‘I’ in RULESETS memory. This is supplied
  by external process when writing ruleset trees.
- RINDEX/: result for ruleset number ‘I’.
Returned indexed are programmed by users when writing I-R trees, as explained below.

3.3.6.2 RULESETS memory
This memory contains all the I-R trees participating classification. It contains
RULESETS_SIZE\(^{16}\) instructions, describing how to parse I-R trees.

3.3.6.2.1 Instructions format
Each instruction has the following format:

<table>
<thead>
<tr>
<th>31-31</th>
<th>30-26</th>
<th>25-18</th>
<th>17-17</th>
<th>16-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>INSTANCE</td>
<td>VALUE</td>
<td>ACTION</td>
<td>ARGUMENT</td>
</tr>
</tbody>
</table>

Figure 32: RULESETS instructions

Each instruction is divided in a comparison part and an action part.
The comparison part contains a couple of values (INSTANCE/VALUE), which must be
compared to INSTANCE/IDATA inputs:
- If DEFAULT field is 0:
  - If INSTANCE/VALUE fields equal INSTANCE/IDATA inputs, then LOOKUP
    applies the action part.
  - If INSTANCE/VALUE fields differ from INSTANCE/IDATA inputs, then
    LOOKUP continues parsing memory, checking next instruction.
- If DEFAULT field is 1, LOOKUP applies the action part, whatever INSTANCE/IDATA
  inputs are (this is a wildcard branch in I-R tree; such branch is mandatory).

From previous description, we deduce that process ends in two cases:
- A DEFAULT field is found (corresponding to a * in a I-R tree)
- A couple INSTANCE/VALUE matches requested instance/value in memory.
The action part defines which action must be performed if comparison part matches. Field
ACTION can be JUMP (jump to another node in I-R tree) or EXIT (ruleset resolved).

\(^{16}\) Lower than 64k.
Corresponding operations are applied:
- If ACTION is JUMP, then LOOKUP:
  - Forces ROFFS/I to be ARGUMENT, so that next ruleset resolution will start at
    ROFFS/I location.
  - Sets STATUS to CONTINUE
  - Asserts the OK pseudo-signal
- If ACTION is EXIT, then LOOKUP:
  - Stores ARGUMENT into RINDEX/I.
  - Sets STATUS to FINISH.
  - Asserts the OK pseudo-signal.

From previous description, we deduce that transitions from one node to another are
consecutive instructions in RULESETS memory, terminated by a DEFAULT instruction
(for default action), as illustrated below:

![Diagram](image)

**Figure 33: an example of ruleset instructions**

We call “bunch” such series: a bunch defines all the transitions from one node in a I-R tree.

### 3.3.6.2.2 Segments

For optimization issues, RULESETS is divided in segments. Each segment is a series of
SEGMENT_SIZE consecutive instructions. Segments definitions follow two constraints:
- All the instructions within a single segment must belong to the same bunch\(^\text{17}\).
- The beginning of a bunch must be the beginning of a segment.
- If X and Y are two consecutive segments, and X does not contain some DEFAULT
  instruction, then X and Y instructions belong to the same bunch.

Next figure is an illustration of constraints, with SEGMENT_SIZE equaling 4:

\[^\text{17}\] So, some segments may contain some unused instructions.
<table>
<thead>
<tr>
<th>SEGMENT</th>
<th>Non-default instruction</th>
<th>Bunch 1</th>
<th>Non-default instruction</th>
<th>Enclosed in a single segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEGMENT 1</td>
<td>Non-default instruction</td>
<td>Bunch 2</td>
<td>Non-default instruction</td>
<td>Spread over several segments</td>
</tr>
<tr>
<td></td>
<td>Non-default instruction</td>
<td></td>
<td>Non-default instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-default instruction</td>
<td></td>
<td>Non-default instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-default instruction</td>
<td></td>
<td>Non-default instruction</td>
<td></td>
</tr>
<tr>
<td>SEGMENT 2</td>
<td>Non-default instruction</td>
<td></td>
<td>Default instruction</td>
<td>Unused memory zone</td>
</tr>
<tr>
<td></td>
<td>Default instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>SEGMENT 3</td>
<td>Non-default instruction</td>
<td>Bunch 3</td>
<td>Non-default instruction</td>
<td>Enclosed in a single segment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Default instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Unused memory zone</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Figure 34: bunches and segments

Implementation note: the bunch mechanism can't be implemented in a software simulation, since we cannot easily write an address decoder.

### 3.3.7 CLASSIFIERS

CLASSIFIERS sub-block contains all the 1DCL implemented by NDCL block. Classifiers behaviors and specificities are implementation specific. In particular, their configuration process does not follow a specific scheme. On the other hand, mono-dimensional classifiers have a similar interface for starting a classification operation:
Registers CL_INPUT0, CL_INPUT1… are classification inputs. There are as many registers as required by 1DCL\textsuperscript{18}.

Register ARGUMENT is a classifier specific register, which is used to specify some additional information to classification process\textsuperscript{19}. This register shall also be a trigger to start classification operation (writing into that register tells 1DCL to start classifying CL_INPUT0, CL_INPUT1…).

STATUS is a status register. Register contents are classifier specific, but bit 0 must be asserted when 1DCL block is performing a classification operation, and de-asserted when no classification operation is being performed.

RESULT register contains classification result. This should be a 8 bits value.

Above Section “RULESET_RESOLVER” explains how to manipulate those registers.

\textsuperscript{18} For example, an IPv4 address classifier needs one 32 bits register, while an IPv6 classifier needs 4 32 bits registers.

\textsuperscript{19} For example, we can imagine a TCP/UDP port classifier, where ARGUMENT specifies the type of port to be classified (TCP source, TCP destination, UDP source or UDP destination).
Mono-dimensional classifiers are accessed by the TOPOLOGY sub-block – for triggering classifications – and the TRANSITION sub-block – for getting classification result. Synchronization between sub-blocks is made using a set of two instance specific signals:

- **START**: used to trigger a classification and mark 1DCL as busy (i.e. no other classification shall be performed with that 1DCL when START is asserted).
- **READY**: asserted by the 1DCL sub-block to notify TRANSITION sub-block that a result is available.
- **RESET**: asserted by TOPOLOGY’s protocol handlers when a Topological error has been detected; in this way the NDCL can end quickly without waiting for useless monodimensional classifications. It is de-asserted by the very 1DCL when it has stopped its operations. In case of RESET the mono-dimansional classifier returns 0xFF as RESULT and 0x00 as STATUS.

Next table summarizes the 1DCL state according to signals value:

<table>
<thead>
<tr>
<th>START</th>
<th>READY</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Block is idle, waiting for a classification request.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Block is classifying</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Block has finished classification</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>This situation is transitional</td>
</tr>
</tbody>
</table>

Figure 36: 1DCL START/READY signals

The two signals are modified according to the following sequence:

- The TOPOLOGY sub-block waits for the START signal to be de-asserted
- To trigger a new classification, it asserts START, so that the 1DCL sub-block starts classifying
- When classification is completed, the 1DCL sub-block asserts the READY signal, and then waits for READY to be de-asserted
- Asserting READY triggers TOPOLOGY sub-block activity. The sub-block collects 1DCL result
- When result is collected, the TOPOLOGY sub-block de-asserts the READY signal
- When READY is de-asserted, the 1DCL sub-block de-asserts START, so that a new classification can be requested

**NOTE**: this synchronization model has a major constraint: two sub-blocks (i.e. protocol handler belonging to different layers) must not use the same 1DCL to trigger its activity. Practically, we consider that no implementation should use two protocols (on different layers) using the same 1DCL, which seems to be a reasonable constraint.

**Note**: for debugging or process specific purposes an external process may require 1DCL to start classification independently. In such a case, external block may use a lock request to use 1DCL sub-block. Refer to paragraph “Lock system” for a description of lock procedure. Collecting results is done polling STATUS register until bit 0 equals 0, and then read back the results; READY signal should be reset.

---

20 I.e.: for every instance, the 1DCL implements these signals.

21 Actually, the fact of using or not a lock to trigger 1DCL is application specific.
3.4 Software configuration

This section summarizes the operations that need to be performed by software in order to configure a NDCL block.

First step consists in getting information about the block, in order to determine which NDCL software deals with.

