20450 Century Boulevard
Germantown, MD 20874
Fax: (301) 515-7954

NETFP

Software Design Specification (SDS)

Revision A

SDOCM00117686

February 9, 2016

Revision Record	
Document Title: **Software Design Specification**	
Revision	Description of Change
A	1. Initial Release
	2. Added Syslib4 NETCP usage, NETFP Subsystem details, and NETFP procedures.

Note: Be sure the Revision of this document matches the QRSA record Revision letter. The revision letter increments only upon approval via the Quality Record System.

TABLE OF CONTENTS

1 Scope

This document describes the functionality, architecture, and operation of the Network Fast Path (NETFP) Library which allows applications to develop their code to use the NETCP subsystem.

2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

No	Referenced Document	Control Number	Description
1	Syslib User Guide		SYSLIB User Guide
2	NETFP API Documentation		NETFP Doxygen API documentation
3	SYSLIB Unit Test document		SYSLIB Unit Test documentation
4	NETFP Proxy SDS		NETFP Proxy Software Design Specification

Table 1. Referenced Materials

3 Definitions

Acronym	Description
API	Application Programming Interface
DSP	Digital Signal Processor
NETFP	Network Fast Path Library
PA	Packet Accelerator
SA	Security Accelerator
NETCP	Network coprocessor
JOSH	Job Scheduling

Table 2. Definitions

4 NETCP usage

The NETCP subsystem on the Keystone devices allows packet classification, IPSEC processing, Air Ciphering, Fragmentation & reassembly support. The NETCP subsystem comprises of a Packet Accelerator (PA) and Security Accelerator (SA) and numerous PDSP which are programmable blocks. The documents describe the basics of the NETCP subsystem; please refer to the PA & SA documentation for more details.

4.1 Ingress Packet flow

The following figure illustrates the flow of packets through the NETCP subsystem as configured by the NETFP library.

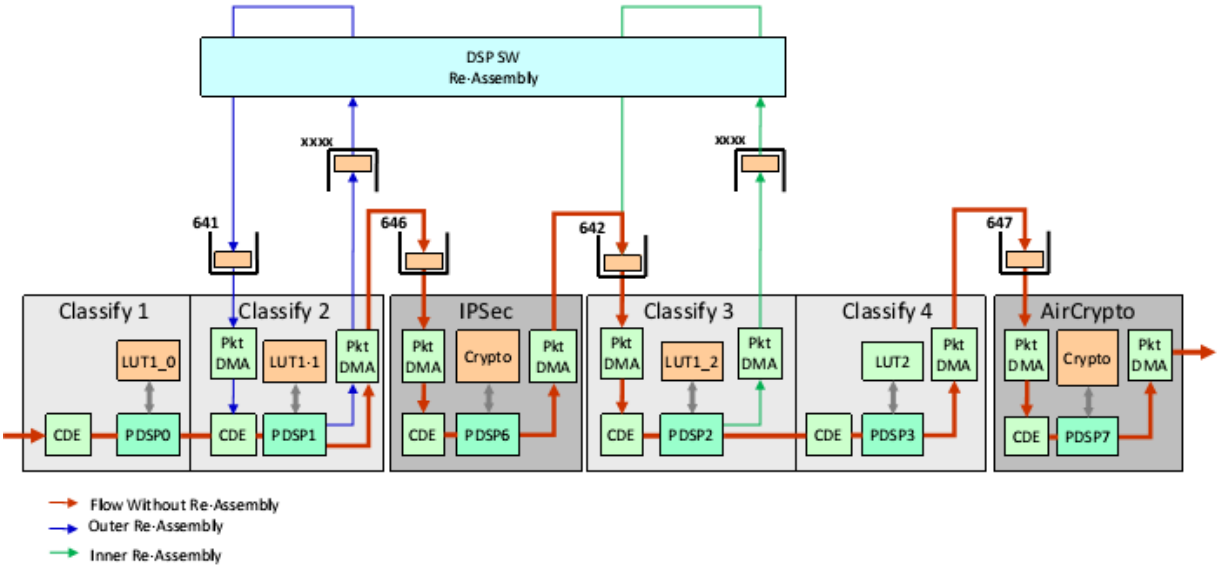


Figure 4-1. NETCP Ingress Packet Flow

The Classify blocks are used to match fields in the packet and take appropriate action. The action might be to continue parsing or could be to send the packet to the host. If the classification fails the packets are pushed out through the fail route. The IPSEC block is used for encryption and decryption of packets while the AIRCRYPTO block is used for the air side ciphering & deciphering.

This is a simplified flow which explains the ingress packet flowing through the system.

1. Packets arrive into the NETCP subsystem and are initially classified in the LUT1-0 block. The fields checked here are those which are in the standard Ethernet header. Typically, if the packet matches the destination MAC address packet processing continues to step (2)

2. The packet is then passed to the LUT1-1 block where the IP header fields are checked. So if the packet matches the destination IP then packet is going either through decryption step (3) or UDP classification step (4).
3. The packet is pushed to the IPSEC block where it is decrypted. After decryption the packet is pushed back to LUT1-2 to classify the inner IP header. If there is a match continue to step(4)
4. The layer4 properties (UDP Port or GTPU Identifier) are matched here and if there is a match the packet can be passed to the Host (DSP or ARM user space processing) or it could be configured to send the packet to the AIRCRYPTO where it can get ciphered and then get pushed to the Host.

Notes:

- Every time a packet needs to be passed to the Host core; the NETCP subsystem uses a "CPPI PA flow" from where a CPPI descriptor is picked up and then the packet is pushed into the configured completion queue
- At any step, if packet is not matching any of the rules, it is forwarded to Linux network stack

4.2 Egress Packet Flow

The picture below illustrates packet processing on egress direction.

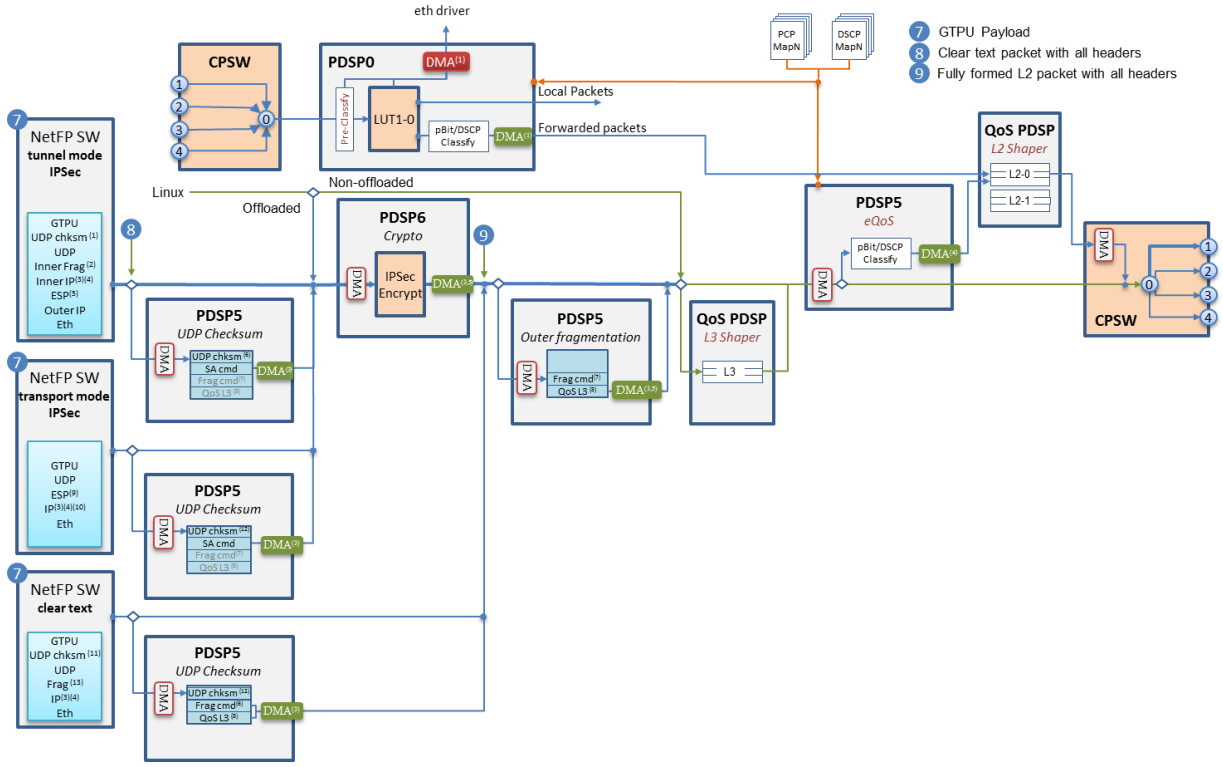


Figure 4-2. NETCP Egress Packet Flow

Packets originated from Linux network stack can be offloaded (encrypted using NetCP HW). Non-offloaded packets as well as packets after encryption can be hierarchically shaped using dedicated HW. Once packet transmitted by Linux stack, there is no SW in the loop any more.
Packets originated from DSP or ARM user space application can use HW acceleration for UDP checksums, and Fragmentation. Those packets may go through HW encryption module followed by hierarchical shaper. Once packets are delivered to HW, there will be no more SW interaction with a packet.
Packets forwarded by NetCP are classified and shaped without SW in the loop.

5 NETFP

5.1 Design Goals

The NETFP library is designed to meet the following major requirements:

- DSP/ARM realm independency
  This implies that the NETFP library should provide the same symmetric set of services irrespective of whether the application is executing at the ARM user space or on the DSP
- Resource Management & Multiple NETFP subsystem coexistence
  NetFP design shall allow multiple independent applications to create independent set of services (for example, WCDMA and LTE radio access technologies). There shall be a clear delineation between common and application-specific set of services.
- Coexistence with a traditional networking stack
- Performance
  NetFP services shall be seen as an acceleration option to the existing Linux network services

5.2 System Architecture

After accounting for the various design goals the NETFP System architecture was designed as follows:

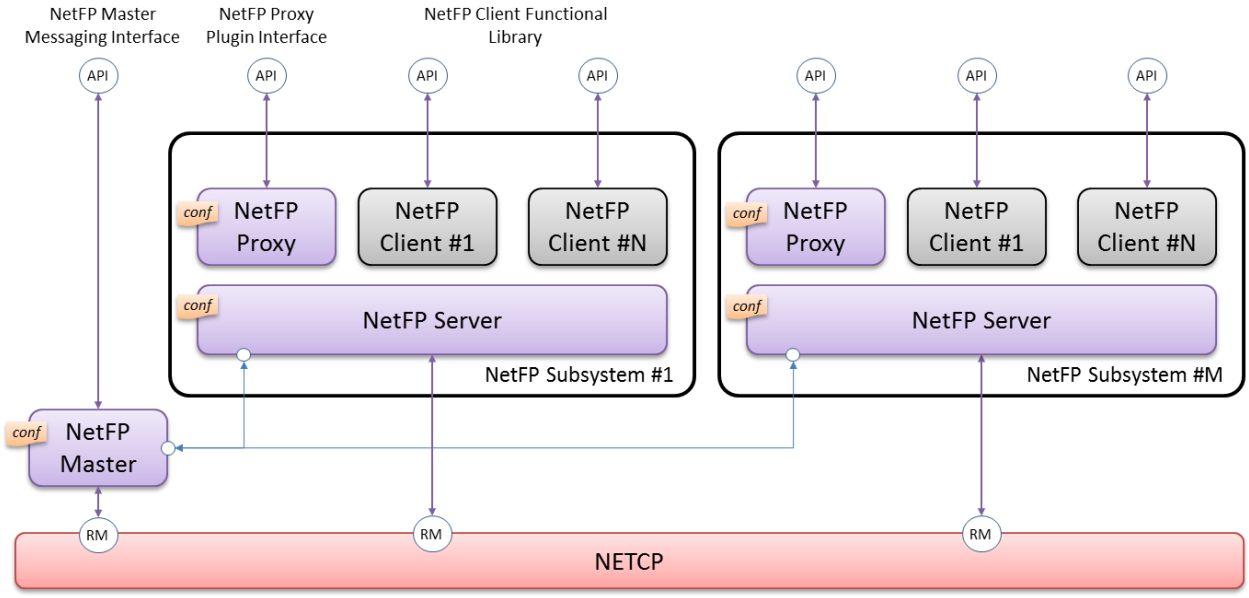


Figure 5-1. NetFP System Architecture

Each module is explained in detail in the following sections.

5.3 NETFP Master

One of the design goals was to ensure that there could a coexistence of multiple NETFP subsystems on the same box. This implied that each NETFP subsystem can be bought up/down without affecting the other subsystem.

The NETFP Master is an application which performs one time initialization and configuration of the NETCP subsystem. There are certain services which are shared by the NETCP subsystem and by moving them out to a NETFP master which is a common application we can satisfy this design goals. The NETFP Master is responsible for the following:-

1. **SA Firmware:**
   The SA Firmware is downloaded and initialized once. This is a common feature which is used by all the NETFP subsystems.

2. **Reassembly Handling:**
   All fragmented packets entering the IP layer classifiers need to be reassembled. The fragments could belong to different flows; each flow could belong to a different NETFP subsystem.

3. **Layer2 QOS Configuration:**
   The master initializes the L2 QOS configuration which requires flows and descriptors to be created. The switch ports can be shared by multiple NETFP subsystems so the flow creation and descriptor management is a common service provided by the master.

4. **Port Mirroring/Capturing:**
   Since the switch ports can be shared by multiple NETFP subsystem, the port mirroring/capturing feature is a common service.

5. **Pre-classification:**
   This allows all broadcast and multicast packets arriving on the switch port to be routed to a particular queue before parsing the LUT1-0 entries. This frees up the LUT1-0 entries. Since this feature is on a per switch port basis and so it is a part of the NETFP master

The NETFP Master exposes a messaging interface which allows the OAM applications/NETFP Servers to communicate with it. The master is configured using a NETFP Master Configuration file. Sample file are present in the `ti/apps/netfp_master` directory.

5.4 NETFP Subsystem

One of the design goals was to ensure that the NETFP services were available on both the DSP and ARM. In order to satisfy the requirement the NETFP module was designed to use the Client/Server architecture as illustrated in the figure below

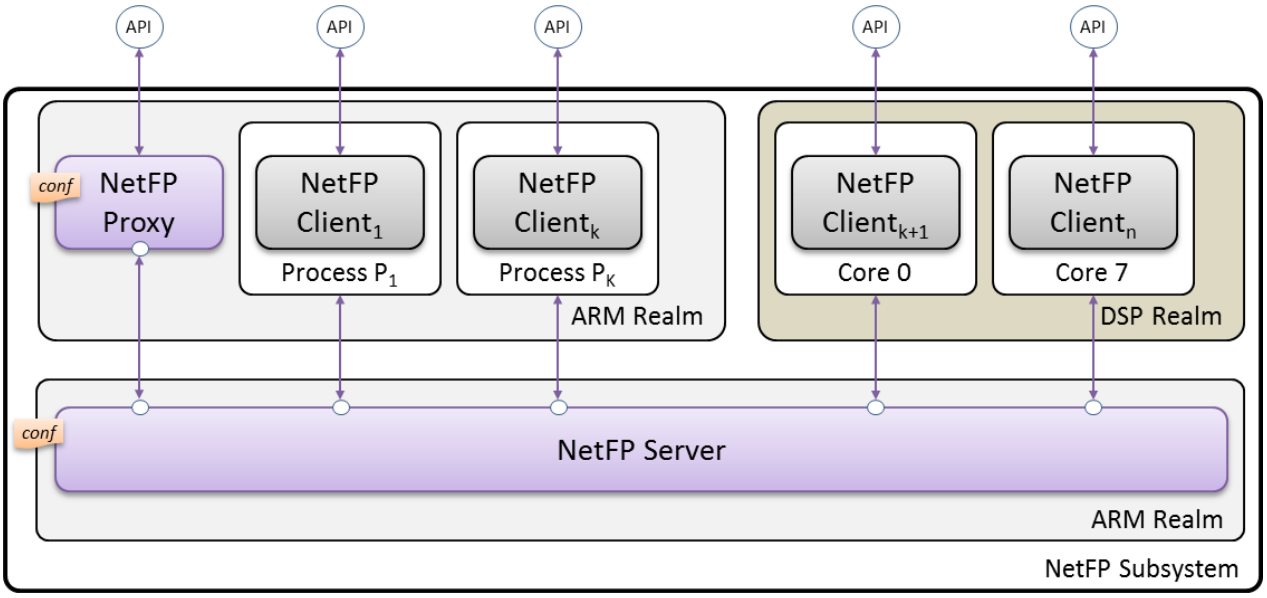


Figure 5-2. NetFP Subsystem

The NETFP server is a standard daemon which executes in the context of the ARM realm. The NETFP server is responsible for the following services:

- Configuration of the NETCP subsystem (PA and SA). This allows a centralized distribution of hardware resources.
- Responsible for the management of any entity which could be shared among the multiple NETFP clients.
    - Networking Interface
    - Secure & Non Secure Fast Paths
    - Security Policy and IPSEC security channel configuration
    - Air Ciphering security channel configuration
- Management of the NETFP clients.

