6WINDGate Exceptions and Linux - Fast Path Synchronization

6WIND

#SPEEDMATTERS For Serious Networks

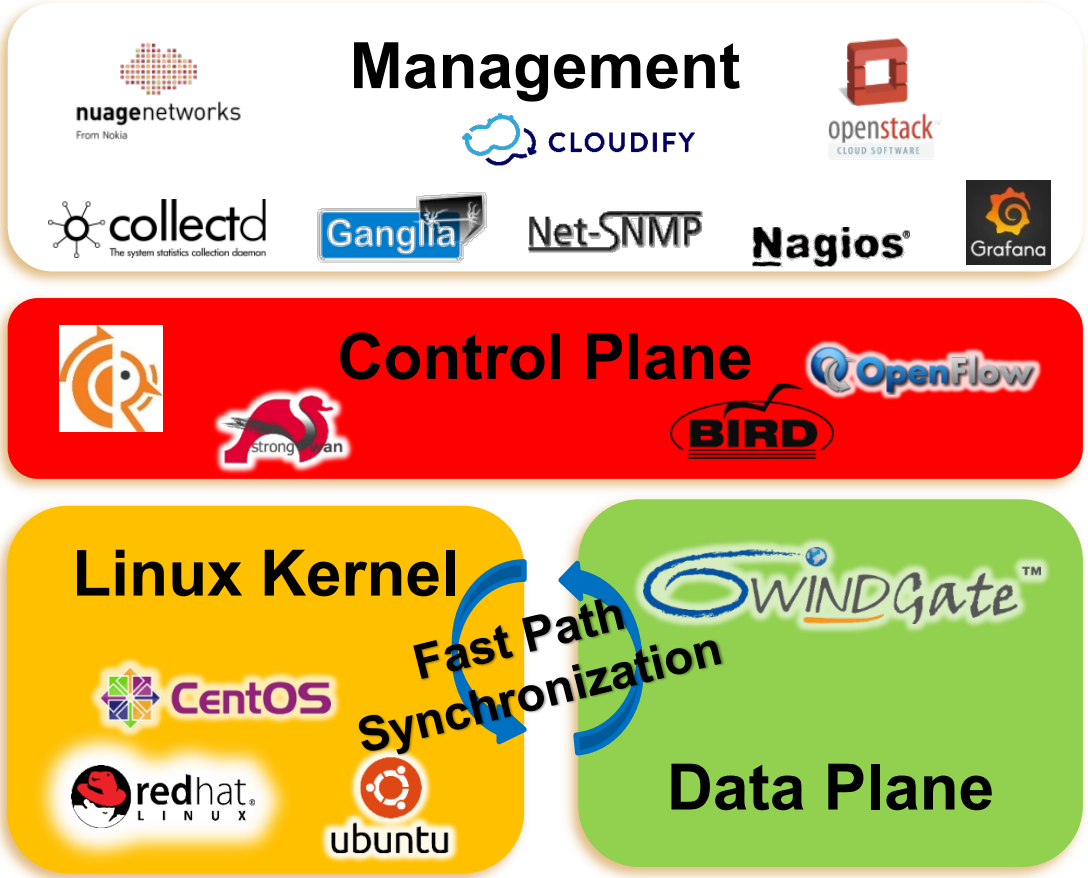
Integration Of Fast Path With Control And Management Planes

- **There are two options to achieve the integration of a high performance isolated Fast Path with Linux Control and Management Planes**
  1. Redesign how Control and Management Planes interact with the Data Plane
     - Requires a significant amount of work to adapt and validate a large number of complex protocols
     - Used by VPP
  2. Rely on the design of a Linux-friendly Data Plane to let the Fast Path act as a transparent solution to Linux
     - No change to existing Linux Control and Management Planes
     - Need for smart collaboration between Linux Networking Stack and Fast Path

- **This second option has been successfully implemented in 6WINDGate using Linux - Fast Path synchronization, taking advantage from the powerful networking capabilities of Linux eBPF**