Then comes a general configuration step, to configure the block, so that it becomes operational.

Once block is active, software has to modify 1DCL and ruleset resolution configuration, when some entries are added or removed from rulesets.

3.4.1 Getting NDCL block information

A NDCL block provides four registers that let external processes recognizing which block it deals with:

- **ID register** contains two hardwired values:
  - VENDOR: identifies product vendor
  - PRODUCT: specifies that block is a NDCL block

- **PRODUCT_ID** register contains a hardwired value which identifying the NDCL implementation (i.e. every implementation of a NDCL block is identified with a unique PRODUCT_ID). Software shall be able to guess the list of implemented 1DCL blocks from that information.

- **PRODUCT_SIZE** specifies the following information:
  - N: number of 1DCL instances
  - P: number of classes
  - M: number of rulesets
  - U: number of lock requests
  - L: number of supported layers

- **RULESETS_SIZE** specifies:
  - The size of RULESETS memory
  - The size of segments in RULESETS memory

3.4.2 General configuration

General configuration stage consists in profiling NDCL behavior all along the session, by programming the following registers:

- CLMASK registers, to specify classes
- RMASK registers, to specify rulesets
- CRLINK registers, to specify the Class/Ruleset link table

Software **must** also create some empty I-R trees in RULESETS memory. Corresponding RINIT registers **must** point to those entries, as shown below:
### 3.5 Interface

This section defines the registers interface for a NDCL block.

"IS" stands for "Implementation Specific".

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Reset value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>0x00000000</td>
<td>–</td>
<td>Product identification</td>
</tr>
<tr>
<td>PRODUCT_ID</td>
<td>0x00000004</td>
<td>IS</td>
<td>NDCL identification</td>
</tr>
<tr>
<td>PRODUCT_SIZE</td>
<td>0x00000008</td>
<td>IS</td>
<td>NDCL parameters</td>
</tr>
<tr>
<td>RULESET_SIZE</td>
<td>0x0000000C</td>
<td>IS</td>
<td>Size of ruleset table</td>
</tr>
<tr>
<td>CLMASK0</td>
<td>0x00000014</td>
<td>0x00000000</td>
<td>Class mask</td>
</tr>
<tr>
<td>To</td>
<td>0x000000094</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLMASK31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Start Address</td>
<td>End Address</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RMASK0 To RMASK31</td>
<td>0x000000098</td>
<td>0x00000118</td>
<td>CRLink registers</td>
</tr>
<tr>
<td>CRLINK_0_0 To CRLINK_31_6</td>
<td>0x0000011C</td>
<td>0xFFFFFFF</td>
<td>CRLink registers</td>
</tr>
<tr>
<td>LOCK0 To LOCK15</td>
<td>0x000004A0</td>
<td>0x000004DC</td>
<td>Lock requests</td>
</tr>
<tr>
<td>READY0 To READY15</td>
<td>0x000004E0</td>
<td>0x0000051C</td>
<td>Ready notifications</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>0x00000520</td>
<td>0x00000520</td>
<td>Unlock request</td>
</tr>
<tr>
<td>TPMASK</td>
<td>0x00000524</td>
<td>0x00000524</td>
<td>List of rulesets to be matched</td>
</tr>
<tr>
<td>ADDR</td>
<td>0x00000528</td>
<td>0x00000528</td>
<td>Classified packet identifier, as previously registered in packet loader.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>0x0000052C</td>
<td>0x0000052C</td>
<td>Layer 3 header offset in packet</td>
</tr>
<tr>
<td>PHYNO</td>
<td>0x00000530</td>
<td>0x00000530</td>
<td>Physical interface identifier</td>
</tr>
<tr>
<td>NXTPROT</td>
<td>0x00000534</td>
<td>0x00000534</td>
<td>Layer 3 protocol identification</td>
</tr>
<tr>
<td>RSTATUS</td>
<td>0x00000538</td>
<td>0x00000538</td>
<td>Classification status</td>
</tr>
<tr>
<td>RINDEX0 To RINDEX31</td>
<td>0x0000053C</td>
<td>0x000005B8</td>
<td>Returned rulesets indexes.</td>
</tr>
<tr>
<td>ROFFSET3</td>
<td>0x000005BC</td>
<td>0x000005BC</td>
<td>Network layer header offset</td>
</tr>
<tr>
<td>ROFFSET4</td>
<td>0x000005C0</td>
<td>0x000005C0</td>
<td>Transport layer header offset</td>
</tr>
<tr>
<td>ROFFSET7</td>
<td>0x000005C4</td>
<td>0x000005C4</td>
<td>Last identified layer header offset</td>
</tr>
<tr>
<td>ROFFSET+1 To ROFFSET+12</td>
<td>0x000005C8</td>
<td>0x000005F4</td>
<td>Identified header offset for additional layers</td>
</tr>
<tr>
<td>RPROT3</td>
<td>0x000005F8</td>
<td>0x000005F8</td>
<td>Network layer protocol identifier</td>
</tr>
<tr>
<td>RPROT4</td>
<td>0x000005FC</td>
<td>0x000005FC</td>
<td>Transport layer protocol identifier</td>
</tr>
<tr>
<td>RPROT7</td>
<td>0x00000600</td>
<td>0x00000600</td>
<td>Last identified protocol identifier</td>
</tr>
<tr>
<td>RPROT+1 To RPROT+12</td>
<td>0x00000604</td>
<td>0x00000630</td>
<td>Additional layers protocol identifiers</td>
</tr>
</tbody>
</table>
4 PACKET LOADER

Packet Loader module provides a networking cache and a standard API to accelerate access to packet fields. To operate in a generic system and to provide the sufficient flexibility, packet loader implements here after described functionalities:

- Cache up to NPKT packets in parallel
- Services up to NCHAN requestors in parallel, each requestor using a dedicated channel.
- Provides the capability to read contents of a packet on a per longword basis from the cache area or from external memory (when the packet is not registered in the cache)
- Associates each packet with dedicated information stored in specialized memory (PDMEM 'packet data memory') and used by protocol stack to store classification results.

Note: Sizing packet loader hardware will depend from the application-aggregated throughput; this will be done when going to real design.

4.1 PL operations

Next figure defines PL operations:
The PL block handles two types of operations:
- Packet storage: initiates the management of a new packet.
- Data request: request for a specific packet field (a longword) in a packet. This is a per-channel request.

Note: if two different applications use the same PL channel, then an external arbitration engine must be implemented to prevent from concurrent accesses.

4.2 PL top view

4.2.1 architecture

This section describes PL block more in detail, next figure is a functional overview of a PL block:
A Round Robin Elector (RR_ELECTOR) is used to arbitrate memory accesses. Every time a request is completed, RR_ELECTOR selects the next request to be served, according to a round robin scheme.
In parallel, a packet manager (PKT_MGR) receives packet storage request, and stores packets information into a local descriptor (PKT_TBL).

PL does not guarantee coherency between requests and information stored into PKT_MGR, but assume that PL operations are atomic. In particular, the overall system is responsible for insuring that a request on a packet will not occur while corresponding packet descriptor is updated.

The **DECODER** sub-block handles the request. It analyzes OP, and performs corresponding operation (read or write data). **DECODER** is also responsible for defining where to find/update data:
- Either in external memory, through the bus,
- Or in a local cache area, if a cache is implemented.
To do so, it requests the packet manager, to get current packet information and state (i.e. which part of the packet is currently cached…).

Paragraph “**DECODER**” enters more into **DECODER** process details. Bus transactions are abstracted through a dedicated block, translating an internal protocol defining requests (the LOAD protocol) to bus operations. Translating and executing bus transactions is under **BUS_INTF** responsibility.

### 4.2.2 Cache area

A PL block may or may not implement a cache area, depending on system implementation. Cache area size (**CCSIZE**) is implementation specific, and depends on:
- Bus capabilities: to optimise burst transactions latency.
- Protocols headers size: to avoid performing more than one transaction to load a common header.

The cache maintains the following information:
- One memory containing loaded (and updated) data.
- One modification mask (**PKT_BOOL**) specifying which longwords were updated and which were not.
When a cache reading is performed, the requested longword is copied from cache to the DECODER register. If writing is requested, the longword in cache is updated and the corresponding bit in the PKT_BOOL register is set.

