6WINDGate™
-
Exceptions and Linux - Fast Path Synchronization
-
v2.0

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

1.1 Purpose of the document

This document provides an overview about the exception and the Linux – Fast Path synchronization mechanisms implemented in 6WINDGate 5.

2 6WINDGATE ARCHITECTURE OVERVIEW

2.1 BENEFITS OF 6WINDGate's LINUX – FAST PATH SYNCHRONIZATION

The 6WINDGate architecture is based on a Fast Path implementation that accelerates the Linux Networking Stack. The Fast Path requires dedicated high-performance packet processing software designed to take advantage of modern multicore processor platforms. This Fast Path is isolated from Linux, running on dedicated cores, to ensure deterministic performance.

Having a high-performance isolated Fast Path is mandatory but not enough. It has to be integrated with Linux Control Plane and Management Planes.

There are two options to achieve this integration:

- Redesign how Control and Management Planes interact with the Fast Path. This requires a significant amount of work to adapt and validate a very large number of complex protocols. Standard Linux networking tools have also to be adapted to work with the Fast Path. This approach has been selected by the fd.io/VPP open source project for example.
- Reuse existing Linux Control and Management Planes. This approach requires the design of a Linux-friendly Data Plane to let the Fast Path act as a transparent solution to Linux.

This second option has been successfully implemented in 6WINDGate using Linux – Fast Path synchronization to provide:

- Support for all major Linux distributions.
- Reuse of all existing Linux management tools (iproute, iptables, ipset, brctl, ovs-*ctl, tcpdump…) with no changes.
- Support with no changes of well-known open source Control Plane applications such as FRRouting and StrongSwan.
- Support with no changes of management tools, either open source such as Ganglia, Grafana, Nagios, OpenDayLight and OpenStack or commercial distributions.

As a summary, Linux running 6WINDGate is Linux.

2.2 6WINDGATE EXCEPTION STRATEGY AND CONTINUOUS SYNCHRONIZATION

To achieve the Fast Path transparency to Linux, 6WINDGate implements what we call "Linux – Fast Past synchronization". It relies on two mechanisms: **exception strategy** and **continuous synchronization**, as described in Figure 1.