The NETFP server can be configured using a configuration file. The configuration file specifies the following information:

- List of all the NETFP clients which can attach themselves with the server
- Maximum number of security channels & the base security context

The NETFP clients execute in the context of the application and provide a well-defined interface which allows the clients to communicate with the NETFP server for configuration. The NETFP client is also responsible for the management of entities which are client specific and cannot be shared across clients:

- NETFP Sockets
- NETFP 3GPP Channels
- NETFP Reassembly

This document explains the design of the NETFP Subsystem.

5.5 NETFP Proxy

The NETFP Proxy is a special NETFP client which interfaces with the NETFP Server. The Proxy executes on the ARM is also an interface to the Linux networking subsystem. The proxy ensures that the NETFP subsystem and the Linux networking systems are synchronized with each other.

NETFP Proxy daemon provides the following functionality:

- Provides an interface to the management application to start/stop security policy offloads to NETFP.
- Validates policy request from application; retrieves all necessary information such as SA properties, policy properties etc. from Linux kernel and sets up the SAs/SPs in NETFP server.
- Interface management
- Route computation
- Keeps the neighbor (ARP) entries in use by NETFP alive in Linux stack to keep stack in sync with NETFP.
- Maintains Linux stack and NETFP in sync in case of SA rekeying events.

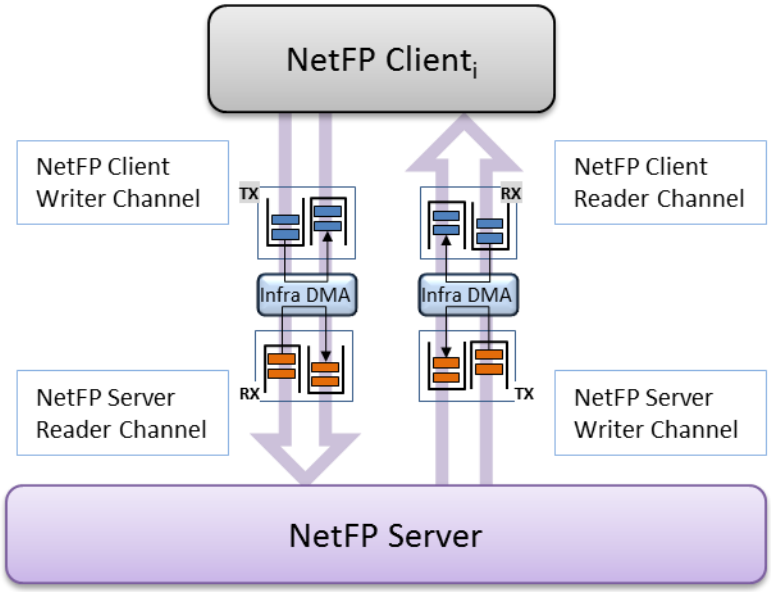
For more information about the NETFP Proxy please refer [4] to the NETFP Proxy SDS.

6 NETFP Subsystem Details

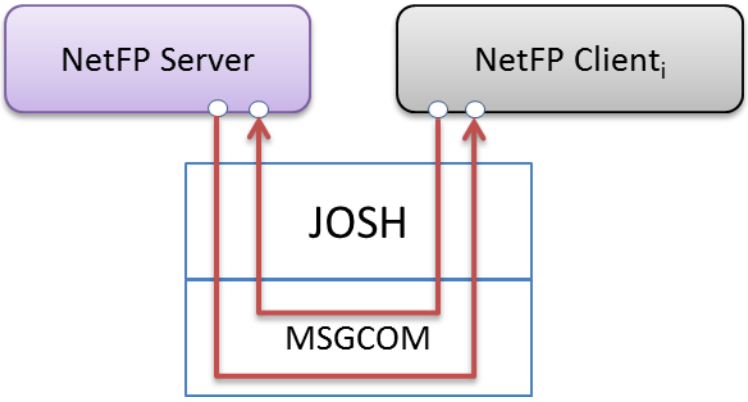
6.1 Basic Communication

The design choice of communications between NetFP Service and Clients is dictated by the need for low latency reliable links between independent entities.

The physical layer communications between NETFP Server and Clients are provided by MsgCom channels and are built using infrastructure CPPI PktDMA channels with Direct Interrupts. The figure below showcases the basic communication pipes.



The logical layer communications between NETFP Server and Clients are provided by JOSH (Job Scheduling services).



JOSH provides the following services:

1. **Job Execution**
   It allows the NETFP Server/Clients to specify functions which can be executed by the remote end as jobs. Once a function is registered as a job with JOSH; it can be invoked by the other entity.

2. **Parameter Passing**
   Parameters to a function can be passed by reference or by value. JOSH provides a protocol which allows parameters to be abstracted from the authors of the function.

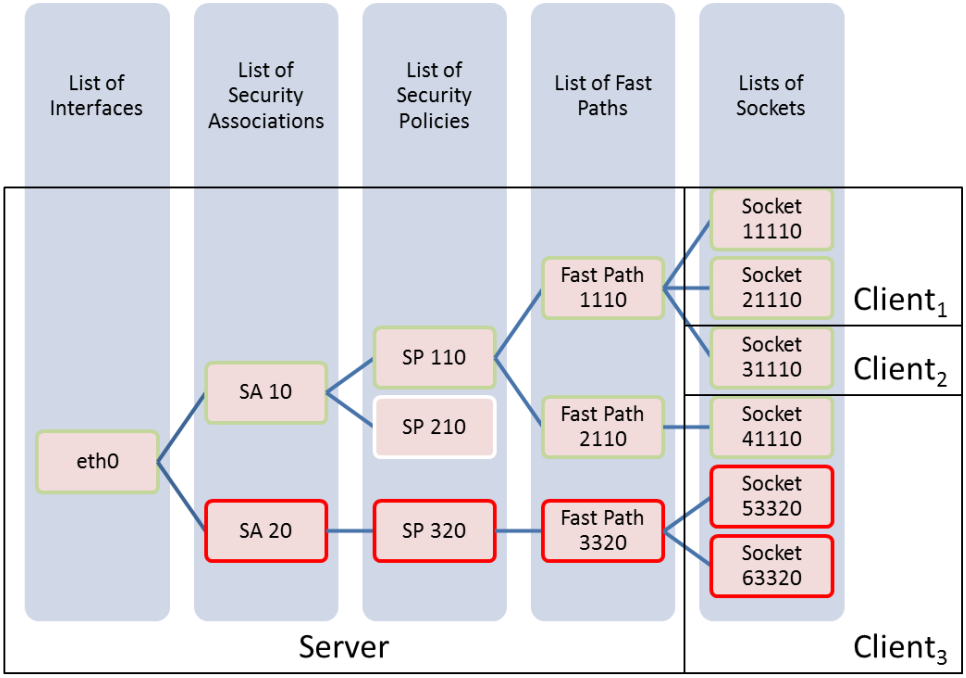
3. **Synchronous/Asynchronous Job execution:**
   Synchronous job execution implies that the callee is blocked till the job is successfully executed by the remote end and the results are posted back.

   Asynchronous job execution implies that the callee is immediately returns after scheduling the job at the remote end. The callee can check as required to determine if the function execution was complete and the status of the operation.

The convention used in the code is that if there is a NETFP function `Netfp_xxx` then the corresponding JOSH job if applicable is called `_Netfp_xxx`

6.2 NetFP Data Objects model

The picture below illustrates the hierarchical design of the data objects maintained in the NetFP subsystem. The objects are distributed between Proxy, Server and multiple Clients. Ingress and Egress components are handled independently, so one can think of two independent sets of objects maintained for inbound and outbound traffic.



Consider as an example a single interface object (“eth0”) is used by two outbound Security Associations (“SA 10” and “SA 20”). “SA 10” is used by two Security policies: “SP 110”, and “SP 210”. Two Fast Paths had been created using policy SP 11: “FP 1110”, and “FP 2110”. Multiple clients created outbound sockets based on FP 1110, 2110, and 3320.
Security Policy “SP 210” is illustrating the offloaded secure Linux traffic, and therefore, the security association “SA 10” is commonly known as a “shared SA”.
The hierarchical aspect is used by NetFP to manage its data objects. For example, if for any reasons (e.g. expiration) the “SA 20” becomes “sick”, the “infection” propagates to the dependent policies, fast paths, and sockets, making all the chain “infected” and not capable of sending packets. Once SA gets “cured” (e.g. successful re-keyed), the chain may become operation again. Please note that this is just a high level illustration, and in practice the rekeying process is a bit more complicated.

6.2.1 Interface Module

The NETFP server module supports the creation of interfaces. There are 2 types of interfaces which can exist in the System:-

1. **Physical interfaces**
   Physical interfaces are actual interfaces which are connected to the switch port and can be used to send/receive data.

2. **Virtual Interfaces**
   These are interfaces which are created on top of the physical interfaces and which use the physical port features to send/receive data. *Example:* VLAN Interfaces or aliased IP interfaces

Physical interfaces are predefined up front in the NETFP Master through the NETFP Master’s configuration file. The NETFP server receives the physical interface configuration dynamically through the NETFP Master using the master exported messaging interface.

The NETFP module supports the creation of multiple interfaces. Interfaces are associated with the following configuration properties:

- Type of interface (Ethernet/VLAN)
- MAC address associated with the interface
- MTU supported on the interface
- Mandatory IPv4 address & subnet mask associated with the interface

Interfaces can be created using the NETFP API: `Netfp_createInterface`. The interface module supports only 1 mandatory IPv4 address and subnet mask which is specified during interface

creation. For IPv6 however the interface module should be capable of the management of multiple IPv6 addresses.

Interfaces can be removed from the system using the NETFP API: `Netfp_deleteInterface`. Interfaces can only be deleted when all the routes using that interface have been removed.

NOTE: Interfaces can be created by any NETFP client but they are shared across all the NETFP clients. This implies that the NETFP Interface API's mentioned above actually translate to JOSH jobs which execute on the NETFP Server.

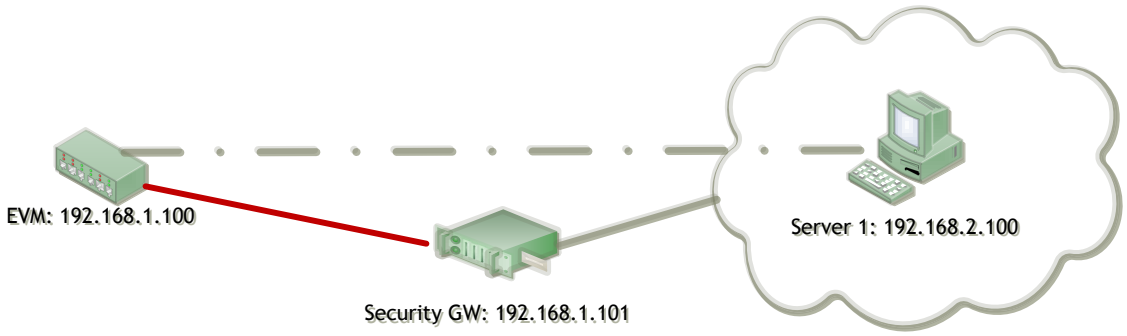
6.2.2 Security Association

Security associations are created in order to use the IPSEC tunnels. Unique security associations are created in both Inbound and Outbound direction. Security association configuration has the following properties:-

- IP Sec Protocol
- Authentication and Encryption keys
- Lifetime
- Security Parameter Index associated with the association
- IP Address of the EVM and the peer where the tunnel terminates

Security Associations are exchanged between the EVM & remote peer through IKE. Please refer to the NETFP Proxy [4] on more details.

The following figure describes the network architecture and explains the various concepts.



The EVM (192.168.1.100) and Security Gateway (192.168.1.101) have a secure link with each other. The security associations are exchanged between these two entities. Even though the final communication is between the EVM (192.168.1.100) and Server1 (192.168.2.100); all packets sent by the EVM on the wire will be destined for the Security Gateway. Similarly on the

ingress side; all packets received from the Security Gateway will be sent by the NETFP to the NETCP subsystem for decryption.

The EVM & Security Gateway IP address combination is called the “Outer IP Header” and the EVM & Server1 IP address combination is called the “Inner IP Header”.

The NETFP API `Netfp_addSA` can be used to add a security association and the `Netfp_deleteSA` can be used to delete a previously created SA. The SA can only be deleted if there is no other entity in the system which is using it.

NOTE: Security associations can be created by any NETFP client but they are shared across all the NETFP clients. This implies that the security associations are managed in the context of the NETFP server.

6.2.2.1 LUT Programming

Security associations involve the programming of the LUT1-1 entry to match the “Outer IP header”. If there is a match the packet is then passed to the SA for decryption; after decryption the packet is passed back to LUT1-2 for “Inner IP Header” lookup. (See LUT Programming section for Fast Path) If there is a match the packet is then passed to the LUT2 for the Layer4 lookup (UDP Port or GTPU Identifier).

6.2.3 Security Policy

Security Policy defines the properties of the tunnel and link with a security association. It also acts as a permission checker who decides the Layer3 and Layer4 properties which are allowed to use the specified security association. Security policies like security associations are unidirectional.

The Security Policy configuration has the following properties:-

- Range of allowed Ingress Inner IP addresses.
- Range of allowed Egress Inner IP addresses.
- Range of allowed source and destination port numbers.
- Range of allowed GTPU IDs.
- Security Association handle (NULL if non-secure)

The NETFP API ensures that the security policy is adhered to and any end-point created using the security policy matches the specified ranges. This is done while creating the Layer3 endpoint i.e. Fast Path and also while creating the layer4 endpoint i.e. socket. If it is deemed that the security policy is being violated the NETFP API will return a permission error.

NOTE: Security policies can be created by any NETFP client but they are shared across all the NETFP clients. This implies that the security policies are managed in the context of the NETFP server.

6.2.4 Fast Path

The fast path defines endpoints for which the security policy is applied. The fast path defines the layer3 connectivity which can be used to receive and send packets. Each fast path in the NETFP system is uniquely identified by a name and can be created once on a specific NETFP client and can get reused.