When cache is flushed (because another block has to be loaded in that same locations), PKT_BOOL bitwise register will tell if a copy-back has to be performed. This mechanism improves multiple memory write, allowing accessing the memory only once packet manipulation is completed.

### 4.2.3 PDMEMS

This memory storage area contains all the PDMEMS associated to packets. Its size is NPKT * PDMEM_SIZE longwords.

### 4.2.4 PKT_MGR

The packet manager maintains packets information. For each packet identified with its packet identifier (a reference number between 0 and NPKT-1), it keeps:

- The base address of packet in memory.
- The list of pieces of packets currently cached\(^{22}\), by means of a 32 bit register (CATALOGUE) working as bitwise.
- The address of packet’s data memory in PDMEMS, not required if each packet has a fixed number of registers for storing its context.

### 4.2.5 RR_ELECTOR

RR_ELECTOR receives the requests from external blocks and selects the request to served. RR_ELECTOR uses CHAN and SELECTION registers to proceed with the request allocation.

- SELECTION is a bitwise register. If bit $X$ is set, then channel $X$ has a pending request.

---

\(^{22}\) For some implementation, the PL block will keep the entire packet into cache area, for some other, it will load pieces of packets on user requests. It may be able to keep more than one piece of each packet in cache area. All this is implementation specific.
• CHAN keeps the served channel.

When RR_ELECTOR has selected a request to be served, it forwards it description to a request decoder (DECODER), through OP and DATA registers:
  • Copies OPX into OP.
  • Copies DATAX into DATA.
  • Asserts REQ signal to initiate DECODER activity.
  • Waits for OK signal to be asserted, meaning that request processing is completed.
  • Copies OP into OPX.
  • Copies DATA into DATAX.

### 4.2.6 DECODER

DECODER interprets user requests, and translates them into access for:
  • Packet if OPCODE is RD or WR, which according to PKT_TBL register, CATALOGUE field results in driving:
    o External BUS_INTF when the packet is not in the cache area
    o Internal CACHE_AREA.
  • PDMEMS if OPCODE is RDD or WRD.

The decoder flushes caches when:
  • A cache area is overwritten (i.e. load another piece of packet).
  • A packet storage request is initiated (flush old packet if required).

### 4.2.7 BUS_INTF

This is a bus/implementation specific functional sub-block.
It reads PKT_BOOL to understand if BUS registers contain data to be flushed from cache.

### 4.2.8 Endianess

PL Endianess is big endian, data are organized according to the scheme compliant with the usual endianess implemented in TCP/IP protocol stack and described hereafter:

<table>
<thead>
<tr>
<th>Longword</th>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01020304</td>
<td>0x01</td>
<td>0x02</td>
<td>0x03</td>
<td>0x04</td>
</tr>
</tbody>
</table>

### 4.3 Using PL block

This chapter defines how PL interacts with the system.

#### 4.3.1 packet storage

**THIS OPERATION PROCEED WITH:**
  • Load the contents of a packet into internal memory
  • Store information about a packet into PL, so that it is able to load and update parts of this packet (through a caching mechanism for example).

To initiate packet storage, applications must:
Verify that STORED is not asserted\(^{23}\).
Write packet's base address into address register.
Write packet identifier into PARAM register.
Assert STORE
Wait for STORED signal to be asserted.
De-assert STORED signal.

Packet address MUST BE long word aligned. Actually, if packet does not start at a longword-aligned offset, heading offset are considered as part of the packet (so user should take care not to modify them).
The packet identifier is an arbitrary identifier between 0 and NPKT-1. The overall system is responsible for assigning a packet identifier for each new packet.

### 4.3.2 Reading data

To read a specific longword from current packet, using channel \(X\), applications must:
- Verify that OK\(X\) is not asserted\(^{24}\).
- Write longword offset into OP\(X\), with an OPCODE set to RD, and the appropriate PKT_ID.
- Assert REQ\(X\).
- Wait for OK\(X\) to be asserted.
- Read longword value in DATA\(X\).
- De-assert OK\(X\).

Some specific implementation may not support read operations\(^{25}\). In such a case, OP\(X\) returns OPCODE ERR when application tries to perform a read.

### 4.3.3 Writing data

To write a specific longword into current packet, using channel \(X\), application must:
- Verify that OK\(X\) is not asserted\(^{26}\).
- Write new longword value into DATA\(X\).
- Write longword offset into OP\(X\), with OPCODE set to WR, and the appropriate PKT_ID.
- Assert REQ\(X\).
- Wait for OK\(X\) to be asserted.
- Poll OP\(X\), until OPCODE becomes ACK.
- De-assert OK\(X\).

Some specific implementation may not support write operations\(^{27}\). In such a case, OP\(X\) returns OPCODE ERR when application tries to perform a write.

---

\(^{23}\) This is optional, depending on how many external blocks ought to initiate packet storage.

\(^{24}\) If one and only one requestor uses channel \(X\), then this operation is not required.

\(^{25}\) For example, if PL is a sub-block of a co-processor that never needs to read packet contents.

\(^{26}\) If one and only one requestor uses channel \(X\), then this operation is not required.

\(^{27}\) For example, if PL is a sub-block of a co-processor that never needs to update packet contents.
4.3.4 Reading a PDMEM

To read a specific longword from current packet’s PDMEM, using channel $X$, applications must:

- Verify that OKX is not asserted\(^{28}\).
- Write longword offset into OPX, with an OPCODE set to RDD, and the appropriate PKT_ID.
- Assert REQX.
- Wait for OKX to be asserted.
- Read longword value in DATAX.
- De-assert OKX.

4.3.5 Updating a PDMEM

To write a specific longword into current packet’s PDMEM, using channel $X$, applications must:

- Verify that OKX is not asserted\(^{29}\).
- Write new longword value into DATAX.
- Write longword offset into OPX, with OPCODE set to WRD, and the appropriate PKT_ID.
- Assert REQX.
- Wait for OKX to be asserted.
- Poll OPX, until OPCODE becomes ACK.
- De-assert OKX.

4.4 PL API

This chapter specifies the software API provided above PL Driver.

4.4.1 PL_init

**Prototype:** `int PL_init (void)`

**Description:**
This function initializes the PL block. Function should be called by the system during startup, to get an operational PL block.

**Arguments:** `void`

**Returns:** `int`

- 0: block successfully initialized.
- Else: an error occurred.

4.4.2 PL_release

**Prototype:** `void PL_release (void)`

---

\(^{28}\) If one and only one requestor uses channel $X$, then this operation is not required.

\(^{29}\) If one and only one requestor uses channel $X$, then this operation is not required.
This function stops PL block execution, and releases the possible software resources used to
driver/emulate PL block.

**Arguments:** void

**Returns:** void

### 4.4.3 PL_remove

**Prototype:** void PL_remove (pkt_id)

**Description**
This function removes a packet from the list of packets to be processed. This means that the
pkt_id used to refer to it is now available for another packet. This also implies that its context
(PDMEM) is to be flushed from memory.

**Arguments:** pkt_id

**Returns:** void

### 4.4.4 PL_flush

**Prototype:** void PL_flush (pkt_id)

**Description**
This function flushes a packet from cache.

**Arguments:** pkt_id

**Returns:** void

### 4.4.5 PL_get_spec

**Prototype:** void PL_get_spec (int * NCHAN_ptr, int * N_PKT_ptr)

**Description**
Gets the specifications of driven PL block: NCHAN and NPKT. Notice that block must first
have been initialized with PL_init.

**Arguments:**
- OUT int * NCHAN_ptr
  - Filled with NCHAN
- OUT int * N_PKT_ptr
  - Filled with NPKT

**RETURNS:** VOID

### 4.4.6 PL_store

**Prototype:** int PL_store (int pkt_id, longword * pkt_ptr)

**Description**
Initiates a packet storage request.

**Arguments:**
- IN int pkt_id
  - Assigned packet identifier (decided by the overall system).
- IN longword * pkt_ptr
  - Packet base address.

**Returns:** int
- 0: packet ready to be requested.
- Else: an error occurred.
4.4.7 PL_read

Prototype: int PL_read (int pkt_id, int chan_id, word offset, longword * result_ptr)

Description: Performs a read request.
Arguments:
- IN int pkt_id
  - Packet identification.
- IN int chan_id
  - Channel identification.
- IN word offset
  - Offset of the longword to be read.
- OUT longword * result_ptr
  - Filled with the longword read.