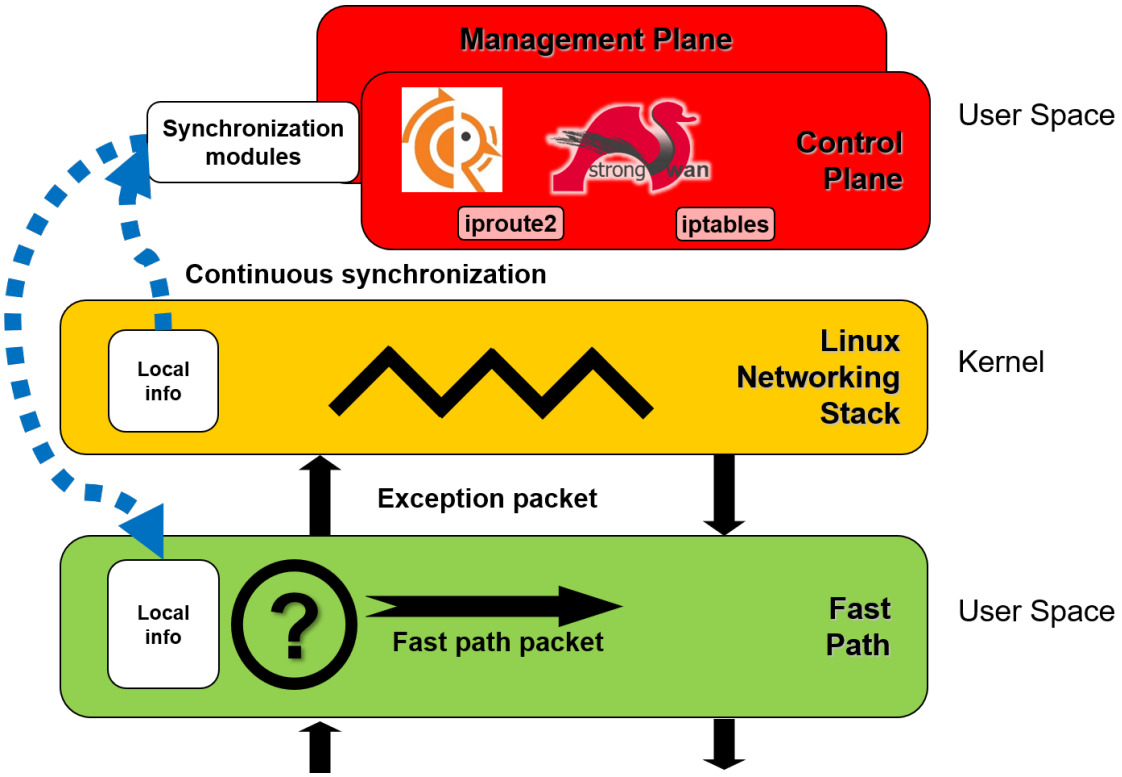

Figure 1: 6WINDGate - Exception and Continuous Synchronization

When local information is missing in the Fast Path to process a packet, when a packet type is not supported by the Fast Path, or when a packet is destined to the local Control Plane, then it is diverted to the Linux Networking Stack. These packets are known as exception packets and this mechanism is called the **exception strategy**.

The Linux Networking Stack is responsible for processing packets that could not be processed at the Fast Path level. These packets will be either processed by the 6WINDGate Linux Networking Stack, or by the Control Plane. It is to be noted that, in most cases, this accounts only for a few percentages of the traffic.

In the case of exception packets due to lack of information, the information learnt in the Linux Networking Stack during the processing of the packet will be transparently synchronized into the Fast Path. This way, subsequent packets of the same flow will then be handled by the Fast Path. This is the mechanism of **continuous synchronization**.

A good example is the case of a packet being diverted to the Linux Networking Stack because L2 forwarding information is missing in the Fast Path. The 6WINDGate Linux Networking Stack will receive the packet, perform L2 resolution and forward the packet. Thanks to the 6WINDGate architecture, the new L2 entry will automatically be configured in the Fast Path, so that a next packet of the same flow is processed in the Fast Path.