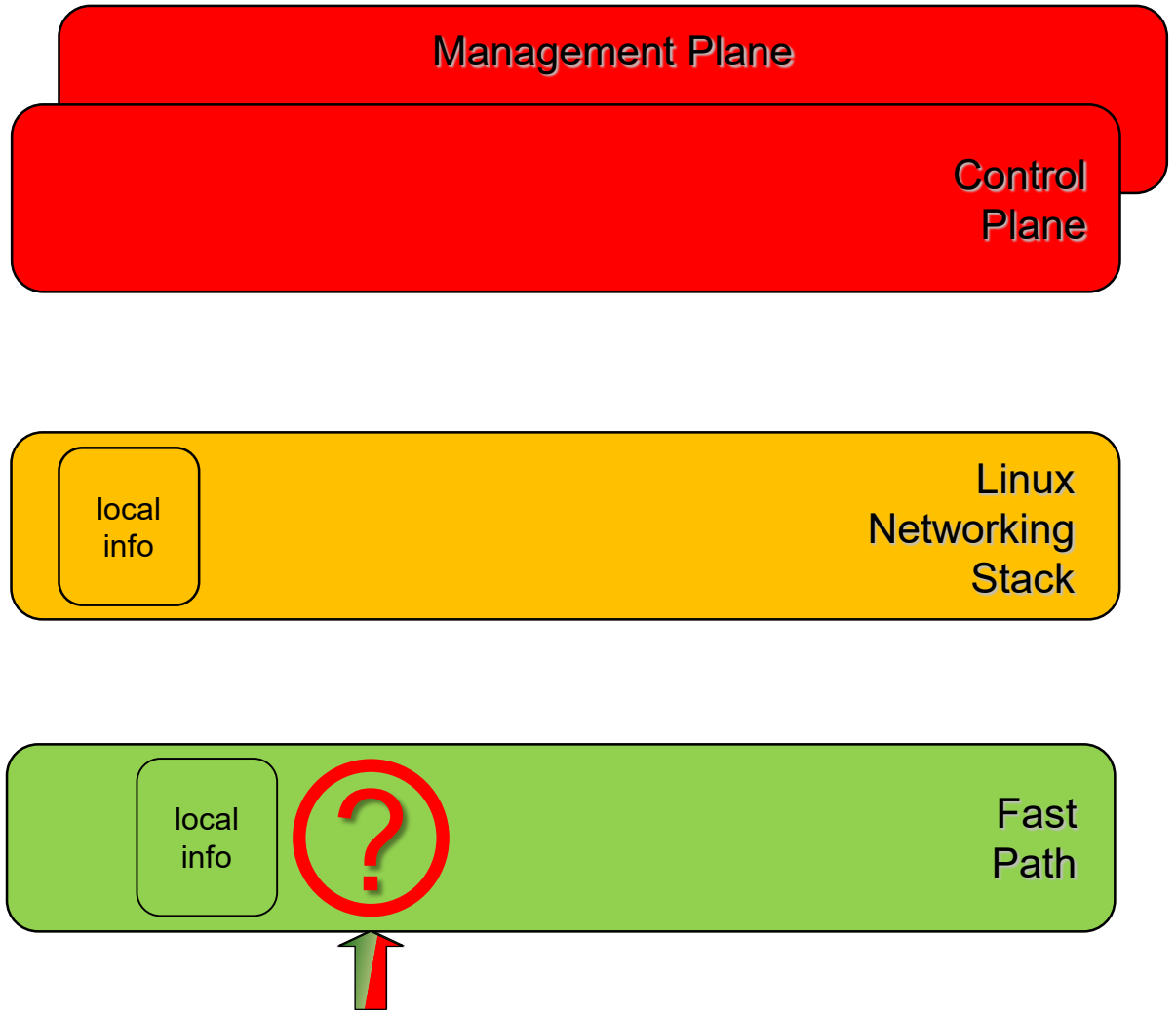
Seamless Integration With Linux

- Existing Linux applications are not modified and developing new applications is pure Linux development
- Compatible with third-party open source or commercial Control Plane applications that configure Linux (routing, IKE, ...)
- Linux management tools can be re-used (iproute, iptables, ipset, brctl, ovs-*ctl, tcpdump, etc.)
- Leverage eBPF to make Fast Path transparent to applications