Returns: int
- 0: longword successfully read.
- Else: an error occurred.

4.4.8 PL_write

Prototype: int PL_write (int pkt_id, int chan_id, word offset, longword data)

Description: Performs a write request.
Arguments:
- IN int pkt_id
  - Packet identifier.
- IN int chan_id
  - Channel identification.
- IN word offset
  - Offset of the longword to be written.
- IN longword data
  - New longword data.

Returns: int
- 0: longword updated.
- Else: an error occurred.

4.4.9 PL_read_data

Prototype: int PL_read_data (int pkt_id, int chan_id, word offset, longword * result_ptr)

Description: Performs a read request from packet’s PDMEM
Arguments:
- IN int pkt_id
  - Packet identification.
- IN int chan_id
  - Channel identification.
- IN word offset
  - Offset of the longword to be read (between 0 and PDMEM_SIZE – 1).
- OUT longword * result_ptr
  - Filled with the longword read.

Returns: int
- 0: longword successfully read.
- Else: an error occurred.

4.4.10 PL_write_data

Prototype: int PL_write_data (int pkt_id, int chan_id, word offset, longword data)

Description:
Updates one longword into packet’s PDMEM.

Arguments:
- IN int pkt_id
  - Packet identification.
- IN int chan_id
  - Channel identification.
- IN word offset
  - Offset of the longword to be written (between 0 and PDMEM_SIZE-1).
- IN longword data
  - New longword value.

Returns: int
- 0: longword successfully updated.
- Else: an error occurred.

4.5 PL register map

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Reset value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS</td>
<td>IS</td>
<td>0</td>
<td>Address of a new packet to store</td>
</tr>
<tr>
<td>PARAM</td>
<td>IS</td>
<td>0</td>
<td>Packet parameters</td>
</tr>
<tr>
<td>OP0 To OPNCHAN-1</td>
<td>IS</td>
<td>0</td>
<td>Packet access operations</td>
</tr>
<tr>
<td>DATA0 To DATANCHAN-1</td>
<td>IS</td>
<td>0</td>
<td>Packet access data</td>
</tr>
</tbody>
</table>

Figure 42: registers map
5 THE FIREWALL

5.1 Introduction

To improve the overall performances of a gateway using NDCL and PL hardware modules, a dedicated Firewall is developed. In addition a complete system approaches improve portability and design re-use.

The main features of our Firewall are:
- Support for IPv4
- Compliant with IPv6
- Stateful Inspection support for TCP and UDP protocol
- ICMP, IGMP packet filtering
- Programmable “over IP” protocol packet filtering.
- API definition for add-ons as upper layer stateful inspection modules
- Operating System Independent (as much as possible)
- Protection against common hacker attacks
- Protection against Denial of Service (DoS)
- Web-based management interface
- Comprehensive event logging
- Strong pre-defined security policy
- Small footprint
- Tightly integrated with NDCL and PL components
- Architecture open to support existing and forecoming Application Level Gateway (i.e. ALG for HTTP, FTP, etc...)
- Flexible and scalable access policies

5.1.1 The case of Linux IPv4 protocol stack

As our development is based on Linux, an introduction to Linux Firewall is described hereafter.

Below Figure shows the simplified Linux IP networking stack.
When a network interface card (NIC) receives an ethernet frame, which matches the local MAC address, the broadcast or the multicast address, its device driver issues an skb (sk buffer) to the protocol stack (refer to “Integration” chapter for implementation details).

A packet journey in the protocol stack crosses 3 netfilter hooks; each of them may have external registered functions that are called:
- PRE_ROUTING hook is called at the entry point of the packet journey
- FORWARD hook is called after routing decision is performed
- POST_ROUTING hook is called at the output point of the packet journey.

In addition to packet journey related hooks, 2 additional hooks:
- LOCAL_IN hook when locally generated packet enters in the protocol stack
- LOCAL_OUT hook when locally generated packet exit from the protocol stack

### 5.2 Firewall Architecture

This section describes STM Firewall architecture.

Making usage of NDCL, the dedicated firewall translates its configuration parameters as a set of NDCL rules. Each rule is made up of constrains on packet fields and of a description of action to be taken on the packet when it satisfies that rule. This ruleset are called acl-ruleset.
Each classified packet reports the classification result trough the PL. The firewall will collect that classification result to provide a comprehensive output describing how the packet must be handled.

Next figure represents, in a schematics way, the logical flow of operations without take care about real system architecture.

![Diagram](image)

**Figure 44: Protocol stack, NDCL and Firewall work-flow**

In addition to the information provided by the classification, the FW may required other packet information for its processing (e.g. information that change at packet frequency).

**5.2.1 Functional Blocks**

STM Firewall functional architecture is described in the following picture.
OLIPHANS Firewall Engine

- Dynamic State Table (DST)
- Connection Tracking Engines (CTE)
- Act Sub-system (AS)
- Log Sub-system (LS)
- Firewall Configuration Tool (FCT)
- Security Policy Parser (SPP)
- FSPD
- LR

packet Loader

Conf registers | Sync registers | Output registers

Input registers | NDCL

Read: a functional block reads the contents of pointed register or memory.
Write: a functional block writes the contents of pointed register or memory.
Trig: a functional block trigs the activity of pointed functional block.

Figure 45: Overall architecture of OLIPHANS firewall engine

Firewall building blocks are:
- Firewall Security Policy Database (FSPD). To process the packets, the firewall must be configured by a set of rules expressed by constrains on packet fields. These configuration parameters are stored in FSPD.
- Firewall Configuration Tools (FCT). This is a graphical User Interface Command Line (CLI) for the user to initiate the security policy. FCT translates and writes firewall Security Parser Data (FSPD) in NDCL rulesets format.
- Security Policy Parser (SPP) checks if packets are conforming to security policy. It interprets NDCL outputs.
- Action Sub-system (AS). This component applies on packets the action as specified in the matching rules set by the user. AS returns to the calling system providing the firewall result.
- Logging Sub-system (LS). This component takes account of the packet analysed by the firewall. The accounting level for a packet is specified in the FSPD in the rule matched by the packet.
- Log Repository (LR). The Logging sub-system writes to this component the produced accounting information.

### 5.2.2 Functional overview

When a packet is extracted from the backlog queue, it is cached in the PL module, and becomes available for NDCL. NDCL starts packet classification coherently to its configured and active rulesets. After NDCL processing, the protocol stack begins to process packet till it reaches hook from which Firewall engines is called. Firewall engine polls NDCL's synchronization register to guaranty the coherency between packet classification and Firewall packet activity. The Security Policy Parser (SPP) implements the needed synchronization task.

The result of NDCL classification is loaded in PL cache area. Packet topology is provided as an output of NDCL classification. Packet topology is a structure (one per identified protocol in the packet) made of an identifier for that protocol and of a pointer to the packet area where this protocol is identified.

To allow anti-spoofing, FW configures the NDCL so that each physical interface is associated with a set of IP source addresses allowed for that physical interface.

- The firewall configures NDCL rulesets to returns a code to the SPP that is interpreted after by the AS and LS modules.

When a log is requested, the LS module writes the corresponding log to the Log Repository.

The Action identifier returned by NDCL classification selects AS appropriate behaviour, which may change packet’s data path in the protocol stack or generate a control packet. The Actions used in the rules are:

- **ACCEPT**: accept the packet. AS sends to Protocol Stack calling system a command to put the packet on the output interface.
- **DROP**: drop the packet and do not notify the sender. AS sends to calling system a command to discard the packet.
- **REJECT**: drop the packet and notify the sender with an ICPM message-notifying destination unreachable.
- **REDIRECT**: packet is forwarded to upper level modules. Those modules are needed to implement per-application connection tracking engines.
The Log identifier returned by NDCL classification selects the appropriate behaviour of LS, which may write different levels of information on Log Repository. The logging level identifiers are:

- **NOTRACK**: log sub-system does nothing. There is no packet accounting
- **SHORT**: In this case the log sub-system takes into account IP and the upper layer protocol parameter.
- **LONG**: The whole packet header is logged.
- **SUPERLONG**: The whole packet is logged.