2.3 6WINDGate MAIN COMPONENTS

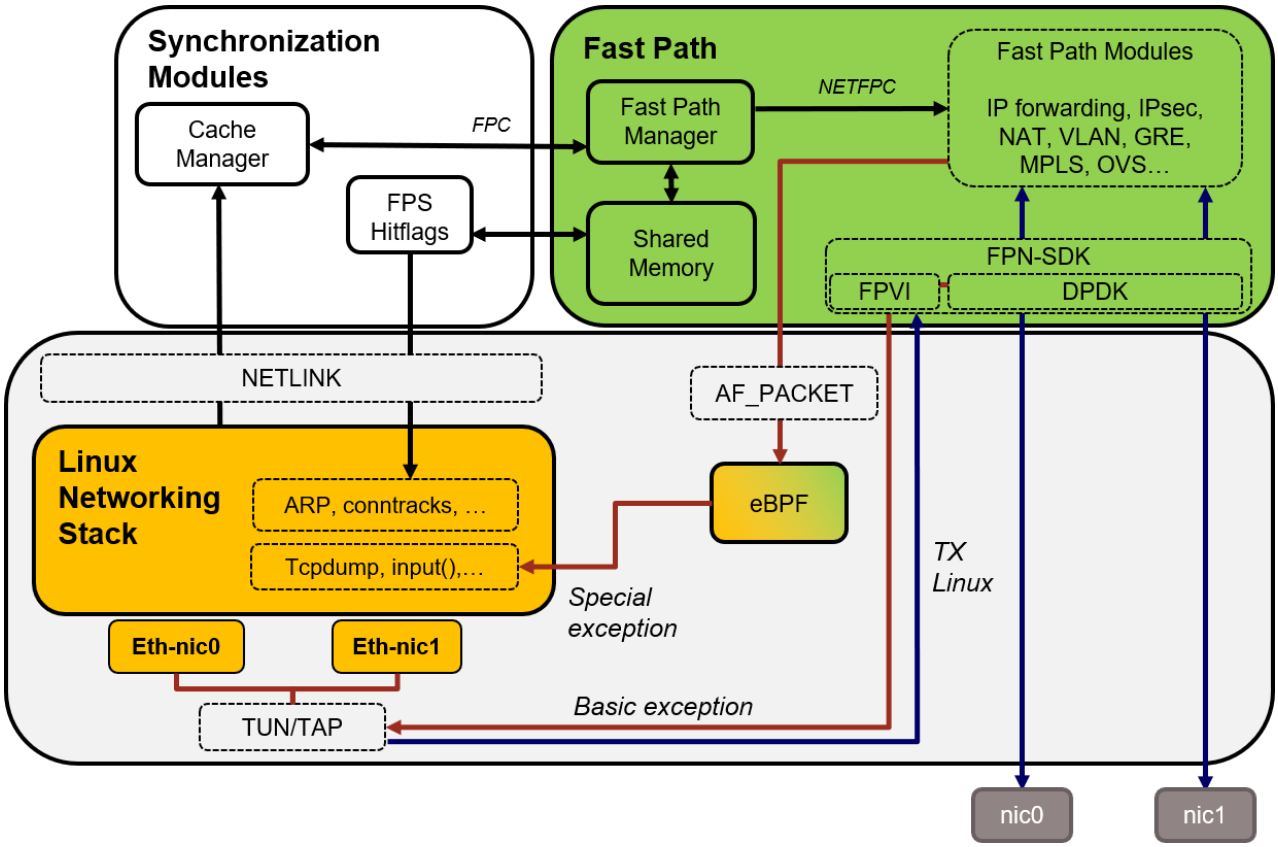

Figure 2: 6WINDGate Main Components

Figure 2 details the main components of the 6WINDGate architecture, as previously introduced. These components will be detailed in the subsequent paragraphs:

- The 6WINDGate Fast Path Networking - SDK (FPN-SDK) provides an abstraction layer to the 6WINDGate Fast Path modules through the FPN API. The FPN-SDK is implementation-dependent; a specific FPN-SDK is required for a given implementation of the Fast Path for a processor environment (DPDK, processor SDK).

- The 6WINDGate Fast Path modules process packets efficiently according to local information stored in the Shared Memory.

- The NETFPC API triggers events in the Fast Path from the 6WINDGate Control Plane.

- The 6WINDGate Cache Manager and Fast Path Manager are Linux userland modules allowing continuous synchronization between the Linux Networking Stack and the Fast Path. Both modules communicate through the Fast Path Control (FPC) API.

- The Linux Netlink API, running without any modification, notifies the Cache Manager of kernel events and state changes for interfaces, Layer 2 - Layer 3 tables, IPsec… It is also used to interface the 6WINDGate Linux Networking Stack to the Control Plane.

- The 6WINDGate Fast Path Virtual Interface (FPVI) allows the communication between the Fast Path and the Linux Networking Stack for the implementation of the exception strategy. Exceptions (refer section 3) are either directly sent to a TUN/TAP Linux driver or provided to an eBPF program

in charge of inserting the exception packet at the right location in the Linux Networking Stack.

- The Fast Path Statistics (FPS) module gathers counters from Fast Path protocols and builds global statistics for the system (Fast Path plus Linux Networking Stack).

- The Hitflags daemon updates hitflags into the Linux Networking Stack when packets go through the Fast Path. Hitflags inform the Linux Networking Stack about updates of ARP entries, conntracks, Linux Bridge…

3 EXCEPTIONS

3.1 EXCEPTION CONCEPT

In the 6WINDGate architecture, all packets are received by the Fast Path, but some of them are delegated to Linux according to the exception concept:

- Local destination,
- Missing processing information (ARP, IPsec SA…),
- Unaccelerated protocol.

The exception concept applies to all protocols that have to be split into two parts:

- The Fast Path only implements packet processing to be done on each packet. This is performed by a simplified IP stack that finds the necessary information in a local memory that has been previously updated by high level protocols (signaling).
- When a received packet is too complex to be processed at the Fast Path level, it is forwarded to the Linux Networking Stack through an exception using a dedicated API called FPVI. For instance, it can be:
  - A packet intended at the Control Plane (ICMP echo request, routing packets, IKE packets…),
  - A packet for which processing information is missing (No L3 route available, No L2 address available for destination/gateway, no IPsec info (SP/SA), missing conntrack info…),
  - A packet for a protocol delegated to Linux such as ARP/NDP or ICMP (TTL expiration).

It can be noted that exception packets are only a few percentage of the traffic making useless to have a full and complex IP stack at the Fast Path level.

3.2 EXCEPTION TYPE

Two kinds of exceptions are defined according to the process to be applied on the packet:

- The first type of exception is called "Basic Exception". For this type of exception, the Fast Path can provide the original incoming packet to the Linux Networking Stack, where it is processed as incoming on a standard network interface.