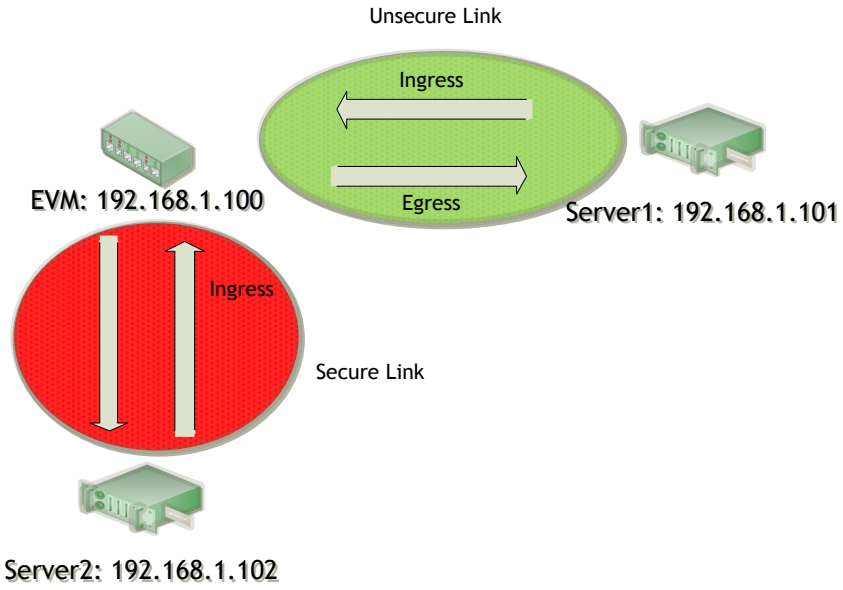
Fast path configuration is also associated with a security policy identifier. This security policy identifier may or may not be associated with a security association. If the fast path is associated with a valid security association it is called a secure connection and all packets sent or received on that fast path will be exchanged using the security association configuration.

Fast Path can be of the following types:-

1. Inbound
   The fast path defines the layer3 connectivity which is required to receive packets on the EVM. Packets destined to the EVM should have the destination IP address the same as the EVM IP address. The source IP address should be that of the remote end.

2. Outbound
   The fast path defines the layer3 connectivity which is required to send packets out of the EVM. Packets have the destination IP address as the remote IP address and use the EVM IP address for their source IP.

The following figure explains these concepts:-



The connection between the EVM and Server1 is *unsecure* which implies that packets exchanged between the Server1 and EVM are plain-text. The inbound Fast Path between the EVM and Server1 will have the following properties:-

- Destination IP address is 192.168.1.100 (EVM IP Address)
- Source IP address is 192.168.1.101 (Server1 IP Address)
- Security Policy Identifier is unspecified (Since this is an unsecure connection)

Egress Fast Path between the EVM and Server1 will have the following properties:-

- Source IP address is 192.168.1.100 (EVM IP Address)
- Destination IP address is 192.168.1.101 (Server1 IP Address)
- Security Policy Identifier is unspecified (Since this is an unsecure connection)

The connection between the EVM and Server2 is *secure* which implies that packets exchanged between the Server2 and EVM are encrypted as per the security association. Security associations are typically exchanged between the EVM and Server2 via IKE. The inbound Fast Path between the EVM and Server1 will have the following properties:-

- Destination IP address is 192.168.1.100 (EVM IP Address)
- Source IP address is 192.168.1.102 (Server2 IP Address)
- Security Policy Identifier should be specified and should match the Ingress security association which is exchanged between the EVM and Server2

Outbound Fast Path between the EVM and Server1 will have the following properties:-

- Source IP address is 192.168.1.100 (EVM IP Address)
- Destination IP address is 192.168.1.102 (Server2 IP Address)

- Security Policy Identifier should be specified and should match the Egress security association which is exchanged between the EVM and Server2

Outbound fast paths can be created but these are not operational until the fast path has been marked as active. An outbound fast path is active only if the route resolution procedure has been completed and a next hop MAC address is known.

NOTE: Fast Paths can be created by any NETFP client but they are shared across all the NETFP clients. This implies that the fast paths are managed in the context of the NETFP server.

6.2.4.1 LUT Programming

For the "Ingress" Fast path the LUT1-x entry needs to be programmed and this is done as per the following rules:-

- "Ingress" fast path is non-secure the LUT1-1 entry is programmed with the IP address configuration passed during NETFP creation.
- "Ingress" fast path is secure the LUT1-2 entry is programmed with the IP address configuration passed during NETFP creation. This rule is linked with the previously created LUT1-1 rule which was added while adding the security association.

6.2.5 Socket

Sockets define layer4 connectivity just as fast paths define layer3 connectivity. Sockets unlike fast paths are bidirectional i.e. they can be used to send and receive data. Sockets are core specific and cannot be used across multiple cores.

6.3 NetFP Procedures

This section describes basic procedures and sequence of operations that are expected to be performed by Application software to fully utilize NetFP System.

The picture below illustrates following procedures:

- Physical Interface Setup,
- Security Policy Negotiations,
- Security Policy Offload,
- Interface configuration,
- Inbound and Outbound Fast Paths Setup,
- Socket/LTE Channel Setup,
- Route Updates and Notification Handling, and

- Re-Keying

From the Application perspective, we had identified following participating components:

- Operation, Administration, and Management (OAM) process,
- IKEv2 daemon,
- Linux Networking and IPSec Stack, and
- Fast Path Application. This Application is a user of NetFP client services (NetFP libraries).

From NetFP system perspective, the participating components are:

- NetFP Master,
- NetFP Proxy, and
- NetFP Server

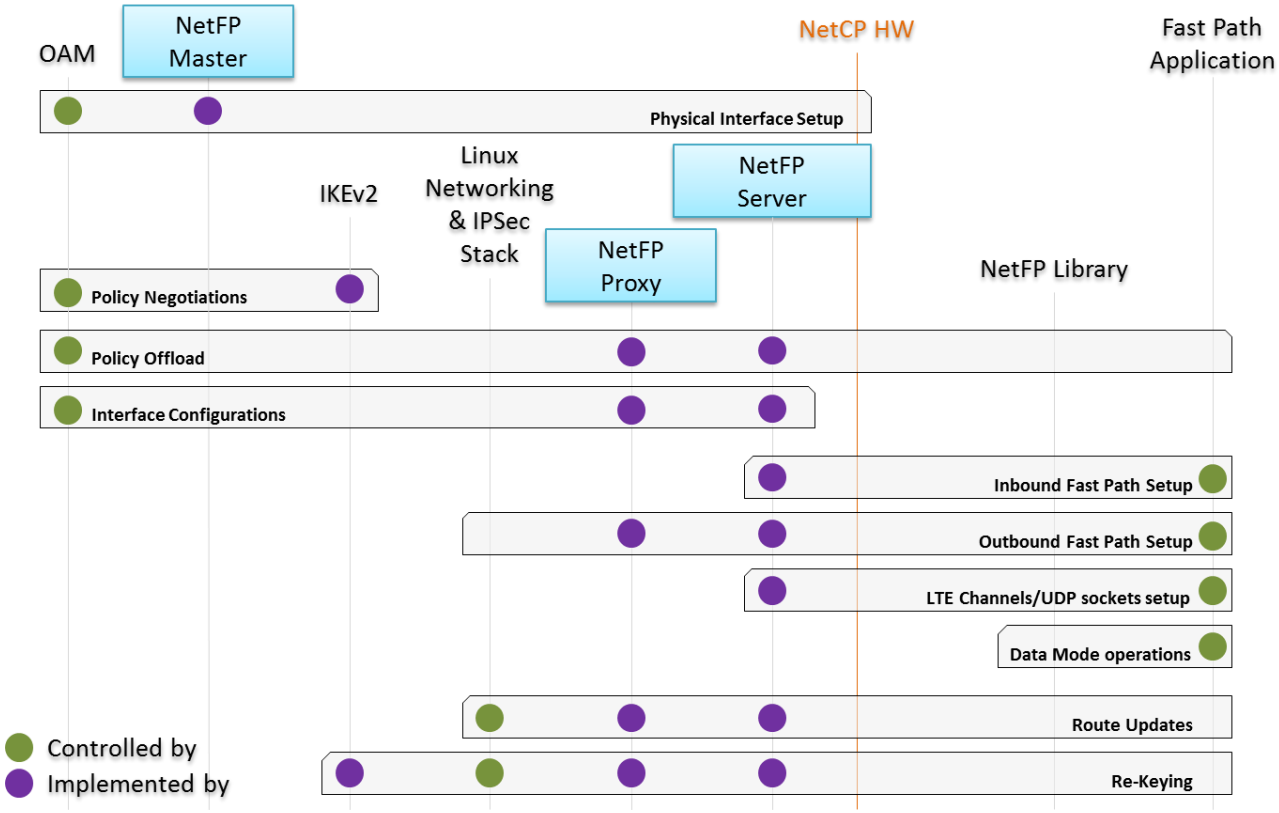


Figure 6-1. NetFP Procedures

6.3.1 Physical Interface Setup

This procedure is used to initialize hardware and SW components used by NetFP service. The procedure is implemented by NetFP Master Module.
The startup configuration is done via NetFP master configuration file. This file is used for one-time NETCP initialization, and provides physical interface configurations at system start up time. This configuration includes:

- Programmability of L2 QoS features:
    a) Initialization of eQoS functionality
    b) Establishment of the rules for locally generated traffic to mark the packets and map them to proper L2 QoS flows
    c) Setting the rules for forwarded traffic
    d) Setting the rules for non-IP traffic
- Programmability to Reassembly feature
- Pre-classification feature

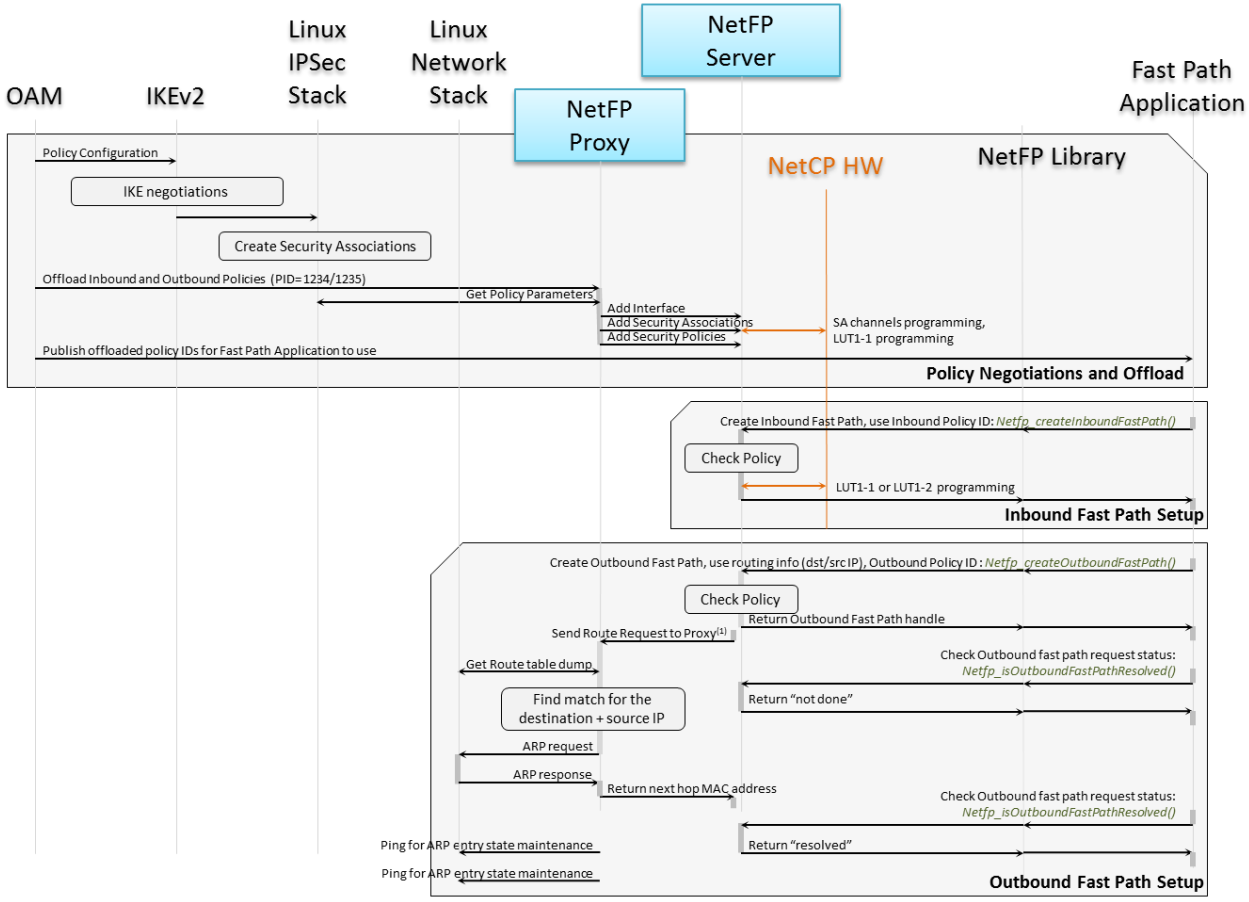
In addition to the start-up parameters, NetFP master provides a messaging interface to support initial parameters re-configurations. OAM can dynamically change the mapping for egress/forwarding traffic, and modify non-IP packets handling. OAM can control port mirroring feature. Messaging interface provides OAM with reassembly stats.

6.3.2 Security Policy negotiations

OAM provides the configuration parameters and policies to a IKEv2 daemon (strongSwan). StronSwan negotiates security associations with the peers. NetFP services are not used during this procedure.

6.3.3 Security Policy Offload

After security associations are negotiated, OAM may decide to “offload” (i.e. transfer cryptographic operations from Linux IPSec SW to dedicated hardware (NETCP Security Accelerator).
OAM process will use the NetFP Proxy API to indicate which policy it desires to offload. NetFP Proxy will obtain all necessary details from Linux IPSec Stack (like ChildSA IDs, algorithms, and key materials), and communicate those to NetFP Server. NetFP server will do proper NETCP programming, which includes configuring NETCP LUTs and security channels.
Since the policy ID is needed to establish the fast path, it is expected that OAM also communicate this IDs with the Fast Path Applications. It may or may not use SYSLIB services (like Name Space data base) to publish this information.
More details on the procedure are shown in the Fast Path creation diagram below. For exact API calls please consult the doxygen API documentation.



6.3.4 Interface Configuration

NetFP proxy provides an API for a controlling Application (OAM) to create and manage an Interface. For example, OAM can set L3 Shaping marking and mapping rules on per interface base. In the case OAM decides to change existing interface properties, the changes are propagated to concerned clients via the notification system.

6.3.5 Inbound Fast Path Setup

Inbound Fast Path Setup is controlled by Fast Path Application executing NetFP Client API. This procedure can be viewed as the establishment of one directional ingress peer-to-peer Layer3 (IP) connection. NetFP client API communicates with NetFP Server, where the policy get checked, and NETCP LUT programing is happening. This API is blocking.
More details on the procedure are shown in the Fast Path creation diagram above. For exact API calls please consult the doxygen API documentation.

6.3.6 Outbound Fast Path Setup

Outbound Fast Path Setup is controlled by Fast Path Application executing NetFP Client API. This procedure can be viewed as the establishment of one directional egress peer-to-peer Layer3 (IP) connection. This procedure includes route resolution steps, so the API usage pattern reflects that. Application first executes an API call to trigger the egress path route resolution.

Since Route resolution may take time, the API signature allows for Fast Path Application to continue, and provides additional API to enquire the status of route resolution process.

6.3.6.1 Route Resolution

The goal of the Route resolution is to determine proper interface and the next hop MAC address to which the packets need to be sent. Once successfully completed, Client will obtain all information needed to mark, shape, and fragment the packet stream.

The determination is always based on both source and destination IP addresses. This information is passed from the Client to the Server and then from the Server to the Proxy. Proxy always consults Linux networking stack. Route resolution supports multiple routing tables and source based routing.
Below is the outline of the NetFP Proxy actions:

- NetFP proxy consults Linux routing table(s) and determines
    a) The IP address of the next hop, and
    b) The [managed] interface to be used. Note that the interface can be either physical (e.g. eth0, eth1) or virtual (e.g. br0)
- NetFP proxy triggers the MAC address resolution of the next hop IP address
- NetFP proxy looks up neighbor cache and finds the next hop MAC address
- NetFP proxy determines the physical interface to be used. In the case of the bridged interfaces it consults the forwarding database.
- NetFP proxy returns back to the server
    a) Interface information (always including physical interface),
    b) source and destination MAC address (next hop) to be used
- NetFP proxy starts maintaining the neighbor caches and (optionally) bridge forwarding database for every interface reported back to the Server
- NetFP proxy starts monitoring the events for every interface reported back to the Server

When an outbound fast path is deleted the NETFP server needs to notify the proxy to stop maintaining/monitoring the interface.