**Note**: the Last rule of acl-ruleset must be a clean up rule, this rule allows not recognized packets to systematically be DROP (packet action) and SHORT (log action).

In addition the Firewall must track TCP session or virtual session of TCP to check if its associated state machine is not corrupted. This function needs 2 modules:

- the DST (Dynamic State Table) storing the state of the active connections
- the CTE (Connection Tracking Engine) which analysis the packet state.

A DST entry consists of a session descriptor and session state parameters. A session descriptor is a unique identifier of a session of transport protocol (Typically TCP and UDP). It is derived form a set of fields belonging to the headers of the different layers of the same packet. The session identifier is unique for all packets belonging to the same transport session. The state session parameters are a set of variables used to store the state (or a derived state) of transport protocol session. In addition, CTE stores useful connection related information in DST table.

The CTE implements the following behaviour:

- for each new session, add an entry in DST.
- Remove a session entry from the DST when it terminates or expires.
- When connection tracking is set for a packet, check for the session entry pointed by the session descriptor if the packet is synchronized with its session state parameters. Finally it updates its DST state parameters accordingly to the received (and expected) packet.

The firewall implements optionally stateful inspection or packet filtering on selected protocols.

### 5.3 Firewall Security Policy Database (FSPD)

#### 5.3.1 Rules and Ruleset

A **rule** \( R \) is a set of conditions defined on \( k \) fields of packet header. We denote with \( H \) the set of these \( k \) fields of interest and call it the **channel identifier**. Each field \( R[i] \) in a rule can specify any of the three kinds of matches: exact match, prefix match, or range match;

- It is an **exact match** if a single value is specified for the \( i \)th rule field (i.e. \( R[i] \)) and the channel identifier field \( H[i] \) is equal to \( R[i] \);
- It is a **prefix match** if a prefix is specified for the \( i \)th rule field and the first \( \text{length}(R[i]) \) binary bits of the channel identifier field \( H[i] \) are the same as those of \( R[i] \);
- It is a **range match** if a range of values \( R[i]=\text{val1}...\text{val2} \) is specified for the \( i \)th rule field and the channel identifier field \( H[i] \) falls into that range, i.e. \( \text{val1} \leq H[i] \leq \text{val2} \);
A rule $R$ is a matching rule if each field $H[i]$ matches the corresponding field $R[i]$. The type of match is specified by $R[i]$; it could be an exact match, a prefix match or a range match.

A set of $N$ rules is called ruleset, and is denoted by $RS$. Today implementation defines each filter by:
- destination IP address
- source IP address
- protocol type
- destination port (if Protocol type is TCP or UDP).

Source port

Note: For ICMP / IGMP, instead of destination transport port, the condition is expressed through ICMP Message Type field.

Transport-derived state tracking needs other header information that is specific for each protocol (i.e. TCP flags field for session state analysis). Meanwhile, stateful inspection does not require to configure dedicated rules per connection direction (upstream and downstream).

Next figure shows an example of a ruleset from the user point of view. Fields of the rules are expressed in mnemonic form, but they are stored in the FSPD in different way as we will see later in the document.

<table>
<thead>
<tr>
<th>Rule ID</th>
<th>IP Source</th>
<th>IP Destination</th>
<th>Service</th>
<th>Action</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Internal-net</td>
<td>Any</td>
<td>http</td>
<td>Accept</td>
<td>Notrack</td>
</tr>
<tr>
<td>1</td>
<td>Internal-net</td>
<td>DNS-server</td>
<td>DNS-query</td>
<td>Accept</td>
<td>Short</td>
</tr>
<tr>
<td>2</td>
<td>AdmWorkstation</td>
<td>Any</td>
<td>Echo-request</td>
<td>Accept</td>
<td>Short</td>
</tr>
<tr>
<td>3</td>
<td>Any</td>
<td>admWorkstation</td>
<td>Echo-reply</td>
<td>Accept</td>
<td>Short</td>
</tr>
<tr>
<td>4</td>
<td>Any</td>
<td>Any</td>
<td>Any</td>
<td>Drop</td>
<td>Long</td>
</tr>
</tbody>
</table>

Figure 46: A ruleset example

5.3.2 Physical interfaces security parameters

A Firewall must select valid source IP addresses per hardware interface.

A Valid Addresses for an interface is:
- A packet whose source IP address belongs to Valid Addresses is allowed to enter the firewall through the interface.
- A packet whose source IP address does not belong to Valid Addresses is not allowed to enter firewall through the interface.

A specific data structure describing the hardware interfaces is needed. FCT allows users to configure the physical Interfaces security parameters editing this Interface parameters which is stored in a file called Interfaces.def.
Once user has defined interfaces security parameters, FCT Tool translates them in a ruleset format. This anti-spoofing ruleset, is stored in Rulesets.def file.

Each rule of an anti-spoof ruleset contains four fields:
- Ruleset ID which identify the selected ruleset
- Interface reference
- IP source address
- Action field
- Log field

Next figure summarizes components and their relations that made up Firewall Security Policy Database.

![Firewall Security Policy Database Diagram](image)

**Figure 47/ Firewall Security Policy Database**

### 5.4 Firewall Configuration Tool (FCT)

It allows the user to CONFIGURE the firewall. Its main functions are:
- Managing entities as NetObj, ServiceObj, Interface.
- Managing rules and rulesets.
- Deriving anti-spoof ruleset from NetObj and Interface entities.
- Configuring Firewall default parameters (i.e. default policy: DROP or ACCEPT, IP fragment: DROP or IGNORE, spoofing check: YES or NO, Log level of spoofed packets).
- Translating firewall rulesets consistent with NDCL supported format.

#### 5.4.1 Managing NetObj, ServiceObj and Interfaces

Each time a new NetObj is created, a unique identifier is associated to this the new element.
Removing a NetObj or ServiceObj means deleting a Objects raw in “objects.def” file. Before removing this object, FCT parses the rules to verify if this object is used. For each rule Source and Destination fields are analysed. If one of them contains the object identifier to be removed, then scanning stops and FCT notifies to software upper level that object cannot be removed.

FCT manages automatically the unique identifier for NetObj and ServiceObj objects; user does not specify the identifier.

FCT manage the coherency of the configuration parameter, so that there is no contradicting rules.

### 5.4.2 Rules and rulesets management

#### 5.4.2.1 Acl ruleset

To define rules it is necessary to define NetObjs and Services first. FCT stores rules in Rulesets.def file acl section by row line. Rule removal is achieved by removing selected stored row line.

These changes in firewall configuration do not take effect until rulesets update is not instantiated in the NDCL.

Rulesets stored in the Rulesets.def file are not suitable for NDCL, FCT compiles the information before to update the NDCL database.

#### 5.4.2.1.1 Rules on Acl rules

Not all the possible combinations of acl rules are admissible as ruleset.

To explain these possible incoherency situations we first give some definitions. So, we define:

- **Rule index.** It is the storing position in the ruleset table. It is defined in $[0, 255]$. Head rule has index 0. We indicate $r_i$ the rule stored in the $i$ position (index).

- **Rule subspace.** If we denote with $\prod$ the three dimensional space built as Cartesian product of source domain space, destination domain space and service domain space, then we could indicate with $\prod_i$ the subset of $\prod$ defined by rule $r_i$.

- **Action severity.** It is an action property. For Actions already defined its value is assigned as:
  - severity(ACCEPT) = 0
  - severity(REDIRECT) = 1
  - severity(REJECT) = 2
  - severity(DROP) = 2

We indicate with $s_i$ severity of the action defined per rule $r_i$.

Now, given two rules $r_i$ and $r_j$ with $j > i$, we are able to define acl rules compatibility versus their subspace intersection as reported in the following table:

<table>
<thead>
<tr>
<th>$I \equiv \prod_i \cap \prod_j$</th>
<th>$r_j$ coherency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I = 0$</td>
<td>$r_i$ and $r_j$ are compatible</td>
</tr>
</tbody>
</table>
\[ I = \prod_i = \prod_j \]  
\[ I = \prod_i \quad (\prod_{j \geq i} \) \]  
\[ I = \prod_i \quad (\prod_{j \leq i} \) \]  
\[ I \neq \prod_{i \in I} \wedge I \neq \prod_{j \in J} \]  

\( r_i \) and \( r_j \) are not compatible

\( s_j > s_i \Rightarrow r_i \) and \( r_j \) are compatible

\( s_i \leq s_j \Rightarrow r_i \) and \( r_j \) are not compatible

\( r_i \) and \( r_j \) are not compatible

Rules could be considered compatible if and only if ActionID and LogID are set in the same way in each rule. In that case rules are compatible, but not normalized. Otherwise, ActionID or LogID are set in different way, rules are considered incompatible because for subset \( I \), they define two different and contradictory firewall behavior.