  For example, a Basic Exception is raised when the route lookup fails during simple IP forwarding.
- The second type of exception is called "Special Exception". This type of exception is raised when the original packet cannot be restored and sent by the Fast Path to the Linux Networking Stack. The exception packet needs to be injected in a specific location in the Linux Networking Stack packet processing path.

  For example, when an IPsec packet is received and decrypted by the Fast Path and forwarding information is missing for the inner packet, the Fast Path needs to raise an exception, but is not able to restore the original packet. Moreover, the decrypted packet shall not be sent in the standard input path of the Linux Networking Stack, as it would be discarded by the Security Policies. In this case, a Basic Exception cannot be used, and we use a Special Exception to inject the inner packet after the IPsec input processing checks in the Linux Networking Stack processing.

3.3 FAST PATH VIRTUAL INTERFACE

The Fast Path Virtual Interface (FPVI) allows exchanging packets between the Fast Path and the Linux Networking Stack. The FPVI makes Fast Path ports appear as netdevices into the Linux Networking Stack.

The purpose of the FPVI is to:

- Provide a physical NIC representor in Linux for configuration, monitoring and traffic capture.
- Send packets from Linux to the Fast Path (locally generated traffic).
- Exchange exception packets between the Fast Path and Linux.

The FPVI is implemented in Linux using the TUN/TAP driver, and in the Fast Path through the FPN-SDK using the DPDK virtio-user PMD providing a virtual port to each TUN/TAP interface.

Packets to be sent locally by the Linux Networking Stack are directly injected in the outgoing flow to be processed by the Fast Path, using the TUN/TAP Linux driver.

The FPVI implements the exception strategy as follows:

- For Basic Exceptions, the FPVI implements a standard processing through the netif_rx function of the TUN/TAP Linux driver.
- For Special Exceptions, on the ingress path, packets are injected at the right place into the Linux Networking Stack thanks to an eBPF program, as explained in the next paragraph. On the egress path, packets are sent directly using the standard sendmsg() API.

3.4 eBPF PROGRAM

3.4.1 Background on eBPF

First, BPF (Berkeley Packet Filter) is an assembly-like language initially developed for BSD systems. The idea is to filter packets early in the kernel to avoid useless copies to userspace applications like "tcpdump".

Then this technology has been extended with new points of attachments, with the ability to call functions, and it has been optimized to generate code close to CPU machine code.

The first BPF use cases targeted kernel tracing, to debug with minimum overhead any event inside the kernel.

Thanks to the networking hooks like TC and XDP, this usage has been extended to network filtering, for example to implement anti-DDoS, or to model hardware component.

In terms of programming, an eBPF program is a C-like program using the uapi/linux/bpf.h API described in the Linux kernel. A LLVM compiler can translate this program into eBPF assembly instructions. This binary can then be loaded into the kernel, verified by the kernel mainly to make sure there is no loop or non-authorized memory access, and finally executed.

3.4.2 Special Exception with eBPF

The processing of Special Exceptions relies on an eBPF program. The role of this program is to drive the packets to the right hook inside the Linux Networking Stack for further processing aligned with the work already done by the Fast Path.

The Fast Path sends a Special Exception via a dummy interface to which the eBPF program is attached with TC. To specify which operation is needed on the packet, meta data is attached to the packet, by means of a specific trailer, called FPTUN. For example, the trailer is filled with the interface index to which the packet should be injected.

Therefore, the eBPF program responsible for Special Exception processing is called the FPTUN handler.

3.4.3 Example with local delivery over GRE

Figure 3 is an example of GRE packets being processed by the Fast Path. The Inner TCP packet should be delivered to the application as coming from the Linux GRE interface.