6.3.7 Socket Setup

NetFP sockets API are very much like standard BSD sockets.

6.3.7.1 Socket Create

Sockets can be created for both IPv4 and IPv6. This is done through the `Netfp_socket` API.

6.3.7.2 Socket Bind

All sockets need to be "bound" i.e. they need to be mapped to the following properties:

- Layer3: Ingress Layer3 fast path on which the packet is being received
- Layer4: UDP Port or GTPU Identifier on which the packet is to be received.

Note that after the socket has been "bound" it is only a receiving socket and cannot be used to send data. The binding is done through the NETFP API: `Netfp_bind`. Since binding the socket requires HW programming (LUT2 programming), this operation translates to a JOSH job which is executes on the NETFP Server. Application can register a call back function which notifies the socket if there is a configuration change.

6.3.7.3 Socket Connect

If the socket is to be used to send data also the socket should be "connected" while specifying the following properties:

- Layer3: Egress Layer3 fast path on which the packet is to be transmitted
- Layer4: UDP Port or GTPU Identifier for which the packet is to be transmitted.

Sockets are connected through the `Netfp_connect` API. This API call will also translate to the JOSH job to be executed on the NETFP Server.

6.3.7.4 Data Mode Operations

Once the sockets have been created, bound and connected, they can be used to send and receive data. Packets can be sent over the socket using the NETFP `Netfp_send` API. NetFP sockets cache all the information which is required to send data into their local structure, making the send process fast.

Once a packet is received, the function `Netfp_getPayload` can be used to get the actual payload data. The function will skip all the standard networking headers and will return a pointer to the actual data payload.

The diagram below illustrates sockets setup procedures.

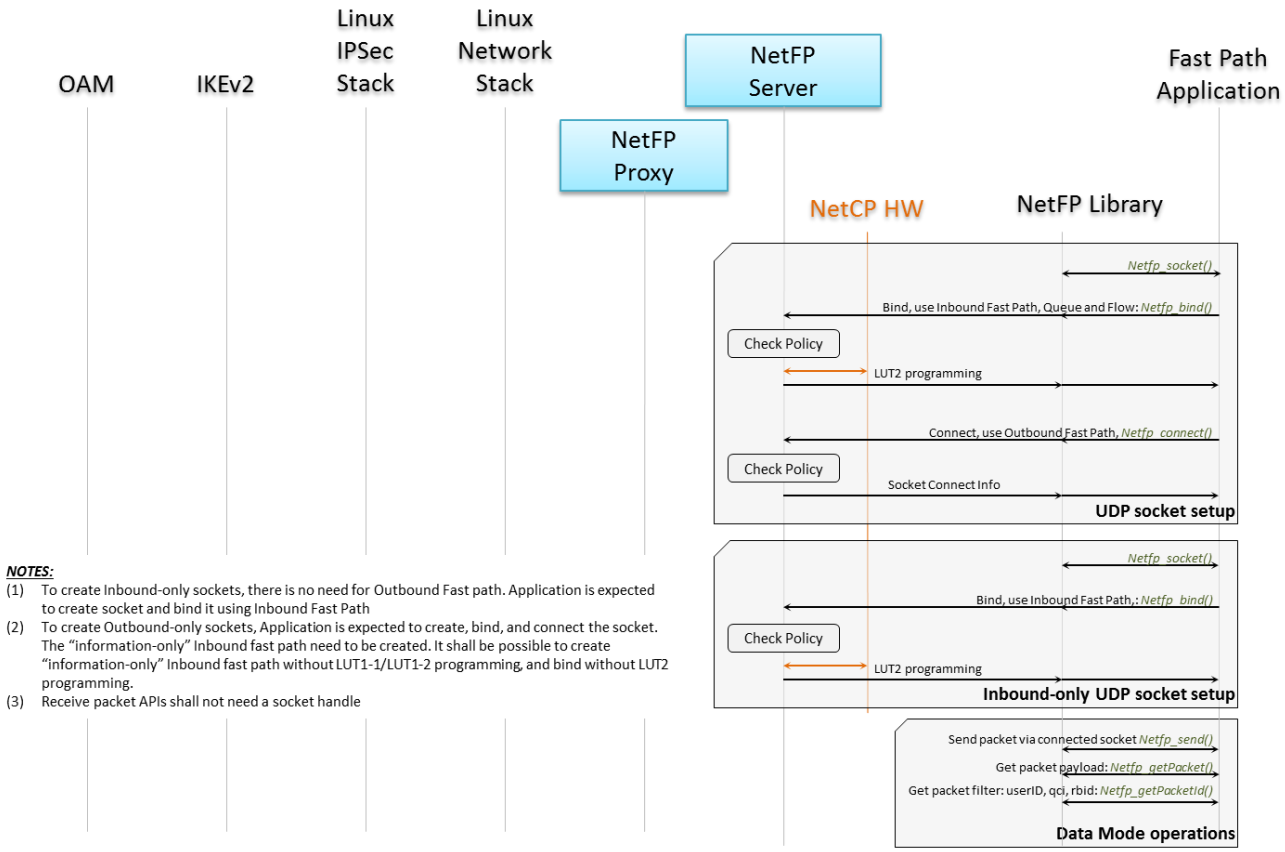


Figure 6-2. Socket Setup Procedures

6.3.8 Event Notifications

In a traditional networking stack every packet is passed through the stack and gets individual treatment. If, for example, the interface is brought down the packet will be dropped, and application trying to send packets will get an error. However, with the Fast Path approach, changes in the stack are not known natively to the NetFP socket, and application will continue sending out packets.
To address the problem, the NetFP is designed to use the hierarchical database and internal event notification mechanism to propagate the information from the changed object to dependent services.

Events could be triggered because of the following reasons:

- NETFP Proxy detects kernel networking event. Events could be neighbor events, interface events etc.
  Example: Interface MTU change, Neighbor is no longer reachable etc.
- NETFP Proxy detects IPSec stack XFRM event. Those can be the SA re-keying events.

- NETFP Proxy gets explicit Route re-configuration request triggered by Application
  Example: Application added a routing table, and issues NetFP Proxy API to update existing routing decisions
- Application executed NetFP Master messaging API to modify physical interface properties.
  Example: new marking policy shall be used for egress packets
- Application explicitly called an API to delete NetFP object.
  Example: Fast Paths are shared across multiple NETFP clients. One NETFP Client wishes to delete a Fast Path and calls `Netfp_deleteOutboundFastPath` API

NetFP Server gets all notifications and generates events to the registered Clients. The NETFP Server

a) Allocates an internal event object. These are pre-allocated for each client.
b) Populates the event object with the event identifier and meta-information
c) Pushes the event to each registered and active NETFP client. This is done using an asynchronous JOSH job because the NETFP clients might not be immediately available to service the event and the NETFP server should not be blocked waiting on a slow client.

Each NETFP Client will eventually process the received event depending upon the event identifier. If needed, the modifications in the local socket structure are done in a non-locking fashion, using the active/shadow memory mechanism. Once a shadow area gets updated, atomic write switches the active/shadow areas, so next packet send will take update into actions.
The event may require (possibly, in addition to local socket data update) the client Application notification. The following table summarizes the various reasons and recommended behavior

Reason Id	**Description**
Netfp_Reason_FAST_PATH_DELETE	Fast Path has been deleted by a NETFP Client. The socket is no longer operational and should be closed.
Netfp_Reason_SA_DELETE	SA has been deleted by a NETFP Client. The socket is no longer operational and should be closed.
Netfp_Reason_SP_DELETE	SP has been deleted by a NETFP Client. The socket is no longer operational and should be closed.
Netfp_Reason_INTERFACE_DELETE	Interface has been deleted by a NETFP Client. The socket is no longer operational and should be closed.
Netfp_Reason_NEIGH_REACHABLE	The neighbor i.e. next hop MAC address has been resolved and the socket is now active and operational.
Netfp_Reason_NEIGH_UNREACHABLE	The neighbor i.e. next hop MAC address is no longer accessible and the socket is now no longer operational.
Netfp_Reason_MTU_CHANGE	Interface MTU has been modified. This is a pure notification event passed to the application which can use this to determine the maximum size of the payload

	to be sent out without fragmentation.
Netfp_Reason_IF_DOWN	Interface has been marked as down. This is a notification that the socket is no longer operational
Netfp_Reason_IF_UP	Interface has been marked as up. This is a notification that the socket is operational.
Netfp_Reason_SP_UPDATE	This is applicable for IPSEC sockets and is a notification to the socket that the SA has been rekeyed.

6.3.8.1 Route Update Operation

Diagram below explains the Route update process. Initiated by Application calling explicit Recalculate Route API, the notification is delivered to NetFP Server by NetFP Proxy.
NetFP server instructs Proxy to recalculate all active routes. Upon successful resolution, the updates on outbound interface changes are delivered to NetFP clients, which update affected sockets and notify affected Fast Path Applications. See the diagram below:

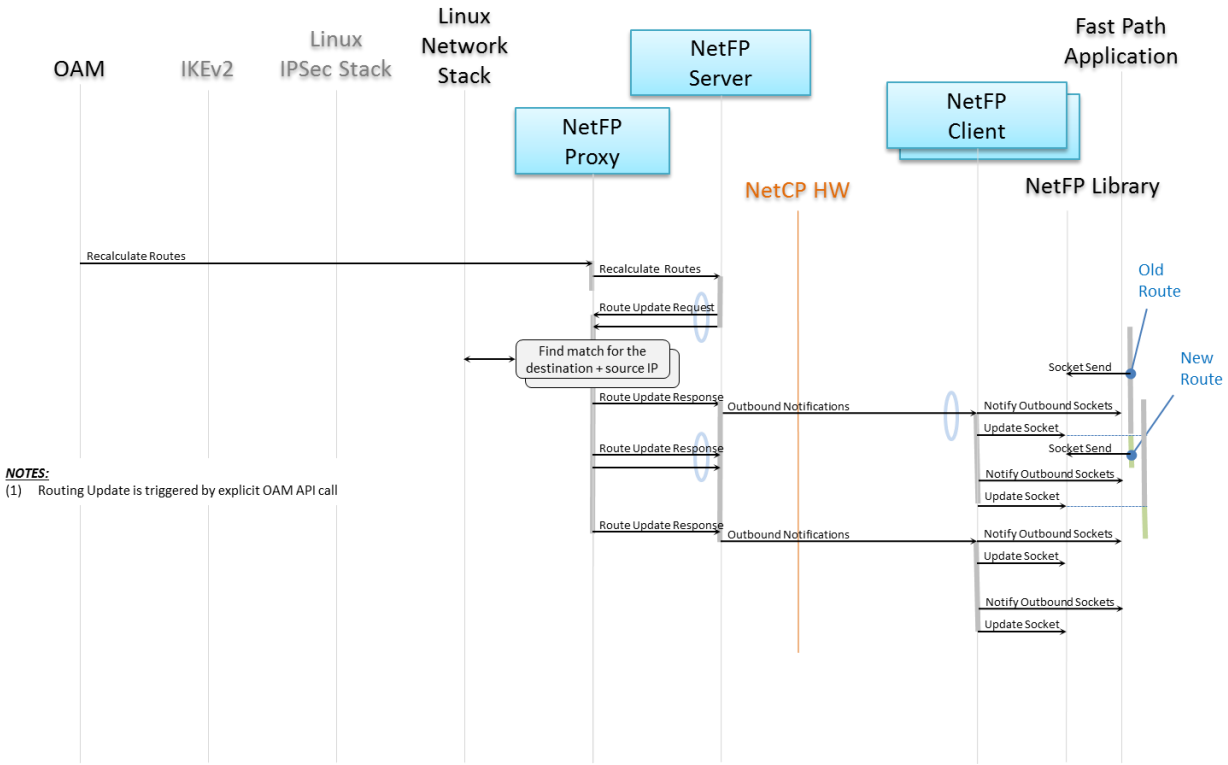


Figure 6-3. Route Update Operation

6.3.9 Re-Keying Operation

Diagram below illustrates the re-keying operation.

Once policy gets offloaded, NetFP proxy requests NetFP server to program LUT1-1 with ChildSA context. It also allocates a virtual link. At fast path creation, the LUT1-2 entry will be programmed with that virtual link to LUT1-1.