Thus, when user inserts a new acl rule, the FCT needs to start a coherency check process. That process verifies that new rule is compatible with the rest of ruleset. FCT applies constrains described in previous table in two steps. First, inserted rule coherency with rule above it is tested, so newly inserted rule plays the role of \( r_j \) and each rule above it plays the role of \( r_i \) with \( i \) in \([0, j - 1]\). Subsequently, rules below new rule are analyzed in order to verify their compatibility with it that now plays the role of \( r_i \) (rule below play the role of \( r_j \) with \( j \) in \([i + 1, numRule - 1]\)).

When newly inserted rules is incoherent with one of already inserted acl rules, FCT stops the validation process and alerts the user with a message notifying the first rule to which it is incompatible. Obviously new rule are not inserted.

5.4.2.2 Anti-spoof ruleset

After the user defines interfaces security policy parameters, FCT, on saving command of Interfaces.def file, translates security policy parameters into anti-spoof ruleset written in the anti-spoof section of the Rulesets.def file.

FCT manage the configuration so that it is consistent and not incoherent.

For each interface the number of rules generated is equal to the number of NetObjs.

5.4.3 Translating and installing rulesets into NDCL

Once user defines and stores firewall security policy, in order to make it running, it is necessary to install the compiled rulesets into the NDCL. FCT translates acl and anti-spoof ruleset into “NDCL configurator” suitable format.

5.5 Security Policy Parser (SPP)

The aim of SPP system is to enforce the security policy rulesets resolution done by NDCL. NDCL resolves two rulesets for firewall engine and provided as ACTION and LOG parameters.

The first security check that SPP has to do, when it receives ruleset resolution data, is to apply the anti-spoofing rules as stored in security policy parameters.

If ActionID of anti-spoof matched rule is set to ACCEPT, the process continues, otherwise, the ActionID is set to DROP and the process stops.

SPP proceed with the drop, sending a DROP command to AS module with a specified LogID.
Note: for anti-spoof ruleset the LogID value MUST be NOTRACK when ActionID is ACCEPT.

If anti-spoof check is passed, SPP analyses the NDCL output and NDCL topological analysis result to determine if further processing is needed (only the over-IP protocol number is used).

If over-IP protocol isn't TCP or UDP, SPP call AS module with ActionID set to DROP and LogID to LS. In all other case SPP calls the appropriate CTE (one for UDP and One for TCP) to investigate on the packet. At the end of CTE investigation process, CTE directly sends commands to AS and LS.

5.6 Action Sub-system (AS)
This component applies the action on the analysed packet. Actions are specified in the matched rule or derived from analysis made on packet by Connection Tracking Engine. It must interoperate with firewall engine calling system (protocol stack) in order to influence the packet journey. On the other hand, the calling system must be able to understand the firewall engine verdict.
We have already defined three possible verdict of packet analysis: ACCEPT. In this case the AS must send a command to protocol stack to say that packet is able to go ahead in its journey.
DROP. AS must inform the protocol stack that the packet under analysis is not conform to the security policy, then it must be removed from global processing of the system.
REJECT. In this case the AS acts as in the DROP case. Furthermore it must build a Destination Unreachable ICMP packet for the sender and put it on the appropriate NIC device driver queue.
In each of the case AS must send a signal to PL in order to inform it that packet analysis is completed. Next packet can be load to be processed.

5.7 Log Sub-system (LS)
LS received command from the SPP and/or the CTEs, it write information log into a log file following a predefined scheme.
To retrieve necessary information, LS performs several read requests to the Packet Loader, which depends on the log level defined for the packet under analysis and specified in the Log command.

5.8 Dynamic State Table (DST)
Dynamic State Table is used by Connection Tracking Engines to store and update the needed information to follow up transport protocol sessions. It is created and maintained in system memory by Connection Tracking Engine, too. Each time firewall configuration is updated, by an installation of a new security policy in NDCL, the Dynamic State table is flushed. This is an obligated requirement to avoid incoherency between static ruleset and dynamic rules, without take a huge synchronization process. However, as we will see later, CTE is able to rebuild active connection inside a fresh DST.
We can figure each entry of dynamic state table as a state session descriptor of an active connection through the firewall plus other fields as timing information and logging instructions. To uniquely identify a TCP or UDP connection it is enough to consider (Src_IP, Src_Port, Dst_IP, Dst_Port, Protocol) tuple.
The best solution to speed up data insertion/retrieval into/from DST is to implement a lookup table indexed by entries unique identifier (primary key). That key may be the 104-bits length string obtained by the concatenation of tuple’s components, as illustrated in the expression below:

$$DCKey = Scr\_IP + Src\_Port + Dst\_IP + Dst\_Port + Protocol$$

As it is not possible to implement a table of $2^{104}$ entries, independently of the single entry length, a hash function is used to reduced the memory areas. In Consequence, it is necessary to store the key in the entry, to discriminate between equal hashed keys (collision in the hash table).

Then a Dynamic State Table entry is made up of 16 bytes as illustrated in next figure.

Table 2: DST entry format

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCKey</td>
<td>104</td>
</tr>
<tr>
<td>rcvTime</td>
<td>120</td>
</tr>
<tr>
<td>State</td>
<td>124</td>
</tr>
<tr>
<td>LogID</td>
<td>128</td>
</tr>
</tbody>
</table>

The fields are:
- **DCKey field** to check for possible collisions of the hashed value of different keys. In this way it is possible to verify that an entry is the desired entry.
- **The State field**: used by TCP-CTE to store the 5 possible state of an active TCP session.
- **The rcvTime field**: specifies the receiving time of the latest packet (expressed in seconds). This value is updated each time a new packet of that session cross the firewall. Its length is 16 bits. It is used to verify that a packet is within certain timeout values. When time reaches rcvTime value plus current state timeout, entry is not more valid and its storing position may be reused.
- **LogID (4 bits)** is the log identifier reported in the authorizing rule. It is sent to LS when CTE analyses a packet belonging to session described by the entry. Actually, we need this information only for packet crossing the firewall in opposite direction of the session opening packet.

Since any session is a bi-directional communication and related entry is created and stored into a position derived from first packet fields, it is necessary to find a way to address the same entry with packets traversing firewall in opposite direction. In this case, values of the fields used to build the key stored in the entry are in reverse order. Therefore, for those packets, the key is made up in the following way.

$$ICKey = Dst\_IP + Dst\_Port + Scr\_IP + Src\_Port + Protocol$$

### 5.9 Connection Tracking Engines (CTE)

Connection Tracking Engines analyse packets and sessions on specific protocol basis. A set of dedicated Protocol-CTE (i.e. TCP-CTE, UDP-CTE, ecc…) is defined to follow the session of the protocols supported for Stateful Inspection analysis. The appropriate Protocol-CTE is selected by the SPP based on packet protocol type field.

If the protocol is not supported for stateful inspection analysis, the packet inspection is ended by SPP (it forwards the acl ruleset resolution result to AS and LS).
SPP called the protocol-CTE with as parameter the NDCL classification result (ActionID and LogID of the matched rule) and the positions of the IP and protocol header into Packet Loader (PL) cache.

**5.9.1 UDP-CTE**

UDP-CTE is called by SPP when a packet is an UDP packet. First of all, UDP-CTE extracts needed fields from the IP and UDP headers.

**UDP connections are simpler to maintain, as they are stateless so, once the necessary fields are extracted from the packet, UDP_CTE builds DCKey and ICkey**

UDP-CTE builds and uses DCKey if ActionID=ACCEPT, and ICKey in the others cases.