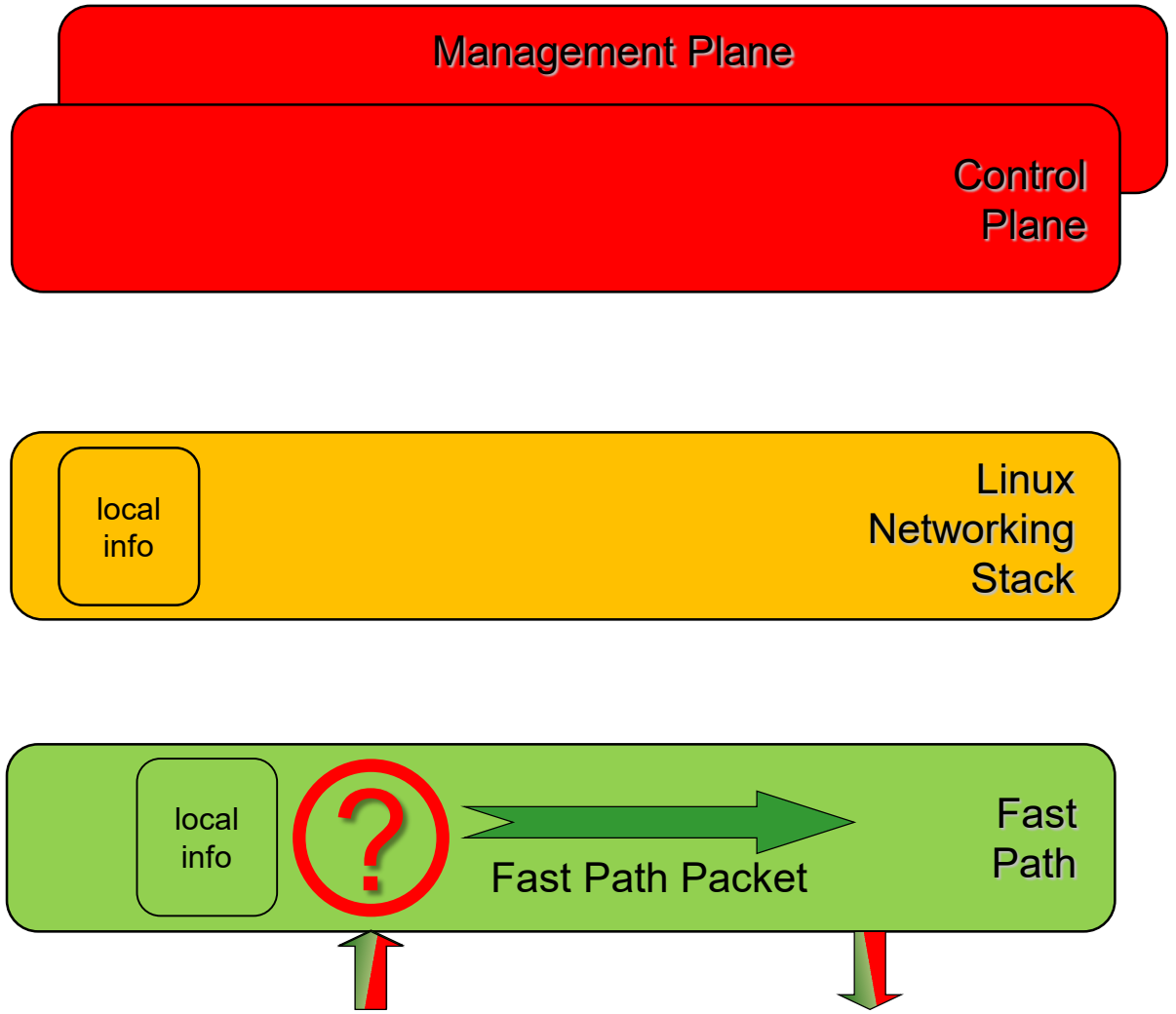


Linux Running 6WINDGate is Linux

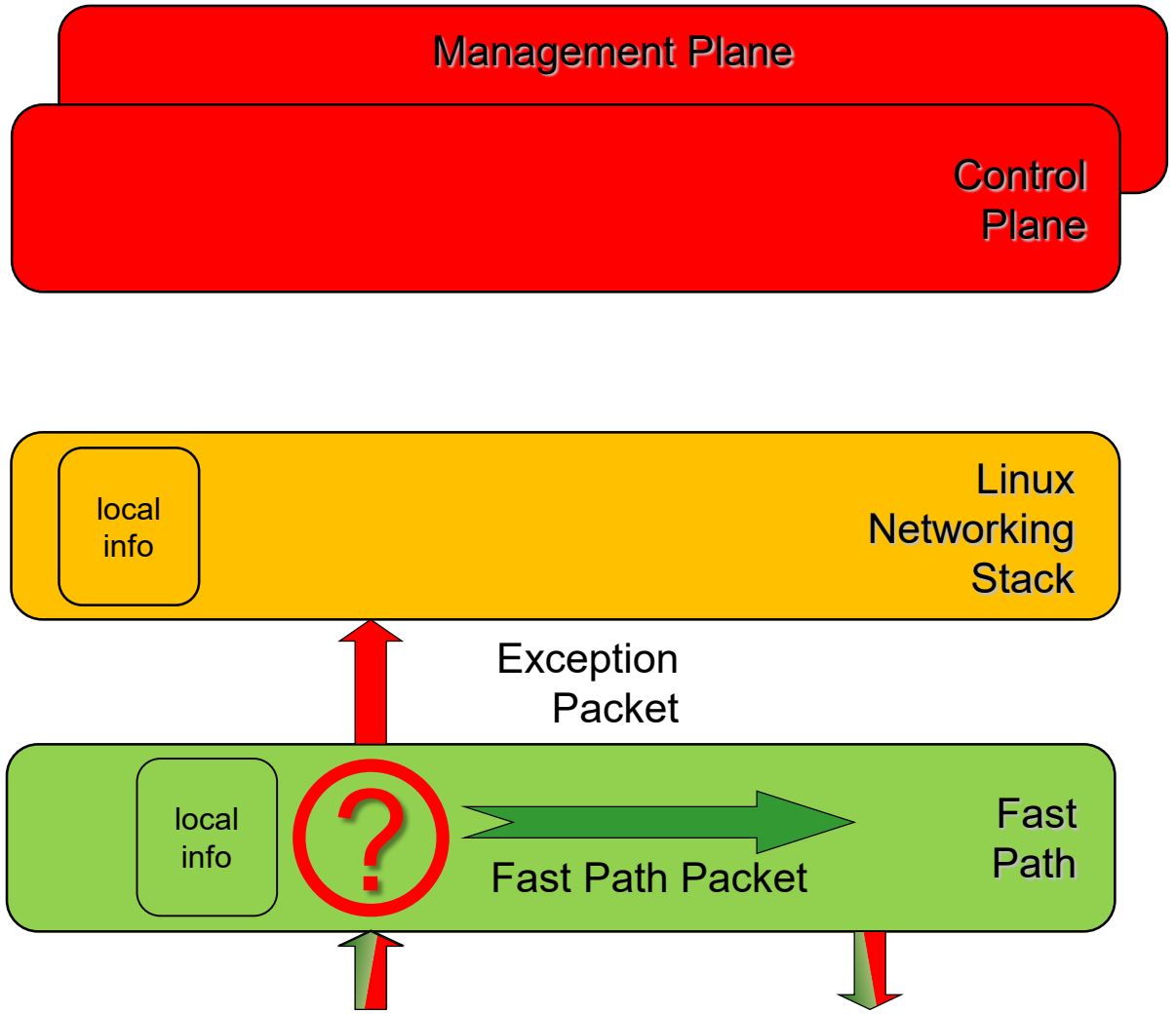
Exceptions And Continuous Synchronization