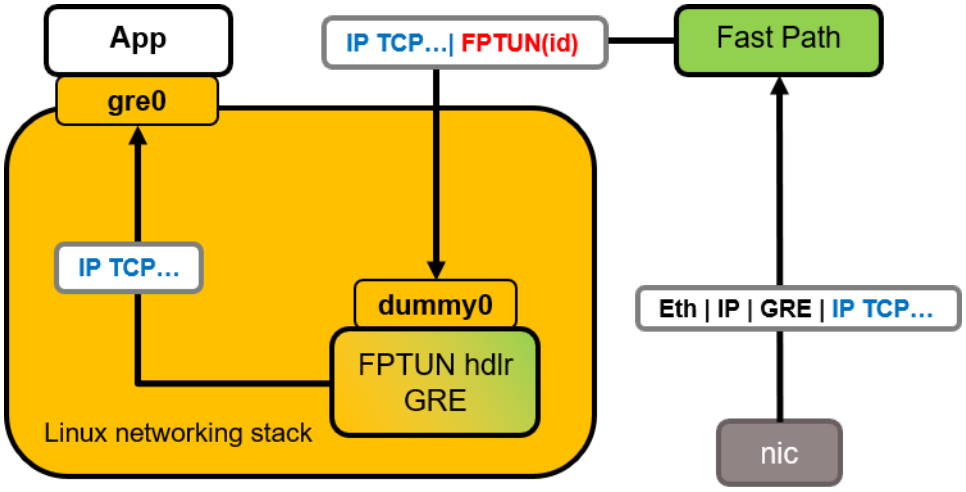

Figure 3: eBPF Example of Local Delivery Over GRE

The Fast Path receives the packet, decapsulates the GRE header and sees that the inner packet is intended for local delivery. It appends a FPTUN trailer with the index of Linux gre0 interface to the packet and sends it to the dummy0 interface. The eBPF FPTUN handler receives the packet, retrieves the FPTUN information using the bpf_skb_load_bytes() API, removes the trailer using the bpf_skb_change_tail() API, and finally the packet is redirected using the bpf_redirect() API to the ingress path of the gre0 interface.

The value of the interface index is known thanks to the Linux – Fast Path synchronization mechanism described in Section 4.

3.4.4 Example with "tcpdump"

As Fast Path packets are not visible to the Linux Networking Stack, a specific mechanism is required to provide the "tcpdump" feature. The exception mechanism and the eBPF technology are used again to provide the "tcpdump" behavior and give a convenient way to capture offloaded packets.

First, when tcpdump is called in Linux, the BPF filter is synchronized in the Fast Path, so that the Fast Path will filter packets according to user patterns.

On match, a copy of the packet will be sent as a Special Exception to Linux. This will result on the packet being displayed by tcpdump on the matching netdevice in Linux.

As the Fast Path processes the original packet, a mechanism is needed to avoid actual processing by the Linux Networking Stack. This is done by marking the tcpdump exception packet through the eBFP FPTUN handler. Another eBPF program is attached to the Linux netdevice to drop the marked packet right after it has been cloned and sent to "tcpdump". This is described in Figure 4.

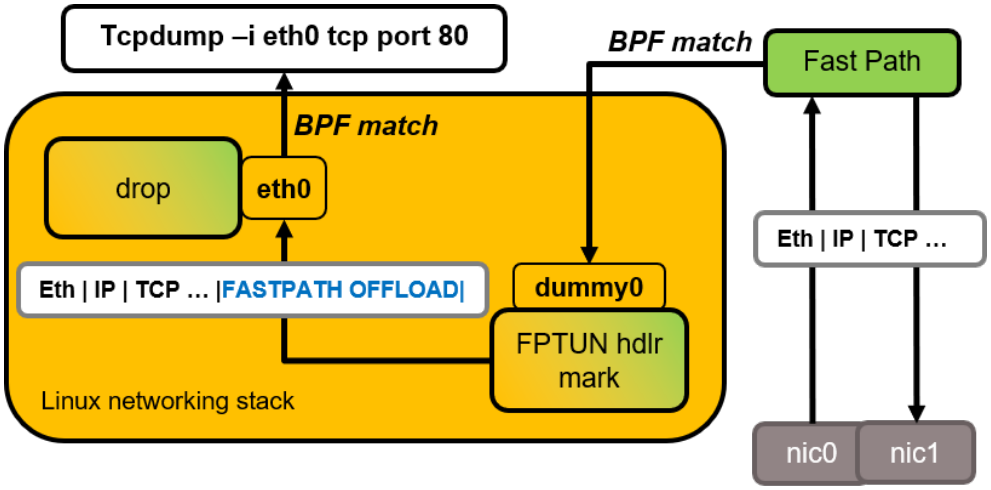

Figure 4: eBPF Example for "tcpdump"

4 LINUX - FAST PATH SYNCHRONIZATION

4.1 OVERVIEW

Figure 5 details the Linux – Fast Path synchronization architecture.