Hashing ICkey and/or DCkey, UDP-CTE checks if DST corresponding entry is equal to the computed key. If yes, analysed packet belongs to an active session, its rcvTime field is updated with the packet receiving time. If no, this is a new session, an entry is added to the dynamic state table, its rcvTime field is set to packet-received time. In both cases packet is accepted by the firewall, thus, UDP-CTE sends ACCEPT command to AS and matched rule’s LogID to LS.

Adding an entry has the effect to authorize packets in the opposite direction within an UDP_TIMEOUT time interval.

If SPP rejects the packet, the UDP-CTE behaviour is slightly different. If an entry exists (analysed packet belongs to an active session) UDP-CTE compares packet-received time with rcvTime value stored in the entry. When this difference is bounded in UDP_TIMEOUT parameter, packet can be accepted. Then, ACCEPT command is sent to AS, session LogID to LS and rcvTime field is updated with the receiving time of packet. If packet receiving time exceeded UDP_TIMEOUT or an entry doesn’t exist, UDP-CTE sends matched rule ActionID to AS, so packet isn’t accepted, and LogID to LS.

Next table summarises activities done by UDP-CTE in the two possible cases.

**Table 3: UDP-CTE functional behavior**

<table>
<thead>
<tr>
<th>ActionID = ACCEPT</th>
<th>ActionID &lt;&gt; ACCEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve necessary fields (to build lookup key) from Packet Loader</td>
<td>Build DCKey and Lookup DST by hash(DCkey)</td>
</tr>
<tr>
<td>Build DCKey Lookup DST by hash(DCkey)</td>
<td>If entry exists (valid timeout, too) Update rcvTime</td>
</tr>
<tr>
<td>If entry exists</td>
<td>Send ACCEPT to AS</td>
</tr>
<tr>
<td>Update rcvTime</td>
<td>Send Entry LogID to LS</td>
</tr>
<tr>
<td>Else</td>
<td>Else</td>
</tr>
<tr>
<td>Insert entry in DST</td>
<td>Send ActionID to AS</td>
</tr>
<tr>
<td>Send ActionID (ACCEPT) to AS</td>
<td>Send LogID to LS</td>
</tr>
<tr>
<td>Send LogID to LS</td>
<td></td>
</tr>
</tbody>
</table>
5.9.2 TCP-CTE

For the TCP protocol TCP flag field influences the packet destiny and checks sequence. So TCP-CTE first action is to extract the TCP flag from it. Therefore it performs a read request to the Packet Loader (i.e. read(tcp_hdr_offset, 13, 1) where the values expressed in the calling instruction have the same meaning illustrated in previous paragraph)

TCP-CTE drops the packet, calling the AS with DROP argument, when:

- TCP flags are set in contradictory way
- TCP flags are set in legal way, but their combination is not what TCP-CTE expects for that session.
- TCP flags are set in legal way, but packet doesn’t belong to any of active sessions

TCP options

Contradictory flags situations are:

- SYN/FIN is probably the best known illegal combination. Remember that SYN is used to start a connection, while FIN is used to end an existing connection. It is nonsensical to perform both actions at the same time. Many scanning tools use SYN/FIN packets, because many intrusion detection systems did not catch these in the past, although most do so now. We can safely assume that any SYN/FIN packets you see are malicious.

- SYN/FIN/PSH, SYN/FIN/RST, SYN/FIN/RST/PSH, and other variants on SYN/FIN also exist. These packets may be used by attackers who are aware that intrusion detection systems may be looking for packets with just the SYN and FIN bits set, not additional bits set. Again, these are clearly malicious.

- Packets should never contain just a FIN flag. FIN packets are frequently used for port scans, network mapping and other stealth activities.

- Some packets have absolutely no flags set at all; these are referred to as "null" packets. It is illegal to have a packet with no flags set.

On the contrary we consider legal packets that have flags field set up as follow:

- SYN, SYN/ACK, and ACK are used during the three-way handshake which establishes a TCP connection.

- Except for the initial SYN packet, every packet in a connection must have the ACK bit set.

- FIN/ACK and ACK are used during the graceful teardown of an existing connection.

- RST or RST/ACK can be used to immediately terminate an existing connection.

- Packets during the "conversation" portion of the connection (after the three-way handshake but before the teardown or termination) contain just an ACK by default.

- Optionally, they may also contain PSH and/or URG.

After a positive TCP flag analysis, the TCP-CTE needs to verify if packet belongs to an active connection, so it needs to build a key for DST lookup. In order to retrieve building key fields, it performs two read requests to the Packet Loader. (i.e. read(ip_hdr_offset, 12, 8) returns the IP source and destination address, while read(tcp_hdr_offset, 0, 4) returns the source and destination port fields. We assume that bytes in headers are numbered starting from zero).

As for UDP-CTE, TCP-CTE builds DCkey if ActionID of matched rules is ACCEPT, and ICkey in the others cases.
In addition to TCP flags control the packet must be compliant with the Finite State Machine of a session. As we know, TCP protocol doesn’t define a FSM for the end-points involved in a session. So, the state for each end-point may be different for each of them during a session (especially in opening and closing phases). For this reason, since our goal is to track a TCP session, not the state of the two session end-points, we define a FSM for a TCP session derived from the TCP FSM. Next figure illustrates it. Remarkable items are:

- In each of the state, only the packets that cause a transition from it to another (or from it to itself) can be accepted.
- For each state of the FSM is defined a timeout value as illustrated in figure 18. TCP-CTE uses these values to enforce timeout constrains on valid packets (packets that cause state transition). Actually, CTE accepts a packet, that might cause state transition, if and only if difference between its receiving time and previous packet receiving time is within the timeout value defined for session current state which packet belongs to.
- Each TCP session, having an associated entry in DST, is considered closed when “current time” is beyond the timeout related to session status plus receiving time of its last packet (stored in its rcvTime field). In the graph the timeout transactions aren’t reported.

In the FSM graph it is also reported events that cause a new FSM instance creation. For TCP-CTE point of view TCP session can be open in two possible cases.

- **SYN packet accepted by acl ruleset which doesn’t belong to an already active session.**
- **ACK packet accepted by acl ruleset which doesn’t belong to an already active session.** Actually, this is a special case needed to reopen a session after a DST flush.
Next table summarize the TCP-CTE behaviour in each of possible cases. It is important to note that the flags condition is expressed in general way. The “per state” acceptable packets are just illustrated by FSM reported in previous figure. On that figure state transition labelled with “client” prefix indicates that the specified packet can be accepted only if it comes from the beginner of session. On the contrary, “server” prefix means that packet can be accepted only if it is a response packet. When a transition is not labelled with any prefix means that packet can be accepted in both directions.

<table>
<thead>
<tr>
<th>ActionID = ACCEPT</th>
<th>ActionID &lt;&gt; ACCEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve tcp flags field from Packet Loader If flags are set in contradictory way Send DROP to AS Else</td>
<td>Retrieve necessary fields (to build lookup key) from Packet Loader Build Dckey and lookup DST Build ICkey and lookup DST</td>
</tr>
</tbody>
</table>

Figure 48: TCP session FSM graph
<table>
<thead>
<tr>
<th>ActionID = ACCEPT</th>
<th>ActionID &lt;&gt; ACCEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>If SYN packet /ACK packet</td>
<td>Send ActionID to AS</td>
</tr>
<tr>
<td>Add new entry setting its state to</td>
<td>Send LogID to LS</td>
</tr>
<tr>
<td>SYN_SENT/RE_OPENING</td>
<td></td>
</tr>
<tr>
<td>Send ACCEPT to AS</td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
<tr>
<td>Send DROP to AS</td>
<td></td>
</tr>
<tr>
<td>Send LogID to LS</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry doesn’t exist</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>If flags are conform to state’s session</td>
<td>Send ActionID (ACCEPT) to AS</td>
</tr>
<tr>
<td></td>
<td>Else</td>
</tr>
<tr>
<td></td>
<td>Send DROP to AS</td>
</tr>
<tr>
<td></td>
<td>Send LogID to LS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry exists</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>If flags conform to state’s session</td>
<td>Send ActionID to AS</td>
</tr>
<tr>
<td></td>
<td>Else</td>
</tr>
<tr>
<td></td>
<td>Send DROP to AS</td>
</tr>
<tr>
<td></td>
<td>Send LogID to LS</td>
</tr>
</tbody>
</table>

Figure 49 TCP-CTE functional behavior

5.10 IP Fragmentation

The NDCL doesn’t classify IP fragmented packets, but recognize it by looking inside the IP flags field. When IP packet is a fragment, the NDCL specifies a special code in one of its output register where it sets L4 protocol identifier on topological analysis of non fragmented packet.