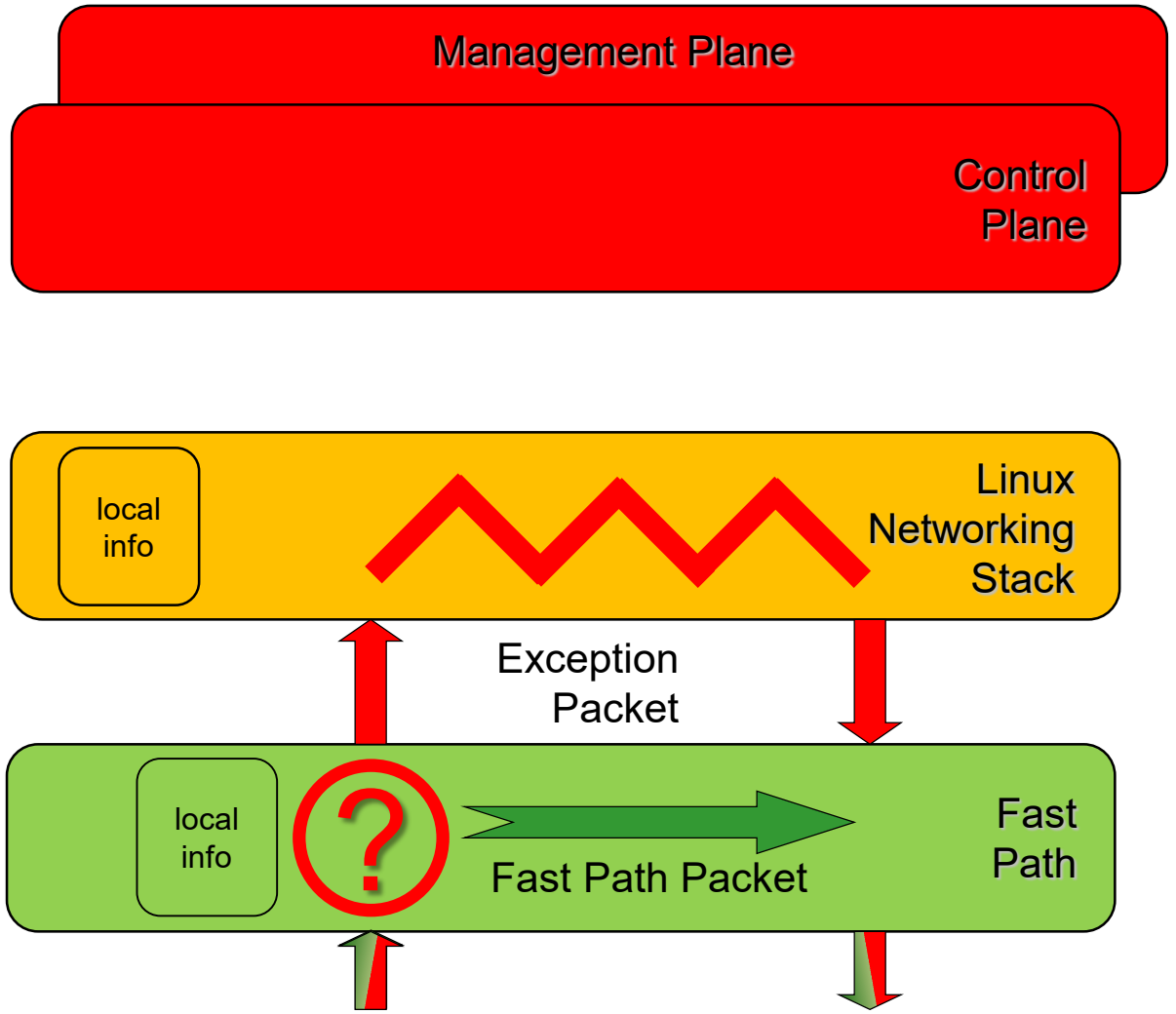
Exceptions And Continuous Synchronization