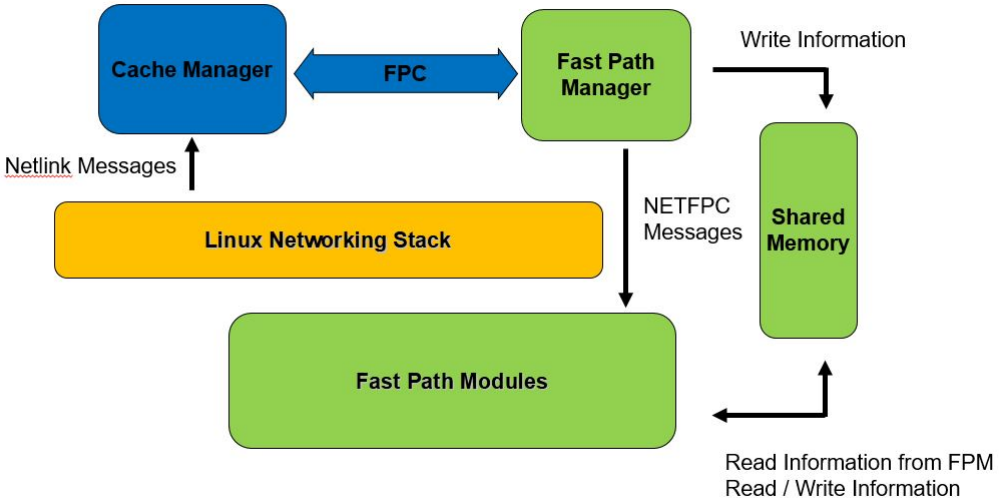

Figure 5: Linux Synchronization Architecture

The Cache Manager is a userland software module that performs synchronization between the Linux Networking Stack and the Fast Path. It listens to the kernel updates (Netlink messages) done by the Control Plane (ARP and NDP entries, L3 routing tables, Security Associations…) and the Management Plane. The Cache Manager synchronizes the Fast Path with this information. Synchronization is made thanks to the FPC API. The Cache Manager sends messages including commands to the Fast Path Manager. Thanks to the Cache Manager, no change is required in the Control Plane and the Management Plane to be integrated with Fast Path modules.

The Fast Path Manager is a userland software module and can be considered as a Fast Path Linux driver. The Fast Path Manager receives command messages from the Cache Manager through the FPC API and analyses these commands to update information for Fast Path modules. The Fast Path sends acknowledgment messages (error management) to the Fast Path Manager using the FPC API.

The update of information by the Fast Path Manager for Fast Path modules can use two different mechanisms:

- The Fast Path Manager writes relevant information for the different Fast Path modules, for instance routing entries, ARP entries, security policies, security associations… in a Shared Memory,
- The Fast Path Manager uses NETFPC. NETFPC is the transport protocol used to communicate between a Fast Path module and its co-localized Fast Path via a network pseudo-interface. This protocol can be used when a notification must be directly sent to a Fast Path module.

4.2 CACHE MANAGER

When the Cache Manager starts, it fetches its state from the Linux Networking Stack: interfaces, route entries... This processing is asynchronously updated when a new physical interface is detected through Netlink or an ioctl.

Then, the Cache Manager sends a reset command to the Fast Path Manager and waits for an acknowledgment of this command before sending any other command.

When the Cache Manager is running, it listens to:

- Netlink events, which are converted and reported to the Fast Path Manager,
- Fast Path Manager responses.

Netkink messages are originated by services (UNIX daemon or kernel modules), they are provisioned into the kernel, and then they are announced over the Netlink socket.

4.3 FAST PATH MANAGER

The Fast Path Manager application is a daemon acting as a server waiting for the Cache Manager to connect.

The initial task for the Fast Path Manager is to get read and write access to the Shared Memory. Then, the Fast Path Manager is waiting for a connection from the Cache Manager to enable FPC communication.

4.4 FPC API

The FPC API is the interface between the Cache Manager and the Fast Path Manager. It defines the exchange protocol and the structures of the configuration messages exchanged between them. The FPC API makes possible to have a distributed system, where the Cache Manager and Fast Path Manager run on different processors.

The FPC API is based on a specific protocol using a connection-oriented UNIX stream. It implements a client (Cache Manager) / server (Fast Path Manager) architecture. Each message is encapsulated with a header that includes a message type, a sequence number, a report, and the length of the message.

4.5 6WINDGate Fast Path Configuration

4.5.1 Shared Memory

The 6WINDGate Fast Path modules read packet processing information from a dedicated memory zone, called the Shared Memory.