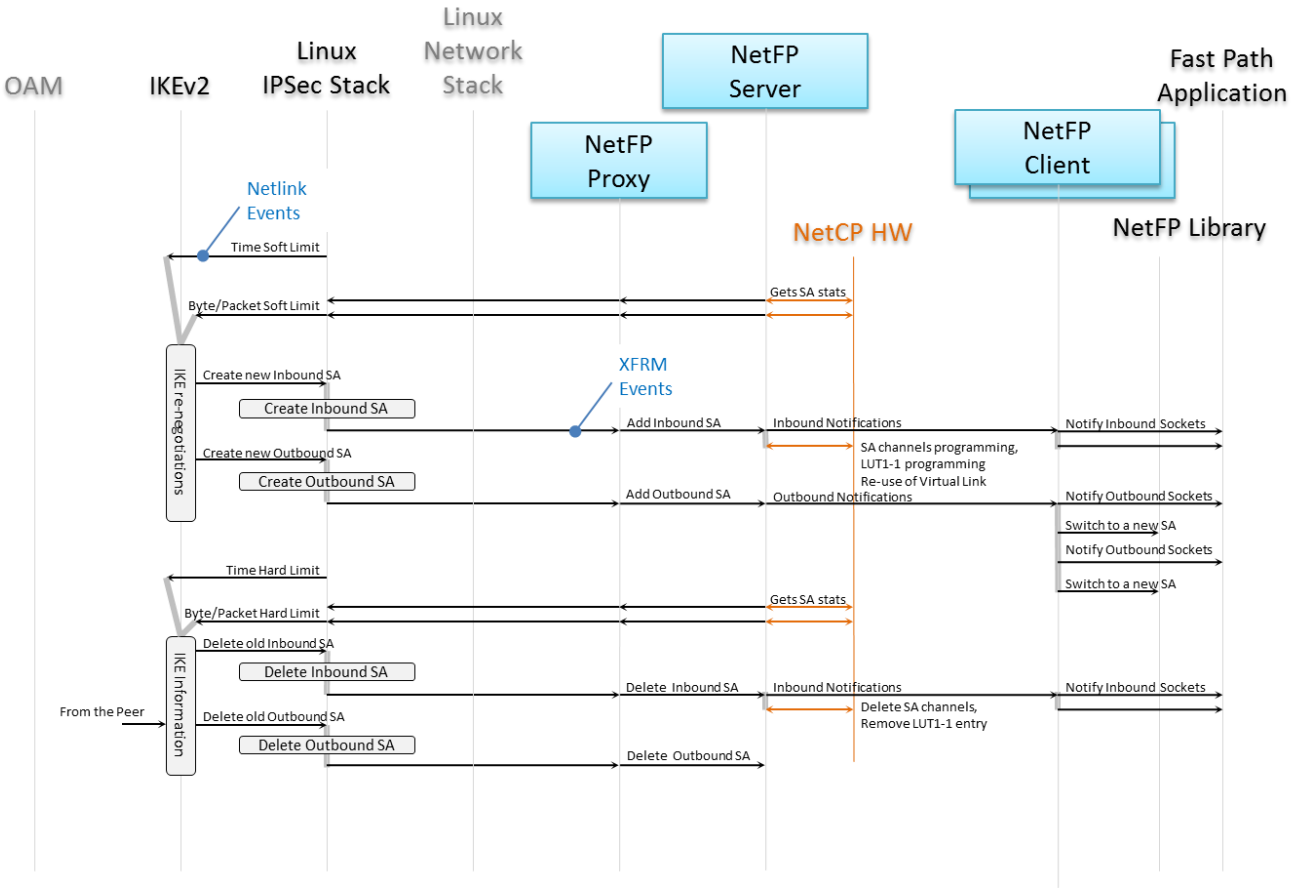


Figure 6-4. Re-Keying Operation

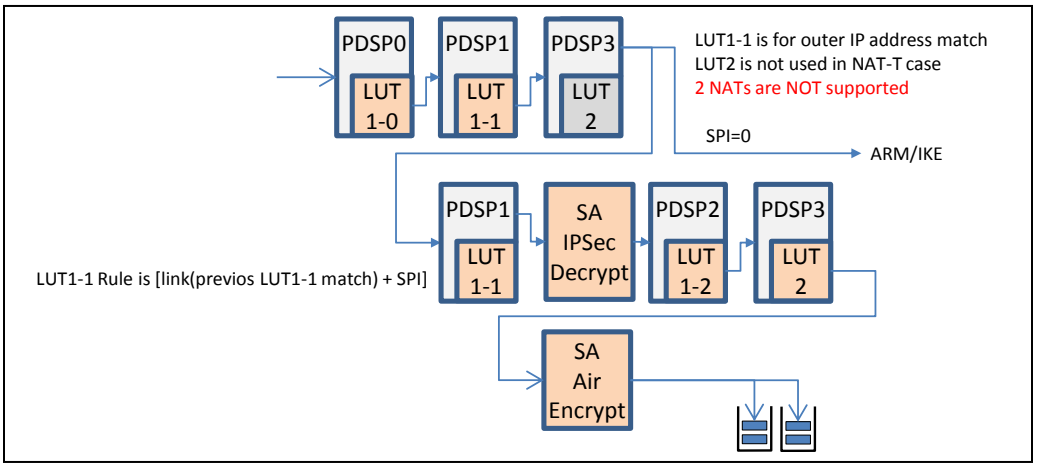
Rekeying for IPSec tunnel is done based on time, byte limit, or packet limit.
For time based rekeying Linux IPSec stack triggers a Netlink event to StrongSwan IKEv2 daemon. For rekeying based on byte limit or packet limit, NetFP server periodically collects the statistics from offloaded SAs and sends it to Netfp Proxy. NetFP proxy updates kernel IPSec stack. Once limits reached, IPSec stack generates a Netlink event to IKEv2 daemon.
Once IKEv2 daemon detects SAs soft limit (based on either timer, byte or packet count) reached, the rekey process is triggered. StrongSwan negotiates new key material and generates an event to IPSecMgr. IPSecMgr calls addSA API with an indication that this is a rekey operation. NetFP Proxy calls Netfp_rekeySA (and not a Netfp_addSA) requesting NetFP server to create a new childSAs both Ingress and Egress with renegotiated context.
NetFP server programs Ingress ChildSA to use the same virtual link. Ingress traffic can arrive on either of the 2 tunnels.
To switch relevant fast paths to use new ChildSA on egress, NetFP server generates an event, and this event is delivered to all relevant clients. The application is expected to execute

provided NetFP Event Handler, which shall update egress channels/sockets to use new SPI. The security policy is now switched to use the new SA on egress.
The old ChildSA are alive until a delete SA event is received on a hard timeout or on delete event using informational message from the peer.
If a hard timeout is received on old tunnels (either via Linux maintained timer or because NetFP Proxy updated stats that meet hard packet/byte limits), Linux IPSec stack will generate an event to IKEv2 daemon, which will in turn generate IKE informational message to notify peer of tunnel going down. Also the Linux stack deletes the SA context, which will in turn generate a delete SA event to Proxy/IPSecMgr. Netfp Proxy then calls Netfp_delSA() to propagate the same to NetFP. NETFP server will clean up LUT1-1 programming for old SA and will propagate the event to all policies using the old SA (Netfp_EventId_DELETE_SP event generated to all FastPaths still using old SA). Since we updated during Netfp_rekeySA() all policies to start using new SA, no fast paths are affected when old SA gets deleted as long as the new SA is still operational.

6.3.10 NAT-T support

IPsec uses NAT traversal(NAT-T) in order to have encapsulating Security payload packets traverse NAT. If NAT-T is enabled, NetFP has support to handle egress/ingress packets with UDP encapsulation.

Diagram below illustrates NetCP support for NAT-T ingress operation.



In NAT-T mode, LUT1-1 is programmed to match outer IP address (secure gateway IP address) with UDP as next protocol to forward packets to PDSP3 to NAT-T handling. PDSP3 supports SPI=0 (IKE) and keep alive messages, they are forward to Linux through interface based routing rules. UDP encapsulated data packets will be forwarded back to PDSP1 to further IPsec processing for packets matches SPI entries. Support to match rule with both SPI and previous link is not currently supported and will be supported in future releases.
NAT-T can be enabled on NetFP master during master starts up. NAT-T condition is detected at runtime by IPsec component(such as Strongswan) and UDP encapsulation information is

forwarded to NetFP proxy when creating SA. These information will be propagated to the sockets that uses the corresponding secure policy during socket connect time.

6.4 QOS

The NETFP module exposes a Quality of Service framework which allows shapers to be configured, egress packets to be marked and mapped into desired QoS channels. The module spans the Linux Kernel, NETFP Master and the NETFP subsystem.
NetFP module supports hierarchical shaping, where egress packets can optionally pass through two layers of shaping: Layer 3 and Layer 2 shapers.
Policing and shaping is performed by dedicated RISC processor(s) (PDSPs) downloaded and configured by Linux Kernel. Application is expected to define deployment specific shaper configurations using DTS files. Syslib ships with a sample QoS configuration, providing support for one layer3 and two layer2 shapers. Layer 3 shaper configuration tree is loaded to one PDSP, while one dual Layer2 tree (to illustrate separate shaping per egress port) is loaded into another PDSP. Please refer to the sample DTS files present in the `SYSLIB_INSTALL_PATH/ti/runtime/resmgr/dts/<device>` for more information. Please refer to the MCSDK documentation for more information on the QOS configuration.

Before NetFP Master starts, Layer 2 shaper is disabled. Linux can send packets directly towards the egress port, or I can use tc commands to Layer 3 shape the outgoing traffic.

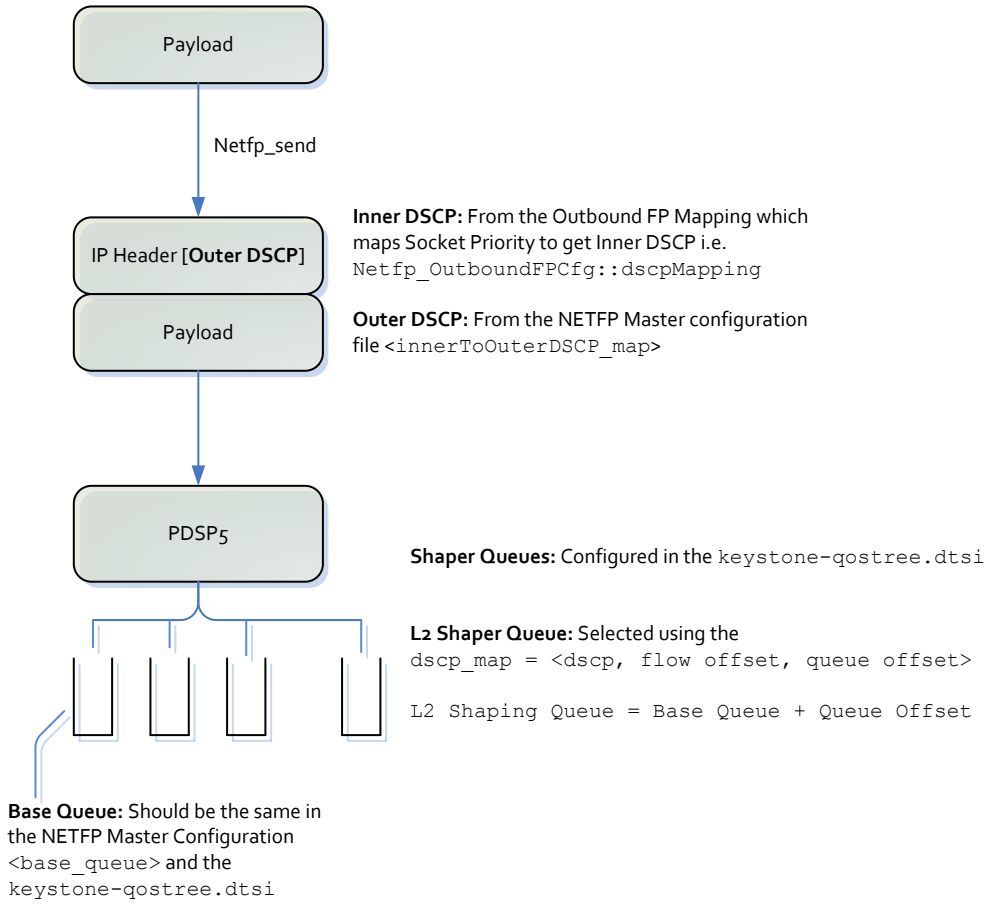
The NetFP Master is responsible for enabling and configuring Layer 2 shaper, and hooking up the NETCP subsystem to use these shapers. This requires the kernel DTS configuration and the master configuration to be synchronized. Failure to do so will result in the packets not getting sent out of the box.

The L2 Shaper is considered to be a **switch** shaper and is configured on per egress port. If enabled, it is expected that all egress traffic irrespective of where it is originating from (Linux or Fast path) is shaped. The shaping is done by NETCP by looking at the DSCP/VLAN Priority bits markings in the packet. Once packet is properly marked, no software intervention is required. The shaping is done **without** the need for any additional `"tc"` commands in Linux. Similarly there is no additional configuration required in the fast path.

The L3 shaper is an *optional* shaper which can be configured in the DTS file. To enable L3 shaper in a fast path, NetFP API `Netfp_setupL3Shaper` shall be called on per interface base. On Linux `"tc"` commands need to be added to use this shaper.

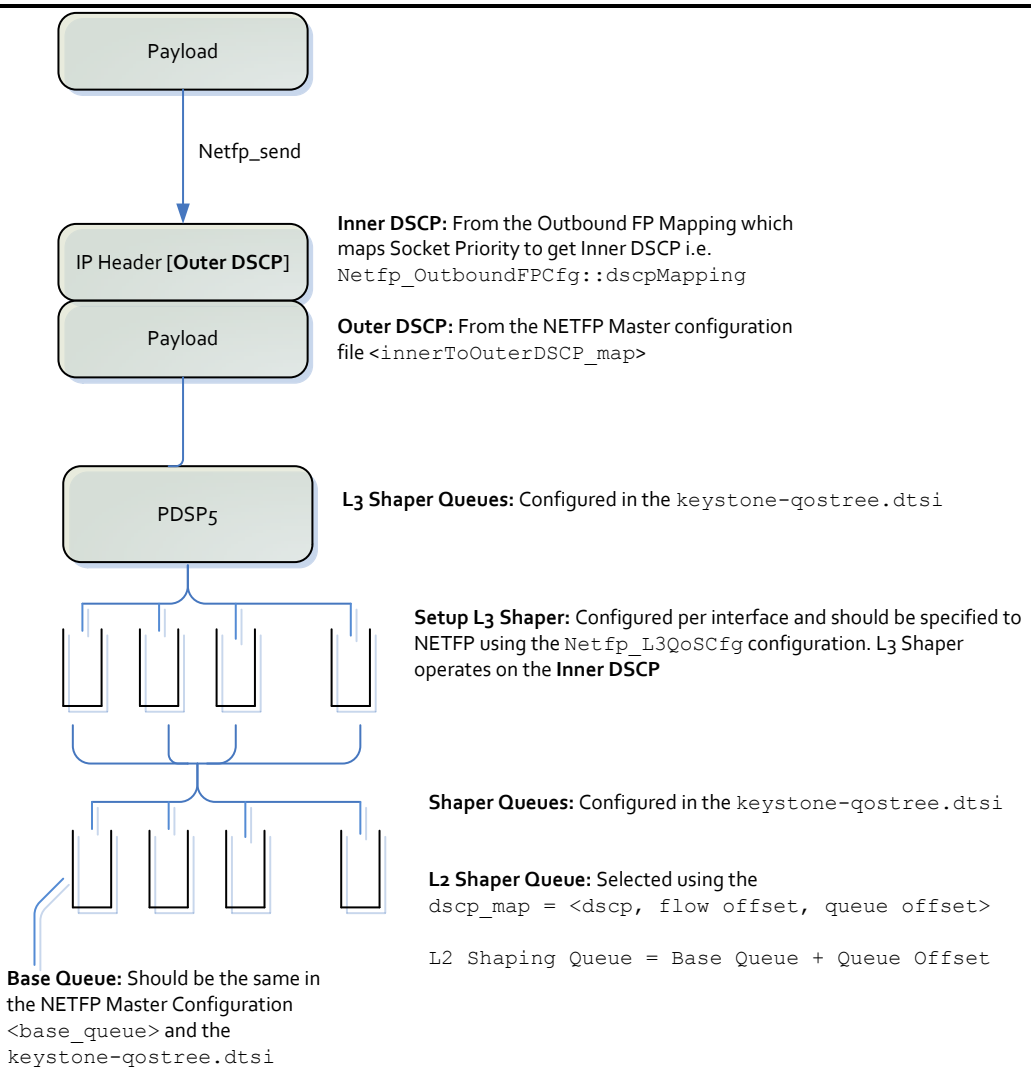
6.4.1 Non Secure L2 Shaper only

The following figure shows the flow of a non-secure packet and the operations done on the packet with respect to the DSCP and the selection of the shaper queues.



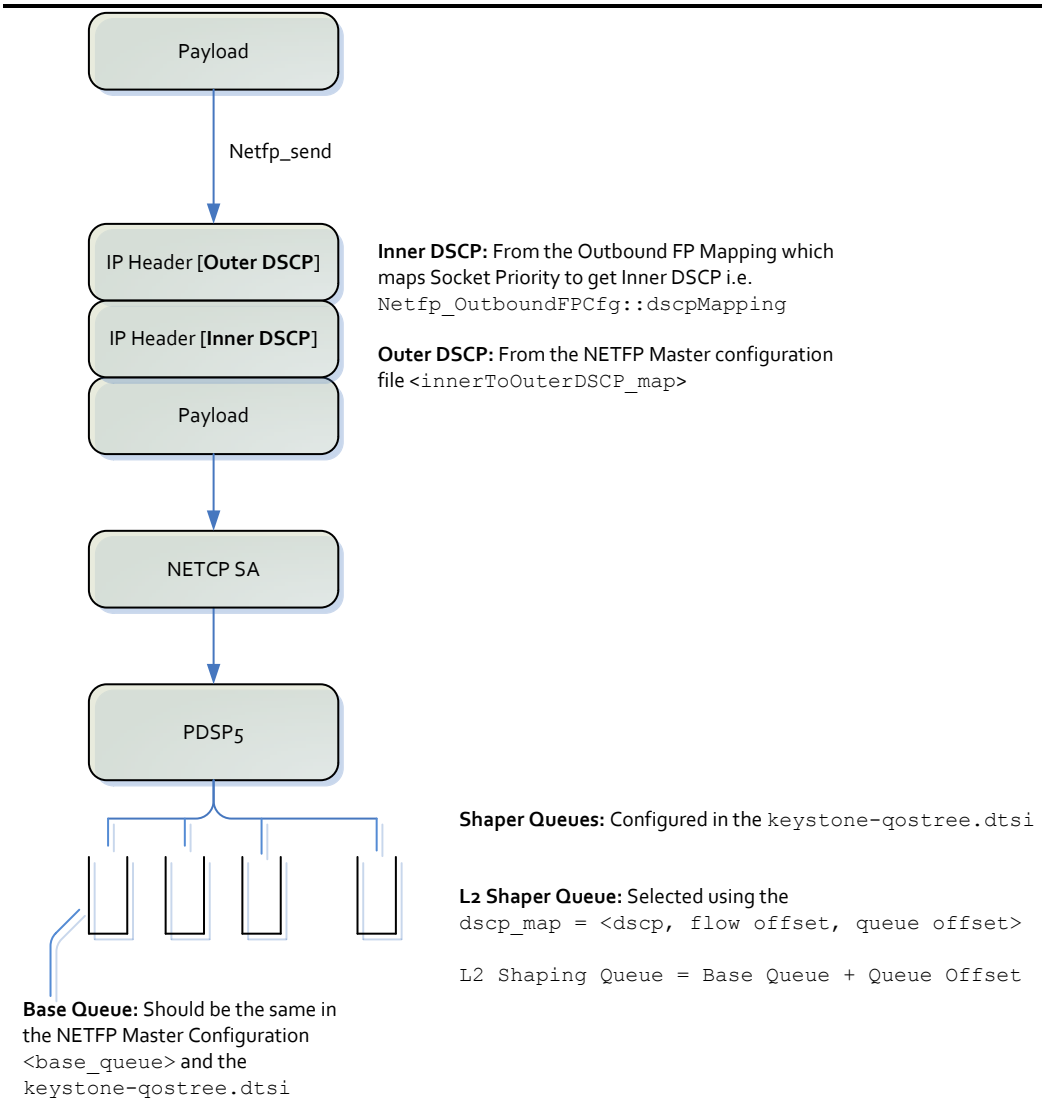
6.4.2 Non Secure L2 and L3 Shaper

The following figure shows the flow of a non-secure packet and the operations done on the packet with respect to the DSCP and the selection of the L2 and L3 shaper queues.