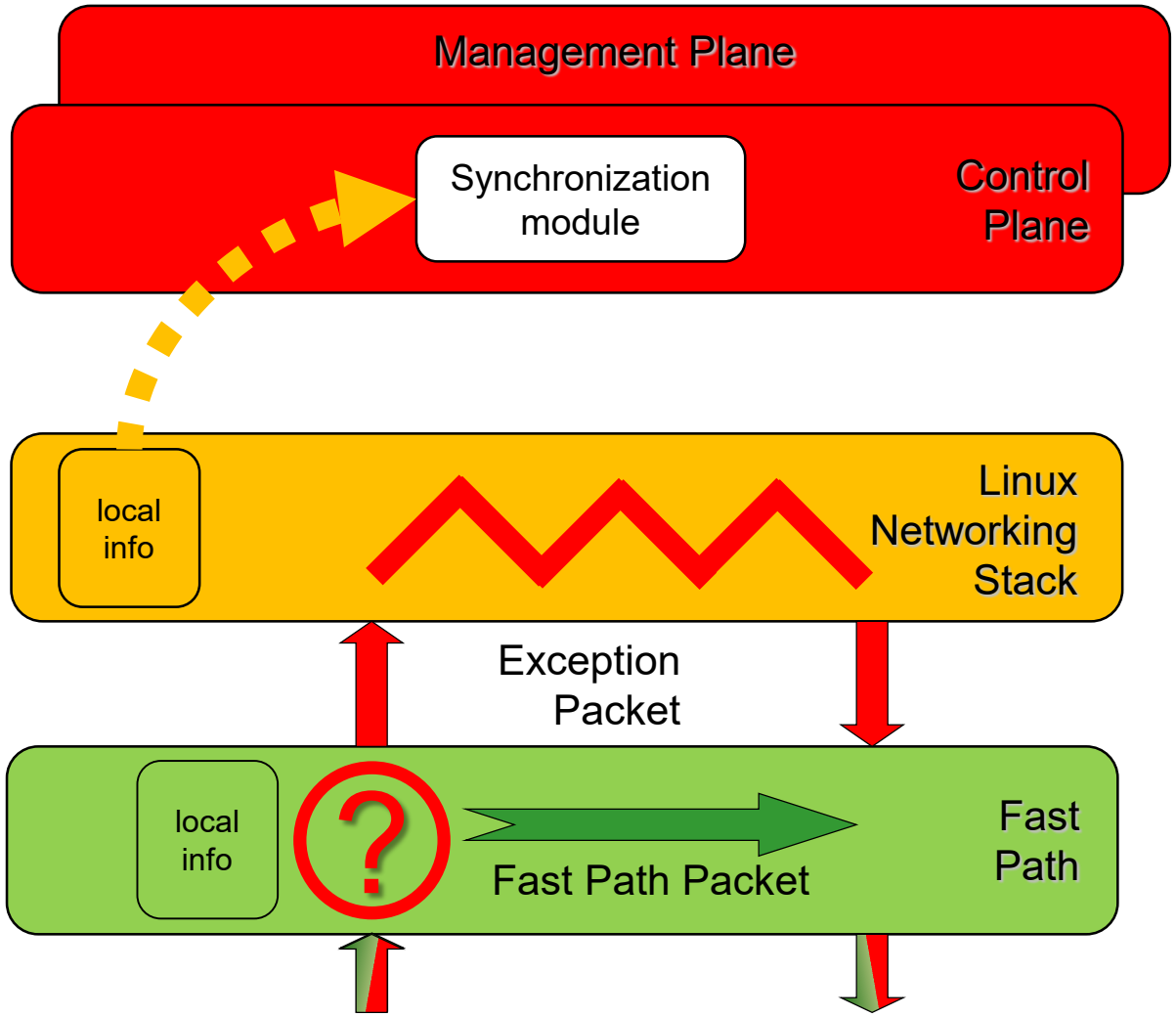
Exceptions And Continuous Synchronization