The Shared Memory allocation is SDK dependent, but its implementation is generic and the same data structures are provided, whatever the underlying hardware or execution environment. Data structures in the Shared Memory have been specifically designed for multicore processing. To achieve a high level of performance, access to the Shared Memory shall be lock free. This is ensured by a dedicated memory allocation that prevents different software modules to write in same locations and by optimized mechanisms to update data such as routing tables in memory.

The information in the Shared Memory is continuously updated by the Linux Networking Stack - Fast Path synchronization mechanism, and is read by the 6WINDGate Fast Path Modules when they need to process a packet.

Taking routing as an example, the 6WINDGate IPv4 and IPv6 Forwarding Fast Path Modules read entries of the routing table in the Shared Memory. When the routing table has to be updated in the Shared Memory

(addition, deletion of a route…), this is done by the Fast Path Manager that has received a command on the NETFPC from the Cache Manager that previously listened to the Netlink messages between the Control Plane routing application and the Linux Networking Stack.

On the other hand, the Shared Memory is also updated by the 6WINDGate Fast Path Modules to maintain a set of Fast Path Statistics, used by the FPS to provide aggregated statistics when required from the Control Plane.

4.5.2 NETFPC

The Shared Memory is a non-interactive configuration mechanism. The Linux Networking Stack - Fast Path synchronization mechanisms write information there, which is used by the 6WINDGate Fast Path modules when they need it.

However, in some cases, an interactive communication mechanism is needed, that allows one side to trigger an event on the other side.

NETFPC is the transport protocol used to communicate between the Fast Path Manager and the Fast Path via a logical network interface. This is an alternative of writing into the Shared Memory when a change in the configuration requires the Fast Path to act immediately, which typically results in updating internal states outside the Shared Memory.

NETFPC is used for instance for:

- Setting the MTU on an interface as the Fast Path owns the drivers,
- Configuring MAC address or promiscuous mode.

NETFPC uses a point to point communication between the FPM and the Fast Path modules.

4.6 VRF SYNCHRONIZATION

Virtual Routing and Forwarding (VRF) is an IP technology that allows multiple instances of a routing table to work simultaneously within the same router. 6WINDGate provides support for VRF in all the Fast Path modules. In Linux, VRFs are configured using network namespaces.

The Linux / Fast Path Synchronization - VRF module implements synchronization of Linux netns to Fast Path VRFs. It based on Netlink, and the libvrf library is provided to help userland applications manage and monitor 6WINDGate VRFs from any Linux userland process.

5 FAST PATH STATISTICS AND HITFLAGS

5.1 FAST PATH STATISTICS

The Fast Path Statistics module synchronizes the statistics of the Fast Path into the Linux Networking Stack. If this synchronization was not implemented, the system statistics would be inaccurate as the Linux Networking Stack is not aware of the traffic managed by the Fast Path.

These statistics are implemented through the following mechanisms:

- The Fast Path modules update the Shared Memory with statistics,
- The FPS daemon reads the Shared Memory statistics,
- The FPS daemon updates the kernel with the statistics, typically using Netlink.

For instance, an IKE deamon like StrongSwan can rely on up-to-date XFRM statistics, without any patch, even though all the IPsec traffic is being handled by the Fast Path.

Not all kernel statistics can be updated using a userspace API. In particular, at the time of writing there is no API to update the interface statistics or IP MIB. However, a library can be pre-loaded for applications using Netlink, like iproute2, net-snmp, bmon, so that Netlink requests for statistics can be updated transparently with the Fast Path statistics from the Shared Memory.

For other type of management applications, APIs are provided to collect both Linux and Fast Path statistics of interface and IP MIB.

5.2 HITFLAGS

Some kernel objects like ARP entries or conntracks follow a state machine that depends on the usage by the Linux Data Plane. As packets are processed by the Fast Path, Linux is not aware that these entries are used, and a mechanism is needed to prevent them from expiring.

This is the role of the Hitflags mechanism.

Hitflags are implemented through the following mechanisms:

- The Fast Path modules update the Shared Memory with hitflags,
- The Hitflags daemon reads the Shared Memory entries, collects the entries marked with the hitflags and resets the flag,
- The Hitflags deamon updates the kernel state using Netlink.

As a result,the state of kernel objects remains alive as long as the Fast Path is actively using them and the packet processing remains steady in the Fast Path.