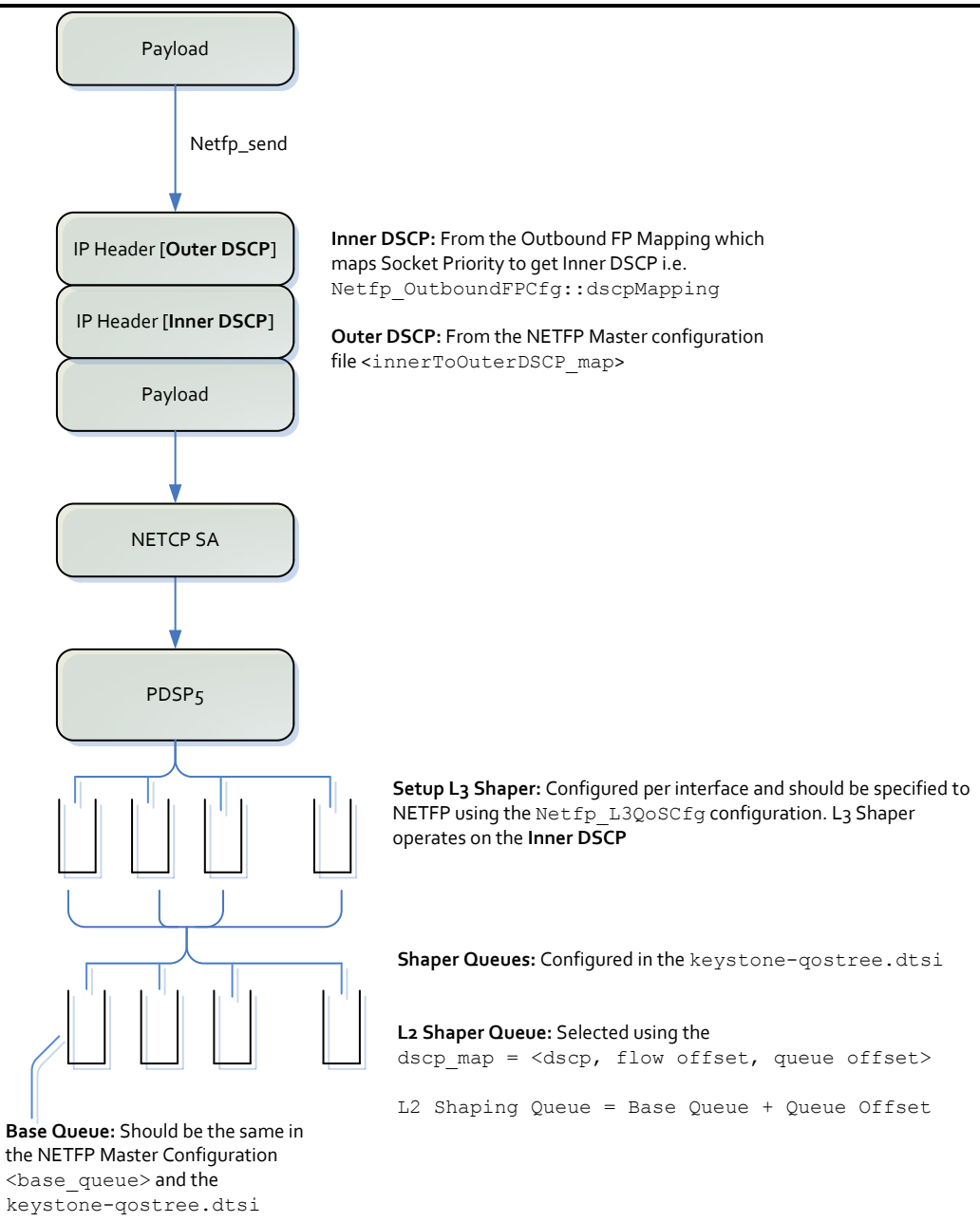
6.4.3 Secure L2 Shaper only

The following figure shows the flow of a secure packet and the operations done on the packet with respect to the DSCP and the selection of the shaper queues.



6.4.4 Secure L2 and L3 Shaper

The following figure shows the flow of a secure packet and the operations done on the packet with respect to the DSCP and the selection of the L2 and L3 shaper queues.



6.4.5 Handling of untagged, non-IP packets

L2 QoS Shaper can be configured to operate in either DSCP or DSCP-PCP mode.
In DSCP-PCP mode:

- VLAN Tagged packets are shaped based on VLAN TCI field value, using the mapping configured in vlan_map of egress port,
- Untagged IP packets are shaped based on DSCP value, using the mapping configured in dscp_map of egress port,
- Untagged, non-IP packets are handled based on rules below

In DSCP mode:

- IP packets are shaped based on DSCP value, using the mapping configured in dscp_map of egress port,
- Non-IP packets are handled based on the rules below

The non-IP packets, generated by Linux Host shall be also sent to L2 QoS shaper. NETCP PDSP5 will determine the output L2 QoS channel and a flow by looking into the `vlan_map` of the egress port based on the global `default_host_priority` value programmed by NetFP Master. It is expected that casual non-IP traffic (like ARPs) will go that route.
In addition, for the special traffic (e.g. PTP/EthOAM), Linux Host may setup a DMA channel with a flow and destination directly into one of the input L2 QoS queues and use `tc` utility to send those packets into the L2 QoS directly.

For non-IP packets, forwarded by NETCP, L2 QoS channel is determined by NETCP PDSP0 based on per ingress port `default_forwarding_priority` value programmed by NetFP Master.

6.5 Reassembly

Reassembly operations can be either performed by NetFP Master (the recommended choice) or by any NETFP client.
When NetFP master is configured to run reassembly, NetFP master spawns a thread that performs all reassembly operations.

If application chooses not to run reassembly at NetFP master, it shall configure NetFP master to skip reassembly operations, and shall call NetFP Client APIs to register itself as a handler of the services. This is done using the `Netfp_registerReassemblyService` API. The NETFP client is then responsible for ensuring the reception and reassembly of all packets (destined to the DSP or ARM). This implies the following:

1. The NETFP client is responsible for providing a context and to invoke the `Netfp_reassembly` function.
2. The NETFP client is responsible for invoking the `Netfp_reassemblyTimerTick` function to handle the aging of reassembly contexts.

Failure to perform the above actions will result in the reassembly process to not function correctly. The NETFP reassembly procedure operates for all packets irrespective of their final destination (DSP Fast Path or Linux ARM networking stack).

6.5.1 Reassembly Operation Details

NetCP FW (PDSP1 for outer IP and PDSP2 for inner IP) are inspecting each incoming packet (frag or non-frag) to make a decision on whether packet shall be forwarded to reassembly SW, or allowed to proceed with HW packet classification

NetCP FW maintains up to 64 (32 for inner and 32 for outer IP) active reassembly flows, where flow is defined as (source IP + destination IP + proto). For any packet (frag or non-frag), NetCP FW checks whether packet belongs to any of the active flow. Packets not matching any of the active flows are forwarded to the HW classification engine. Packets that match any reassembly flow are sent to Reassembly SW.

NetCP FW also maintains a counter associated with every active flow. For every packet sent to Reassembly SW, the counter is incremented. When Reassembly SW returns [assembled/non-frag/recycled] packets back to the FW, it also returns the frag count used to decrement the flow's count.

Reassembly SW handles the packets in the order it receives them from NetCP FW. For efficiency reasons, the expectation is that the reassembly SW works on packets bursts using HW accumulated channel. Parameters (burst size, closure time) are user configurable.

If packet is a fragment, it checks whether it completes a re-assembled packet
    If yes, then newly reassembled packet is forwarded to NetCP FW.
    If no, packet is added into the under construction queue, and awaits for the rest of the fragments to arrive
Else, non-frag are sent back to NetCP FW immediately

6.5.1.1 Handling of DoS frag attack

Reassembly SW implements simple default traffic management procedure and allows for an application to plug-in the custom one. In default implementation, we are checking for the user configurable buffer threshold. If we are running low, we recycle the oldest frags until we get enough buffers back. It is quite simple traffic management procedure, and it may not be adequate, depends on customer definition of DoS protection. The procedure ensures that the eNB is alive, and non-frags are passing normally. The system will combat the offender, proactively dropping his frags, prior to getting into any system memory issues. However, there could be "normal/good" packets dropped (the ones that are ageing during the attack). The buffer thresholds are configurable by application.

6.5.1.2 Handling of Reassembly Timeouts

Application configures the timeout (maximum time to live) for the packets under construction. Once timeout is reached, all frags from the packet under construction queue are recycled, and NetFP FW is informed on the number of frags recycled.

6.5.1.3 Handling of Large Reassembled Packets

Application developers should be aware of the following key points:

1. NETCP HW can handle packets less than 9K.
2. NETFP SW Reassembly module detects that the reassembled packet is greater than 9K. We now know that we cannot pass it back to the NETCP to continue packet lookups so we have 2 options:
    a. Drop the packet in the NETFP reassembly module
    b. Pass the packet to the application

Option a) can be achieved by setting the large packet channel in the reassembly configuration to NULL.

For Option b), the NETFP module can pass the packet to the application; where the application could do some operation on these large packets. (Maybe log some errors using the packet fields etc.) In order to pass the packet to the application; we now have 2 transfer modes:

a. **Queue Mode**: In this mode the chained packets (from the reassembly heap) are passed to the large packet channel. However if the application is not polling or servicing this queue fast enough then we can starve the reassembly heap and can drop fragments.

b. **Queue-DMA mode**: In this mode the chained packets from the reassembly heap are DMA into descriptors from another heap (Say Large Packet Heap). Since the packets from the reassembly heap are immediately cleaned up there is no starvation/dropping of fragments. The large packet channel can be serviced now at its pace.

Applications which will simply drop the large packets can select Option a). There is a counter called "`numLargePackets`" in the Reassembly MCB which tracks the number of large packets received.
Applications which desire to perform any application specific operations on these large packets would need to consider Option b). The servicing rate of the large packet channel can decide between Queue and Queue-DMA.

6.6 Fragmentation

Data is transmitted to the network via sockets or 3GPP channels. The NETFP services ensure that large payload sizes of greater than the interface MTU, will be fragmented at the IP layer as per the IPv4/IPv6 fragmentation specification.
Application can be unaware of the fragmentation process; the operation is implemented internally within the `Netfp_send` API.

6.6.1 Fragmentation Operation Details

Fragmentation process depends on whether packet is originated from Linux or Fast Path, whether packet shall be encrypted or not. In addition, there is a dependency on packet length.

6.6.1.1 For Fast Path originated traffic

In secure case:

- For the large packets of more than 9600 bytes length, inner fragmentation is always performed by NetFP SW
- Else,

  If inner fragmentation option is selected (this option can be selected per ChildSA), NetFP SW does inner-IP fragmentation. The fragmentation is done in such way that it guarantees there will be no post crypto outer IP fragmentation (i.e. IPSec headers/footers are deducted from the interface MTU size). Packets are sent to NetCP. NetCP encrypts, sends packets to the shaper(s), and then to the switch port.

  If outer fragmentation option is selected, inner IP packets (potentially larger than interface MTU size) are sent to NetCP for encryption. Post encryption, if needed, NetCP FW fragments outer IP packets to interface MTU size, sends the packets to shaper(s), and then to the switch port.

In non-secure case:

- For the packets larger than 9600 bytes, fragmentation is performed by NetFP SW
- Else, Packets are sent to NetCP. if needed, NetCP FW fragments outer IP packets to interface MTU size, sends the packets to shaper(s), and then to the switch port.

6.6.1.2 For Linux originated egress traffic

In secure offloaded case

- Linux stack does inner-IP fragmentation. The fragmentation is done in such way that it guarantees there will be no post crypto outer IP fragmentation (i.e. IPSec headers/trailers are deducted from the path MTU size). Packets are sent to NetCP. NetCP encrypts, sends packets to the shaper(s) and then to the switch port.

In secure non-offloaded case

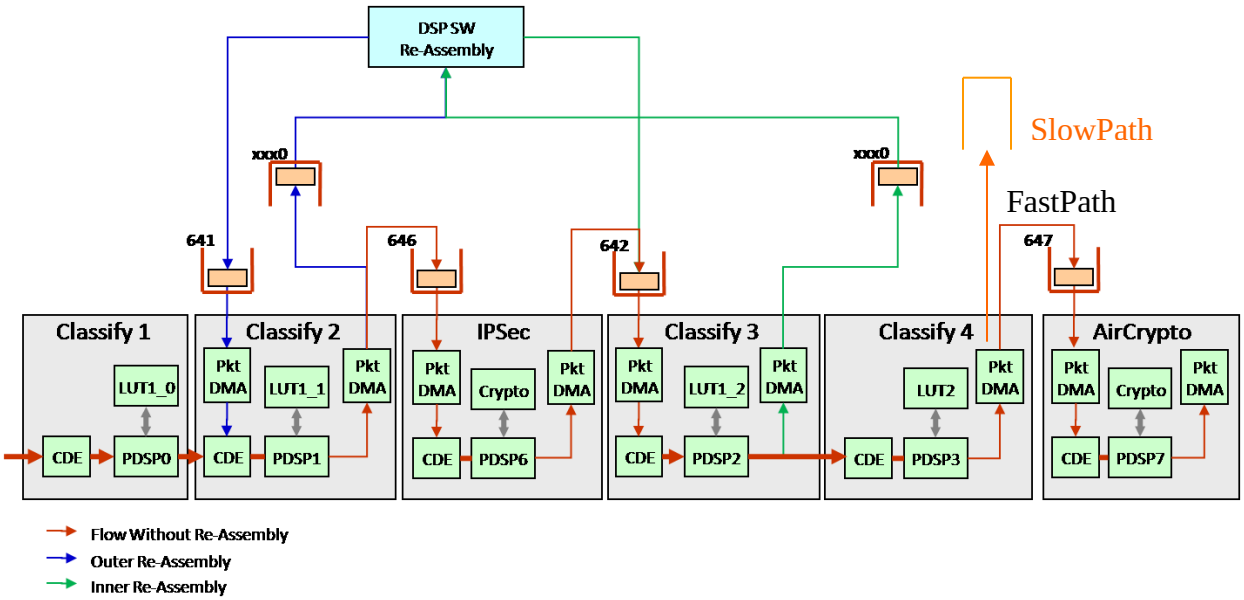
- Linux first encrypts, then does outer-IP fragmentation, then sends packets to the shaper(s) and then to the switch port.