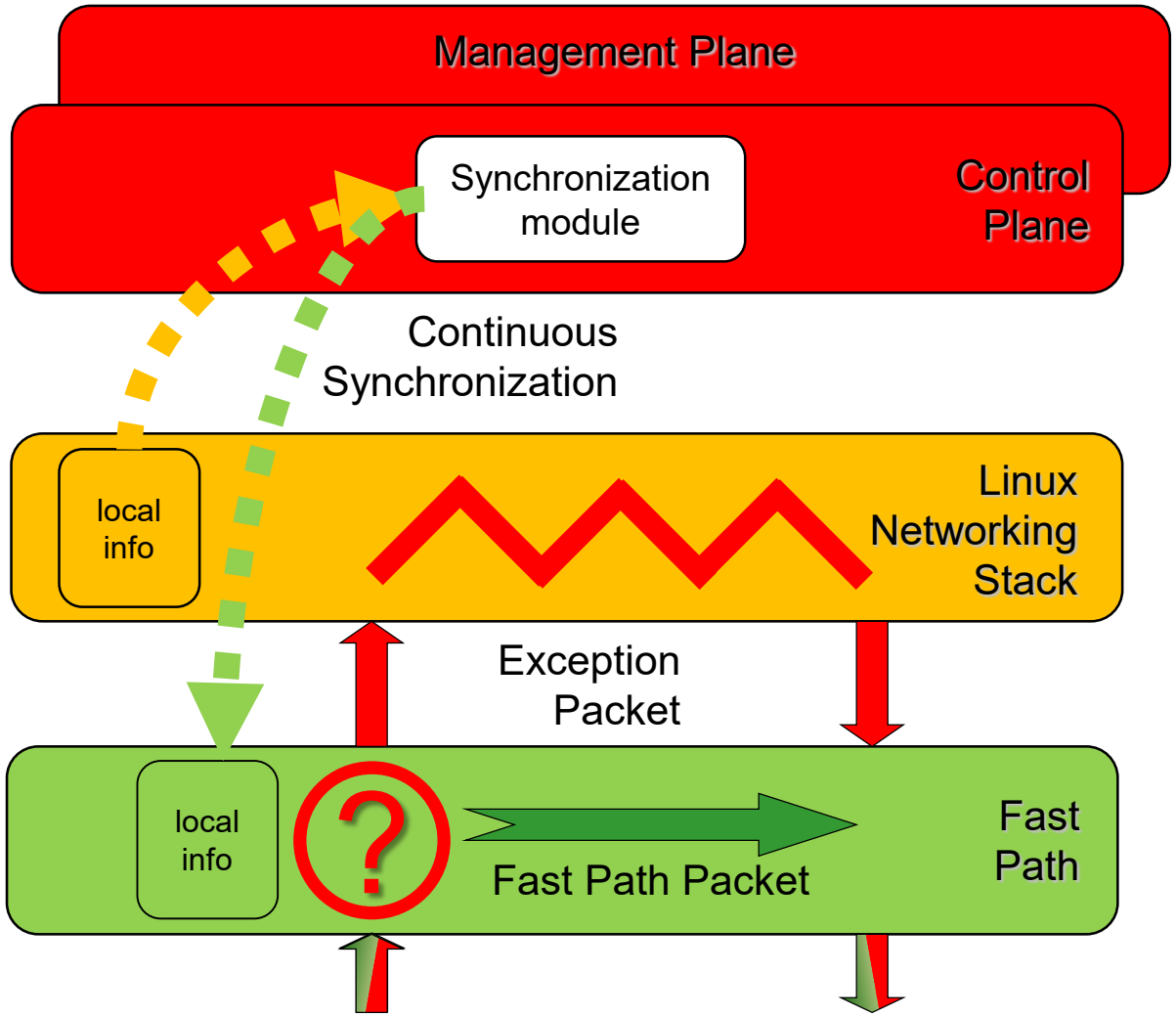
Exceptions And Continuous Synchronization