That output information, written in the local memory of Packet Loader where packet under analysis is stored, allows the firewall (SPP subcomponent) to be aware of the IP fragments, and to change its usual behaviour to manage it.

As we already mentioned in Firewall Configuration Tool paragraph, OLIPHANS firewall will treats IP fragment in two different ways:
- DROP: in this case SPP sends a DROP command to AS and the LOG level code defined for IP fragment to LS.
- IGNORE: in this case SPP completely ignore IP fragment. This means that IP fragments are accepted without any check on them. In order to act in this way SPP, once realized that packet is an IP fragment, sends ACCEPT command to the AS, and from the firewall point of view, packet processing is concluded. This behaviour is very dangerous and should be avoided.

Its default is behaviour DROP, but it can be changed by FCT modifying a general configuration parameter.

Caching to reassemble, before inspect could be result in DoS vulnerability

6 INTEGRATION

The reader will report to “system view” chapter discussing the application architecture and introducing the integration.
6.1 Packet Reception

Figure 50: Packet reception blocks

6.1.1 Classification

When a packet is received on the network device, it raises an interrupt and the interrupt handler invokes the netif_rx() function the entry point in Linux Network layer.

Netif_rx() function is modified to call the Classification Manager.

Normal Linux packet reception starts as soon as the NIC driver stores the packet into the backlog queue. The propose integration diverts the packet reception to first classify and then proceed with the standard packet reception.

The classification manager is the activity scheduler, it synchronize all block accesses maintaining system consistency between packets, modules and configuration.

Due to the parallel processing, a decoupling FIFO is needed between packet reception and NDCL packet manipulation. This is the role of CM_queue, Netif_hook_packet() enqueues the packet in it when called by NETIF_RX process.

When NDCL becomes available, classification process starts, when completed, an interrupt is raised (connect to NDCL_completed event). Finally, if received from a NIC, the classified packet is stored in backlog queue, it continues its journey in Protocol Stack (implemented in _netif_rx() function).
Classification Manager calls the APIs of the drivers with a call flow described later in this document.

IRQ_handler() is the function called following the hardware interrupt generated when NDCL has finished a classification (NDCL_completed = 1). IRQ_handler function ends classification of the current packet.

Note: classification is also based on the interface number, this implementation uses skbuff net_device parameter.

### 6.1.2 Firewall

In Reception's the Firewall is connected to Protocol stack with a Netfilter hook (NF_IP_PREROUTING). When in it’s protocol journey, the packet arrives at this hook, the PL has created the association with a tag ‘a packet identifier) and is already classified. The Firewall can read the classification results stored in PL cache and proceed with it.

### 6.1.3 Execution step

The following execution steps are implemented:

1. NIC raises an interrupt, NIC driver calls netif_rx
2. Netif_rx calls Classification manager
3. The packet is stored (by PL driver)
4. Classification Manager checks if NDCL is busy or not:
   4a NDCL is not engage, The packet is classified immediately.
   4b NDCL is engage. The packet is put in CM_queue.
5. NDCL raises the event NDCL_completed
6. IRQ_handler enqueues the packet into the backlog queue (_netif_rx());
7. IRQ_handler checks if there is a local packet to be classified:
   7a if YES; classifies local packet immediately
   7b if not, and if CM_queue is not empty it takes a packet from CM_queue and restart a new classification
   7c returns if no packet is ready for classification.
8. The following operation takes place after packet enqueueing in the backlog queue.
9. The packet arrives in netfilter hook and it is taken by Firewall (registered on this hook)
10. Firewall reads the classification results (PL_read_data) and in some cases some fields of packet (PL_read)
11. Firewall returns a verdict to Protocol stack.
12. Firewall removes the packet from system (PL_remove)

### 6.2 Locally generated packets

Locally generated packets are handled like packet reception, but need a specific interface, which is described hereafter.
To trap locally generated packet, Classification Manager is connected to Linux protocol stack NF_IP_LOCAL_OUT Netfilter hook. As classification module needs a reference to a physical interface, a fake Interface reference number set to “0xFF” is provided.

As locally generated packet does not suffer from synchronisms, while been requested to classify local packets, the requestor waits for NDCL availability when it is already engage in other classification process.

After classification, the locally generated packet is analyzed by the Firewall. FW reads the classification results and a needed additional field from PL. FW provides a verdict to Protocol Stack.

To make sure that packet classification operates before FW, Classification manager is registered with the highest priority (NF_IP_PRI_FIRST) while Firewall is registered with a lower priority (NF_IP_PRI_FIRST + 1).\(^{30}\)

### 6.3 Configuration

Configuration process is the phase where user configures Firewall with system rules. System rules are well-defined filters to apply to packet reception. The following figure illustrates the blocks used for configuration.

---

\(^{30}\) Max priority is the lowest number.
When user writes the rules through the Firewall Configurator, the rules are converted in a NDCL Configurator format. To avoid concurrent access, NDCL configurator requests a configuration lock to NDCL driver. When completed NDCL configuration lock is de-asserted.

Configurator can:
- Modify class mask
- Modify ruleset mask
- Write ruleset memory
- Write monodimensional classifier memory.

**Note:** When already lock for classification, NDCL driver waits for NDCL availability (loop until register READY1 is updated with the appropriated value) before to process with NDCL configuration update.

### 6.4 Project Integration

This chapter describes how to map the project on the Mediaref. Application directory tree is implemented as described hereafter:

```
drwxr-xr-x  2 root Oliphans 4096 Jul 27 18:28 build
drwxr-xr-x  2 root Oliphans 4096 Jul 28 19:28 CManager
drwxr-xr-x  4 root Oliphans 4096 Jul 21 12:53 firewall-00.03
-rwxr-xr-x  1 root Oliphans 773 Jul 21 12:53 build.sh
-rw-r--r--  1 root Oliphans 154 Jul 26 10:59 make.def
drwxr-xr-x  2 root Oliphans 4096 Jul 29 16:35 ndcl
```
in which:
- **build**: directory contains executable file.
- **build.sh**: script to build and install modules.
- **CManager**: directory of CM code.
- **firewall-00.03**: directory of FW code.
- **make.def**: file contains information on Mediaref Linux root and network file system.
- **ndcl**: directory of NDCL engine code.
- **ndcl-conf**: directory of NDCL configurator code.
- **NDCL_driver**: directory of NDCL driver code.
- **pl**: directory of PL engine code.
- **PL_driver**: directory of PL driver code.
- **simmk**: directory for simmk tool.
- **svi**: directory for svi code.

The following steps must be followed:
- Create Executable modules:
  - Use **build.sh** script to create and install the hardware modules to run on ST20 (NDCL and PL).
  - Use **make** to create drivers modules to run on ST40 (NDCL configurator; classification manager; Firewall).
  - Use **`make install`** to install drivers modules to run on ST40 (NDCL configurator; classification manager; Firewall).
- Use **`insmod simmk.o`** command to Load SimMK. To run the simulation (this command starts SVI on ST40 and NDCL and PL on ST20):
  - Use **startmod.sh** script to load and start ST40 modules:
    - **NDCL and PL driver**.
    - Classification configurator and manager.
  - Use **fw_1.sh** script to load and register the Firewall.

### 7 CONCLUSION
This work has demonstrated that a network protocol based on software can be improved by a set of basic HW modules.

A packet loader, as in traditional processor architecture, enhances the overall performances. The NDCL module improves classification reactivity and classification management. The better one achieves an integration of the NDCL module with the protocol stack, the better will be the performances. In this demonstration, we accommodate with a minimum set of modification in the protocol stack to maintain portability of the protocol stack. It is possible to get even better performances, at the expense of less portability.
All system functionalities were validated after porting HW models onto a real platform, which is by itself a first valuable output of the project. Replacing the SW models of those hardware blocks and based on the simulation result we got on the platform, the estimated gain in throughput by should be a factor of 20 compared to today’s application within STM.