In non-secure mode

- Linux does IP fragmentation and sends packets to the shaper(s) and then to the switch port.

7 LTE Specific operations

The NETFP module exposes a set of the LTE services which can be used by the LTE stack developers to integrate and utilize the NETCP subsystem. The following figure is an illustration on how the NETCP subsystem could be used:-



Data received in the downlink direction is passed to the IPSEC decode engine. Once the packet has been decoded the GTPU identifier is matched. The packet can then be passed directly to the AIRCRYPTO engine for ciphering (**Fast path**) which implies that the application receives the ciphered packet. The application can also be configured to pass the plain text packet before AIRCYPTO (**Slow path**); in which case the application is responsible for packet encoding.

7.1 GTPU Control message

The NETFP client allows an application to register handling GTPU control messages via the `Netfp_configureGTPUControlMessage` API. The NETCP subsystem will pass the received

messages to the application supplied queues where they can be received and processed by the application.

The following GTPU message types supported

- GTPU ping request – Message type = 1
- GTPU ping response – Message type = 2
- GTPU error indication – Message type = 26.
- GTPU header notify – Message type = 31
- GTPU End Marker – Message type = 254
- GTPU parsing error or unsupported message type – Any message not listed below or is not 0xFF.
- GTPU Id mismatch – Message type = 0xFF but the radio bearer for the GTPU ID is not added.

7.2 LTE User

For every user, a NETFP user security context MUST be added. The user context specifies the following properties

- Authentication and ciphering mode.
- Authentication and ciphering keys.
- User equipment identifier.
- Flow used to output control PDUs.
- Destination queue where the output control PDUs are available.

The API `Netfp_createUser` also creates and configures secure channels used by single user for signaling radio bearers: SRB1, SRB2.

Note: Please ensure that the SRB flow identifier has 4 extra bytes so that the NETCP subsystem can insert the MAC-I.

7.3 SRB

7.3.1 Encode

The API integrity protects and ciphers the control plane PDU in the downlink path. The API performs F9 and F8 on the PDU as per the channel configuration. The format of input and output PDUs is as below:

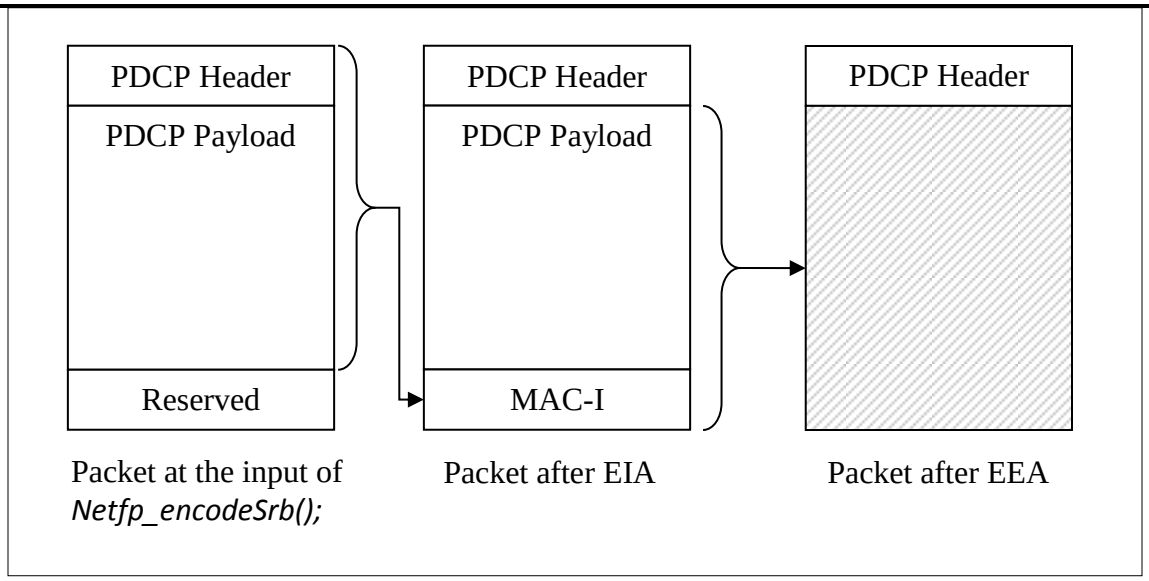


Figure 7-1. Control Plane encoding Operations

Input parameters to the API are

- Packet descriptor that meets the following requirements:
  - The payload pointed by the descriptor shall include fully formed single byte PDCP header
  - The packet descriptor's data pointer must point to the PDCP header
  - The data buffer specified by the `srbFlowHandle`(when the user context was created) MUST have 4 extra bytes so the SA can insert the MAC-I.
  - PS info and EPIB will be overwritten in order to pass commands to the security accelerator
  - Descriptor will be recycled by SA as per the return policy configured in the descriptor
- 32 bit Count-C value
- 8 bit Bearer ID. Valid numbers are: 1 for SRB1, 2 for SRB2

Note: When Snow3G authentication or ciphering is configured, the software library is used to perform the F9 and F8 operation. Since the packet is not sent to SA, `srbFlowHandle` is not used to output the resulting packet. Instead the input packet is modified. The input data buffer MUST have 4 extra bytes to insert MAC-I.

The encoded packet will be placed into the corresponding SRB encode queue.

7.3.2 Decode

The API ciphers the control PDU in the uplink path. The API performs F8 and F9 on the PDU as per the channel configuration.

The format of input and output PDUs is as below:

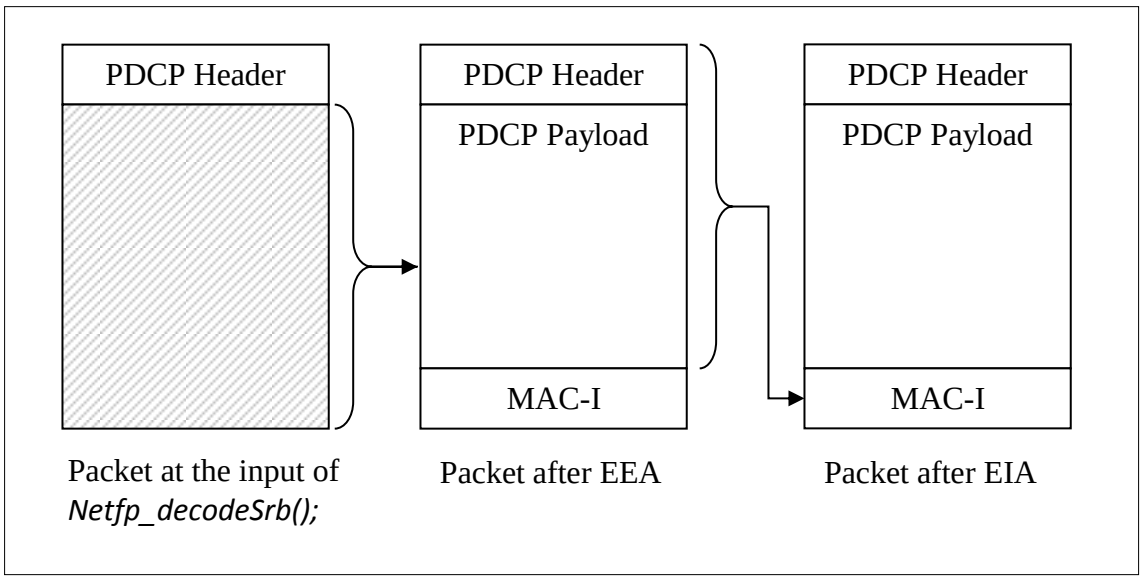


Figure 7-2. Control Plane decoding operations

Input parameters to the API are

- Packet descriptor that meets the following requirements:
  - A 1-byte PDCP header with a valid sequence number must be present in the PDU before calling the API
  - Packet descriptor's data pointer must point to the PDCP header
  - PS info and EPIB will be overwritten in order to pass commands to the security accelerator.
  - Descriptor will be recycled by SA as per the return policy configured in the descriptor

- 32 bit Count-C value
- 8 bit Bearer ID. Valid numbers are: 1 for SRB1, 2 for SRB2

The decoded packet will be placed into the corresponding SRB encode queue. The output PDU will still contain the PDCP header and the 4 byte MAC-I. The check for MAC-I validation errors use API `Netfp_getPacketError`.

7.4 DRB

Data radio bearers are added to the user security context using `Netfp_createLTEChannel` API.LTE channels are bidirectional which implies that the channel is able to send & receive data.

This implies that while creating a channel the "**bind**" and "**connect**" properties both need to be specified.

To receive packets on the channel, the **bind** configuration must be specified.

- Ingress Fast path handle.
- Ingress Data Radio Bearer configuration information and the user security context handle.
  Or
- Source port number

To send packets on the channel, the following **connect** configuration must be specified.

- Egress Fast path handle.
- Egress Data Radio Bearer configuration information and the user security context handle.
  Or
- Destination port number

For radio bearer channels, the security accelerator is programmed with the given ciphering parameters to encrypt and decrypt depending on the direction.

In the "Ingress direction" data radio bearers can operate in the fast path or slow path mode:-

- **Slow Path Mode:** Packets matching LUT2 will be received by the NETCP and will be placed into the DRB ROHC queue (should be considered as the slow path queue). The application is responsible for picking packets from the specific queue and encoding them using the NETFP provided API.

- **Fast Path Mode:** Packets matching LUT2 will be forwarded to the NETCP for ciphering and the ciphered message containing `countC` will be placed into the encoded channel. Applications will thus receive a ciphered message directly into their encoded queue.

7.4.1 Encode

This API ciphers the user plane PDU in the downlink path in standalone (as opposed to fast path) mode. The API performs EEA operation on the PDU as per the channel configuration. Note that this operation may be performed by NETCP as a part of fast path processing.

This API will be typically called by an application when fast path exception case occurs (i.e. active ROHC non-profile0 operations on radio bearer). When the fast path is enabled ROHC is set to zero by the security accelerator. If ROHC has to be set to a non-zero value, fast path must be disabled.

The format of input and output PDU is as below:

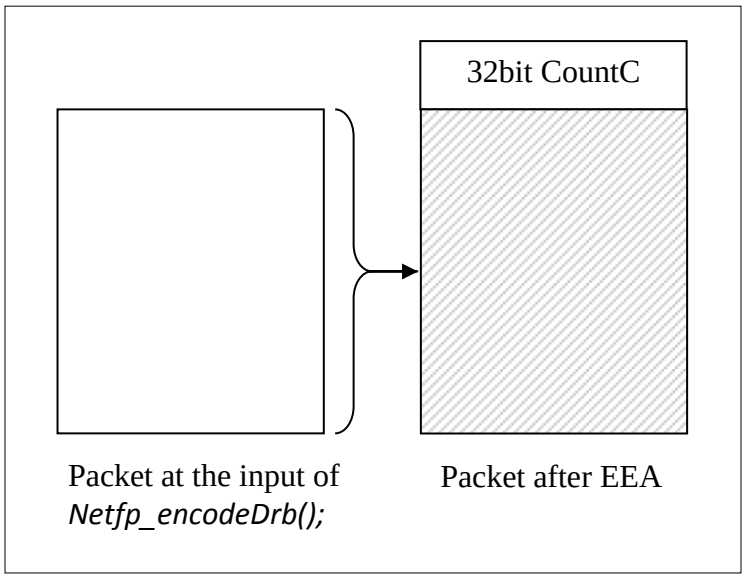


Figure 7-3. User plane encoding operations

Input parameters to the API are:

- Packet descriptor that meets the following requirements:
  - Packet descriptor's data pointer must point to the first message byte to be ciphered
  - Descriptor will be recycled by SA as per the return policy configured in the descriptor. This implies that the application should NOT attempt to free the packet
- 8 bit Bearer ID.

The NETCP will place the encoded packet into the encoded channel queue and will insert the 4 bytes of `countC` used at the start of the data buffer.

7.4.2 Decode

The API ciphers the data PDU in the uplink path. The API performs F8 operation on the PDU as per the channel configuration. The format of input and output PDU is as below:

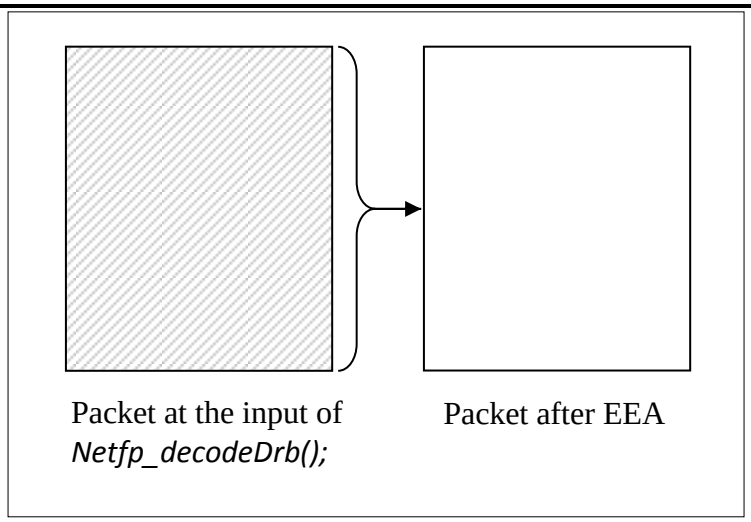


Figure 7-4. User plane decoding operations

Input parameters to the API are:

- Packet descriptor that meets the following requirements:
  - Packet descriptor's data pointer must point to the first message byte to be ciphered
  - Descriptor will be recycled by SA as per the return policy configured in the descriptor
- 32 bit Count-C value
- 8 bit Bearer ID

The `countC` value used is output by SA. It can be obtained by the `Netfp_getCountC` API. The decoded message is placed into the decoded channel.

7.5 De-multiplexing radio bearers

In order to de-multiplex the packets received on the same MSGCOM channel for different SRBs and DRBs, NETFP provides the `Netfp_getPacketId` API. The function decodes the packet and returns the following:

- User equipment Id
- QCI
- Bearer ID

7.6 Ciphering

An Additional API to cipher or decipher data is provided when standard `Netfp_encode/decode` SRB/DRB APIs are not sufficient. The parameters that can be overwritten are

- Type of operation – cipher or decipher
- CountC
- Direction
- Bearer Id
- Destination queue in which output is available

The following table depicts how the type of operation along with direction is used.

Type of operation	Direction	Action
Cipher	1	Normal DL operation. Packets are ciphered by eNB to sent to UE
De-cipher	0	Normal UL operation. Packets that were ciphered by UE are received by eNB and deciphered.
Cipher	0	Used for testing. This combination is used when eNB is trying to generate a ciphered packet to simulate the UE. Since the direction used by UE to cipher would have been 0, the direction used by eNB MUST be also 0.
De-cipher	1	eNB loopback deciphering operation. This combination is used when eNB is trying to de-cipher a packet that it had ciphered itself. Since the direction when ciphering is 1(see row 1), the direction used when de-ciphering MUST be also 1.