Exceptions And Continuous Synchronization



Exceptions And Continuous Synchronization

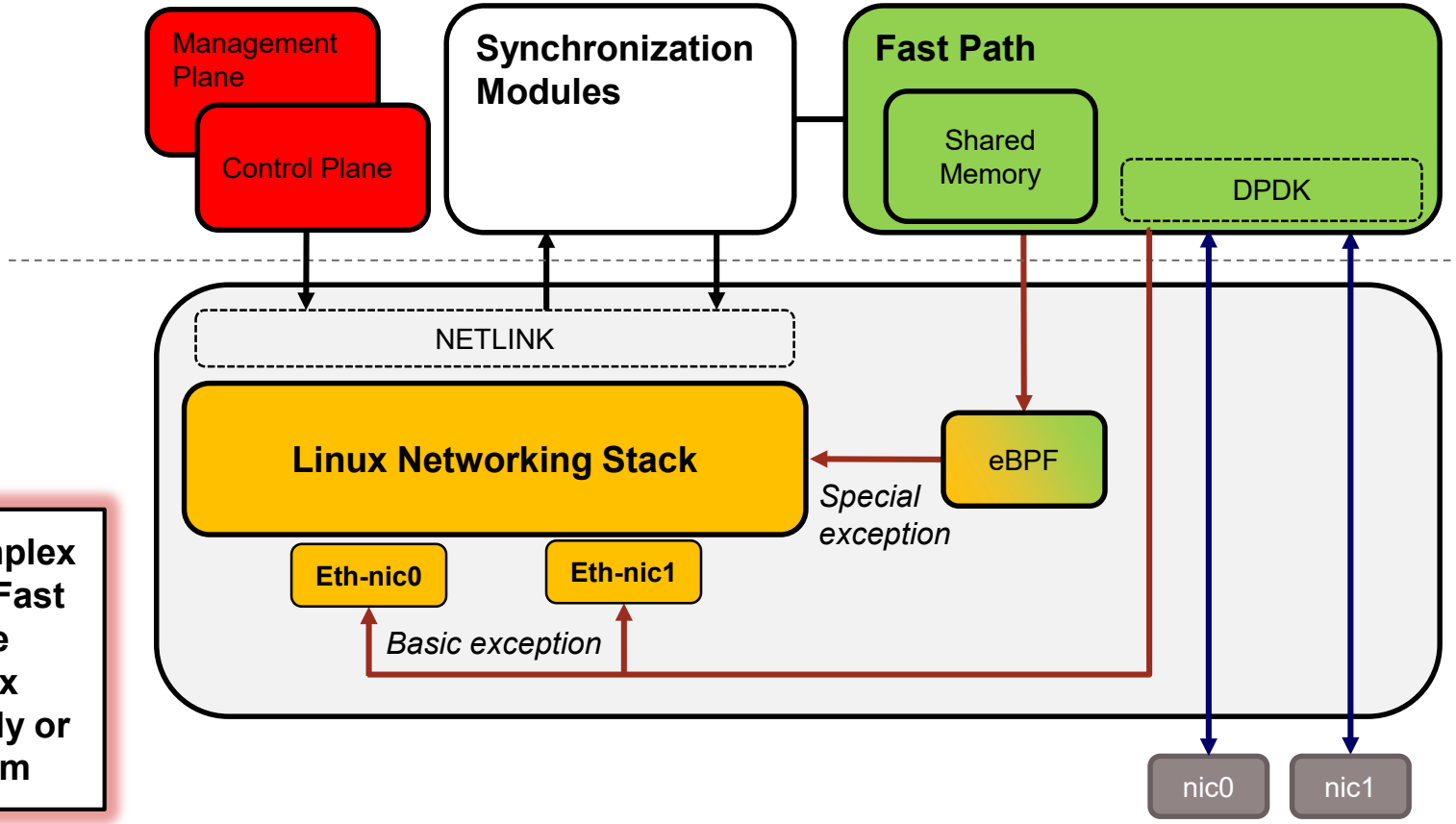


6WINDGate Main Components

As a result, unmodified Linux applications transparently use the accelerated Data Plane as a standard Linux stack

Linux Networking Stack and Fast Path states are synchronized in a shared memory using Netlink

Dedicated optimized userland Data Plane running on top of DPDK



Packets that are too complex to be processed by the Fast Path (exceptions) are reinjected in the Linux Networking Stack directly or using an eBPF program

6WINDGate Detailed Architecture

1 Fast Path

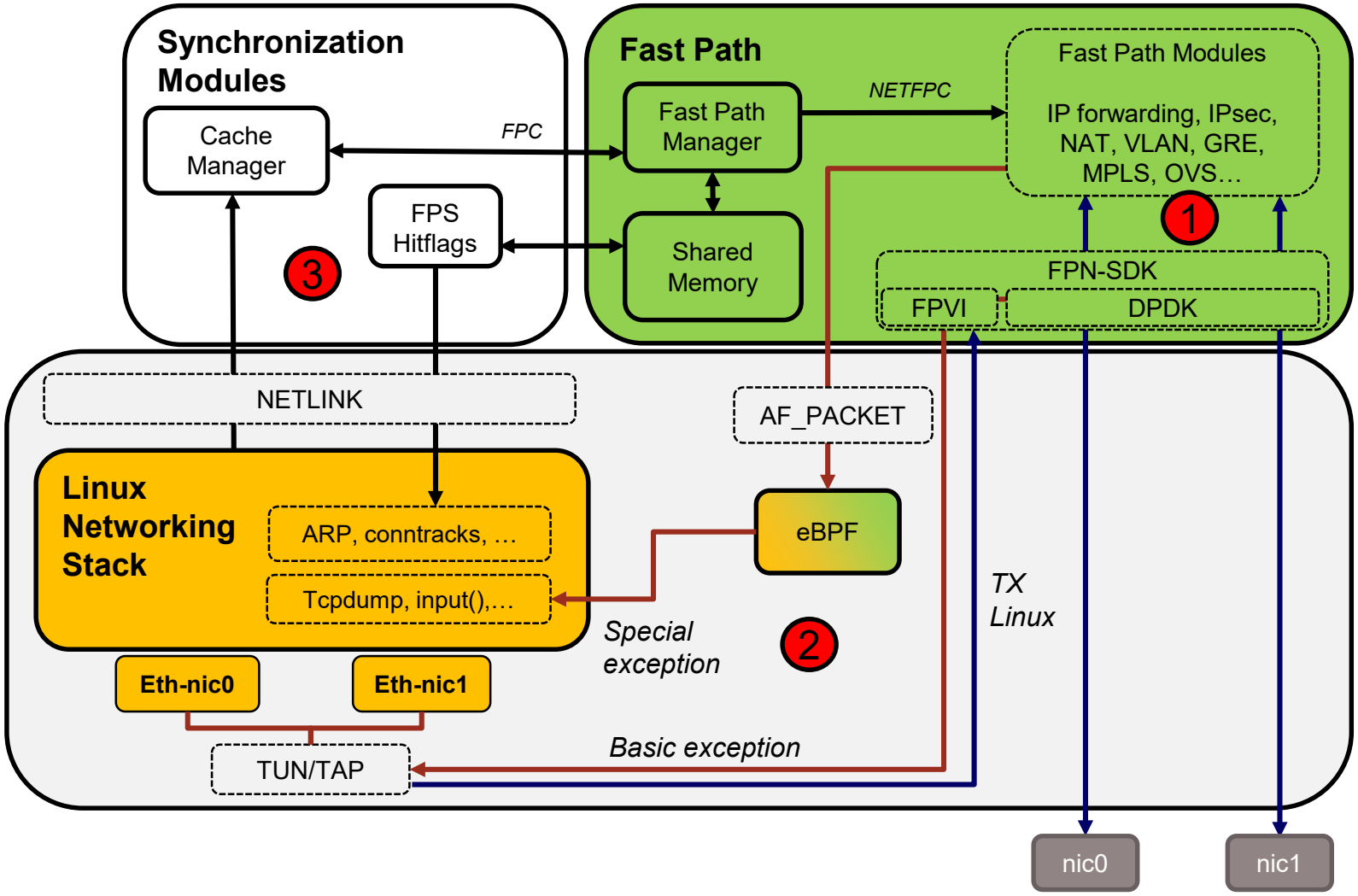
- Fast Path modules on top of DPDK
- Process Linux TX packets
- Read configuration from shared memory and store usage and statistics