7.7 Generating MACI

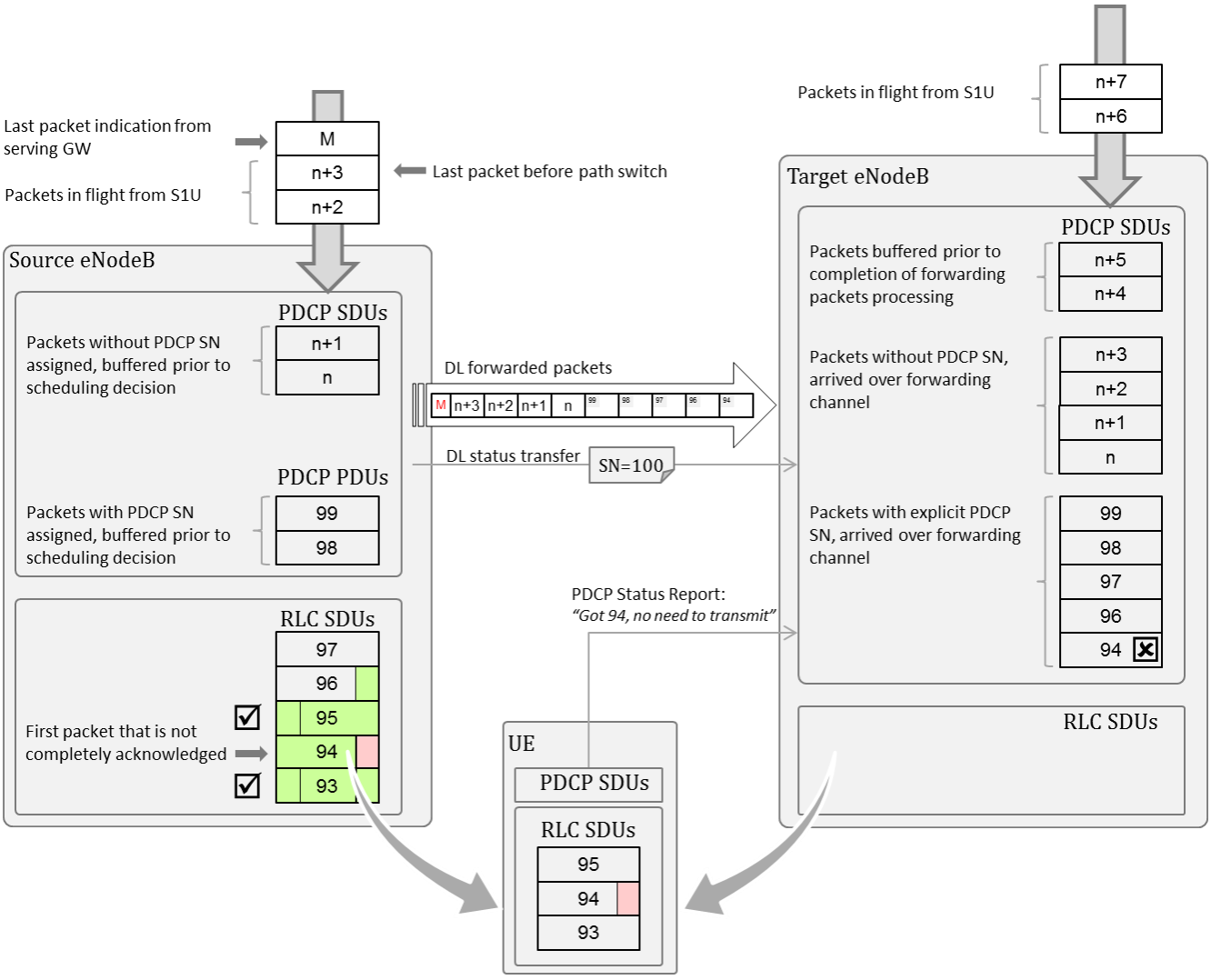
The NETFP module exposes an API to generate MACI (`Netfp_generateMacI`). The parameters that can be overwritten are

- CountC
- Direction
- Bearer Id
- Destination queue in which output is available

7.8 Handover Procedure

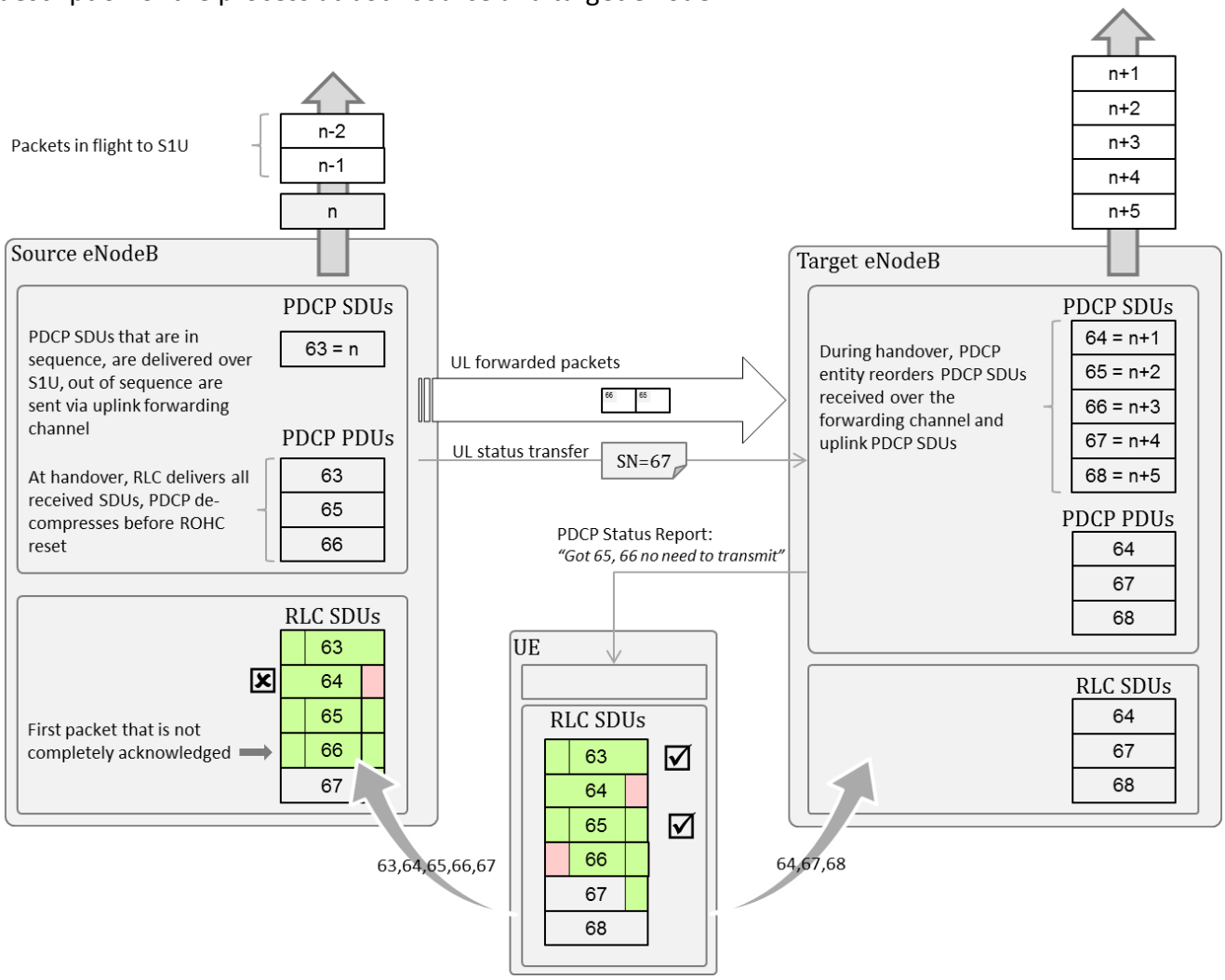
7.8.1 Handover procedures in Downlink

The picture below illustrates the downlink handover procedures details, followed by the description of the process at both source and target eNodeB.



7.8.2 Handover procedures in Uplink

The picture below illustrates the uplink handover procedures details, followed by the description of the process at both source and target eNodeB.



7.8.3 Handover Procedure at Source eNodeB

For the departing UE, the source eNodeB decides which of the data bearers are subject for forwarding of downlink and optionally uplink data packets from the source to the target eNodeB.
Once the transport addresses are known to eNodeB, application creates "send only" GTPU sockets for downlink and uplink packet forwarding.

7.8.3.1 Forwarding downlink packets at Source eNodeB

For each of data bearers that downlink forwarding is requested, application initiates source handover procedure by calling the `Netfp_initiateSourceHandOver` API. After this API is called, all S1U packets are starting getting buffered in the application defined source handover queue.

To achieve lossless handover, application may need to process already ciphered PDCP PDUs and re-create clear text PDCP SDUs. Those PDCP PDUs can be:

- Non fully acknowledged RLC SDUs (example, 94, 97, and 97), and/or
- Buffered PDCP PDUs (example, 98 and 99).

Application developers need to create a **forwarding socket** which is used to send all buffered packets from the Source eNB to the Target eNB. From the source eNB this socket is a send-only UDP socket. This will allow applications to add the GTPU extension headers to any packets sent on this socket.

To de-cipher such packets, application shall call `Netfp_cipher` APIs, indicating explicit COUNTC value and downlink direction. Application shall then restore original PDCP SDU, place the PDCP sequence number into the GTPU extension header and send the packets over forwarding socket using `Netfp_send` API.

Application shall then send all packets from source handover queue (the packets without assigned PDCP SN) over the downlink forwarding socket. Those packets shall be send without GTPU extension header using `Netfp_send` API (packets n, and (n+1) in our example).

Once all packets form S1U are handled, application shall then send the rest of packets (example, (n+2), and (n+3)), followed by the end marker indication over the same forwarding channel.

In status transfer message, application shall indicate the first sequence number that shall be used at target eNodeB for downlink packets. It shall be number PDCP SN=100, in our example.

Once marker is sent, application can close and delete the forwarding socket.

7.8.3.2 Forwarding uplink packets at Source eNodeB

To achieve lossless handover, application shall deliver all in sequence PDCP SDUs to S1U. In addition, all out of sequence PDCP SDUs shall be sent over the uplink **forwarding socket**, with PDCP SN indicated in GTPU header extension. Those packets are sent using `Netfp_send` API. Once all packets are sent, the application can close and delete the uplink forwarding channel. In status transfer message, application shall indicate the sequence numbers of last uplink forwarded packet.

7.8.4 Handover Procedures at Target eNodeB

For the arriving UE, the procedures to handle downlink and uplink data bearers with forwarding channels are described below.

7.8.4.1 Downlink Procedures at Target eNodeB

There are few steps required to implement handover procedure at target eNodeB.

Step 1. Creating the user

Once UE arrives to the target eNodeB, application creates a new user by calling `Netfp_createUser` API. Using the key materials provided, this API instantiates security context for authentication and ciphering operations over signaling radio bearers, and creates both SRB1 and SRB2 channels.

Step 2. Creating the data radio bearers

When UE arrives to the target eNodeB, downlink and optionally uplink data for a set of data radio bearers can be forwarded to it from the source eNodeB.

2.1 For each data bearer that expects downlink forwarding data packets, application shall create "receive only" UDP socket to buffer those packets by calling `Netfp_socket/Netfp_connect/Netfp_bind` API sequence.

2.2 For each data radio bearer, application shall create an LTE channel by calling `Netfp_createLTEChannel` API. This API constructs all necessary security artifacts to handle ciphering operations for that data bearer. Application can specify if the LTE channels are to be created in Fast Path or non-Fast path mode.

Note: For each data bearer that is being handed over, application shall set the flag '`isHOInProgress`' in the bind configuration to indicate that this LTE channel is initialized in the Handover mode. The LTE channel in Handover mode will buffer all the GTPU packets arriving on the S1U link into an internal queue. Those packets shall be processed only after all forwarded packets are handled.

Step 3. Processing downlink forwarded packets

To ensure correct packet ordering in the downlink, application needs to process all packets forwarded from the source eNodeB. These packets will be placed in the queue specified in the "receive only" UDP socket bind configuration. The packets shall be popped from the queue. Applications are responsible for processing the GTPU Header. Application shall cipher each

packet using `Netfp_encodeDRB` API. Application shall provide explicit COUNTC number to be used by the Netfp Library. The value of COUNTC shall be derived either from the GTPU extension header for the packets (packets 94, 96, 97, 98, and 99 in our example), or incremented based on PDCP SN received in DL status transfer message (example, PDCP SN = 100). Therefore, in our example, application shall be calling `Netfp_encodeDRB` API for packets n, (n+1), (n+2), and (n+3) with COUNTC = 100, 101, 102, and 103.
Application shall continue processing forwarded packets until it gets the last packet indicator (or times-out in the absence of it). Application shall now delete the forwarding sockets calling `Netfp_closeSocket` API.

Step4. Completing handover process

For each of the bearer that is in handover state, application shall call the `Netfp_initiateTargetHandOver` API once all forwarding packets are processed. The API behavior is different whether the channel is requested to operate in Fast Path mode or not.

- *For Fast Path Channels*: Application shall specify the next COUNTC to be used. Per our example, application shall set the COUNTC = 104. The API will ensure that all the packets which had arrived on the S1U link and were buffered on the internal queues are sent to the head of the SA ciphering queue. Packet ordering is guaranteed. In our case, downlink radio bearer security context will be set to use COUNTC value = 104, and packets (n+4), and (n+5) will be diverted to the head of the accelerator queue. In addition, this API reprograms the NETCP to forward all future packets (example, packets (n+6), and (n+7)) directly to the air ciphering engine. The channel will be operating in Fast Path mode, and application can get ciphered packets from the encode Queue.
- *For Slow Path Channels:* The API will reprogram the NETCP to forward all future incoming GTPU packets arriving on the S1U link to the application specified ROHC queue.

  NOTE: In this mode, the application is responsible for processing all the packets which were buffered in the internal queue before processing packets from RoHC queue. Application shall use the `Netfp_getTargetHandOverPackets` API to get the older packets (example, (n+4), and (n+5) first). It then will need to encode those packets using the `Netfp_encodeDrb` API, using the COUNTC values of 104, and 105. After that, application can process packets arriving on the ROHC queue.

7.8.4.2 Uplink Procedures at Target eNodeB

For the data bearers that are handed over with uplink forwarding flows, in addition to steps1-4 described above, applications shall:

At step2:

2.3 For each data bearer that expects uplink forwarding data packets, application shall create "receive only" **UDP** socket(s) to buffer those packets by calling `Netfp_socket/Netfp_connect/Netfp_bind` API sequence.

At Step3:

To ensure correct packet ordering in the uplink, application needs to reorder all packets forwarded from the source eNodeB and packets received from the UE. For the forwarded packets, application shall use the PDCP SN delivered via GTPU extension header. Application shall continue re-ordering process until it gets the all expected uplink packets. Application then shall delete the forwarding sockets calling `Netfp_closeSocket` API.

Please refer to the Basic Handover Test in the NETFP Unit Test as an example which showcases the usage.

7.9 Reestablishment Procedure

The reestablishment handover procedure for the eNB should be implemented as follows:-

- Once a LTE channel for a user has to be reestablished; applications need to invoke the (`Netfp_suspendLTEChannel)` API. This will causes all packets matching the GTPU Identifiers to be placed into an internal queue. This user will now be referred to as the *old user*.
- Applications should now create a *new user* with new user configuration and security keys using the `Netfp_createUser` API
- Applications will now create a new Reestablished LTE channel using the NETFP API (`Netfp_reconfigureLTEChannel`). The API internally creates a new security channel in the NETCP with the new keys.
- Applications can process packets which were residing in the internal suspended queue by invoking the `Netfp_getSuspendedPacket` using the *old user* handle. The packets need to encoded using the `Netfp_encodeDrb` but since the encoding needs to be done using the *new user* handle. **NOTE:** Please be aware that though the channel has been suspended this is still a live stream and depending upon the traffic the function might `Netfp_getSuspendedPacket` never return NULL.
- Once the application has processed all the suspended packets and has decided to make the switch to the newer user; applications will invoke the `Netfp_resumeLTEChannel` API. Once the API is invoked all matching GTPU packets will reach either the ROHC Queue (Slow Path Channels) or the Encode Queue (Fast Path Channels). Packets which

were residing in the internal suspended queue are dropped and the statistics are recorded in the extended socket statistics.

Please refer to the Basic Reestablishment Test in the NETFP Unit Test as an example which showcases the usage.