2 Exception path

- Basic RX for packets unmodified by Fast Path
- Special RX with eBPF for injecting packets modified by Fast Path in Linux Networking Stack

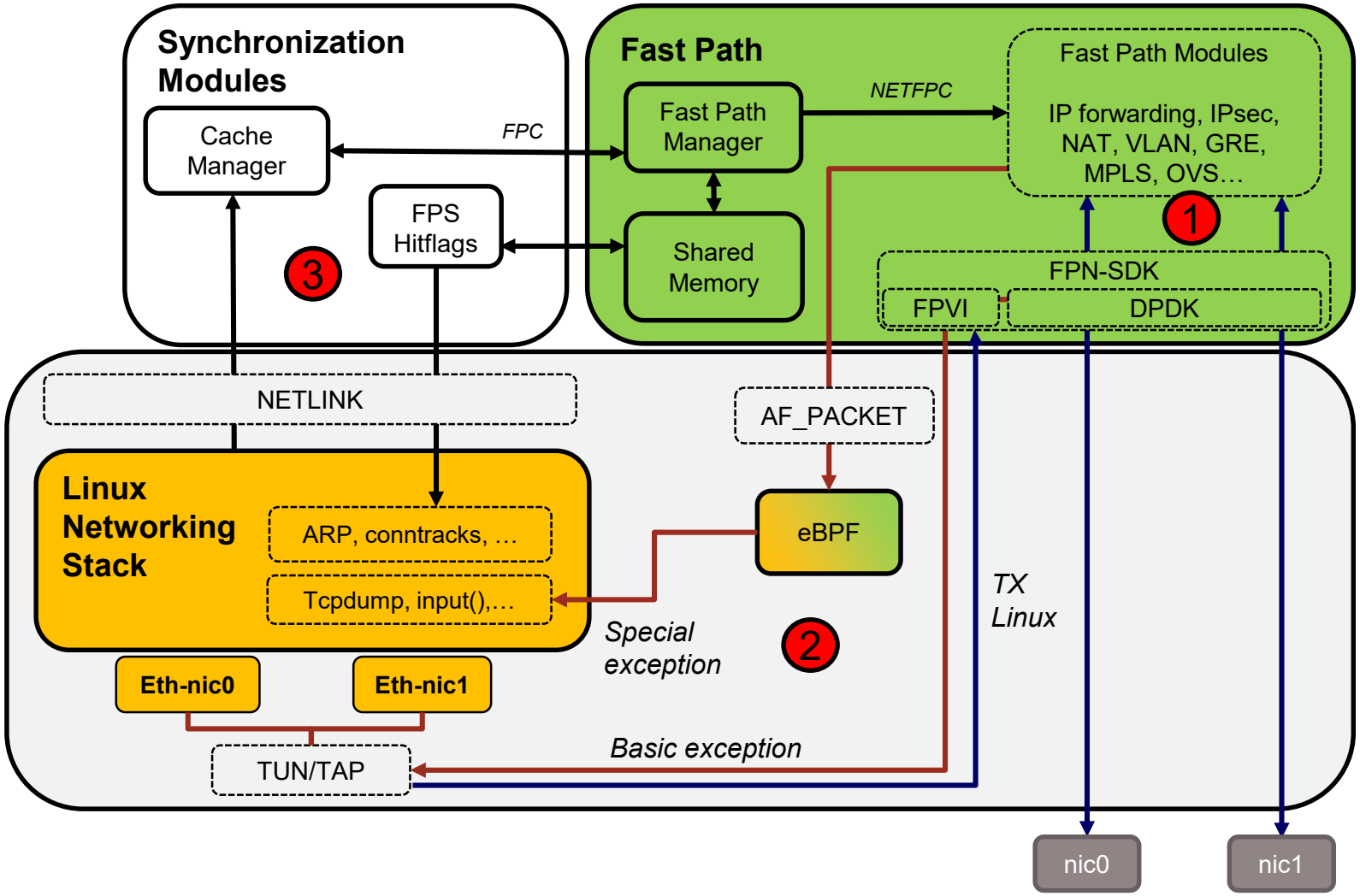
3 Synchronization

- Netlink monitoring to reflect kernel configuration into the shared memory
- FPS / Hitflags to update kernel states from shared memory



6WINDGate Detailed Architecture

1. **Fast Path**
   - Fast Path modules on top of DPDK
   - Process Linux TX packets
   - Read configuration from shared memory and store usage and statistics
2. **Exception path**
   - Basic RX for packets unmodified by Fast Path
   - Special RX with eBPF for injecting packets modified by Fast Path in Linux Networking Stack
3. **Synchronization**
   - Netlink monitoring to reflect kernel configuration into the shared memory
   - FPS / Hitflags to update kernel states from shared memory

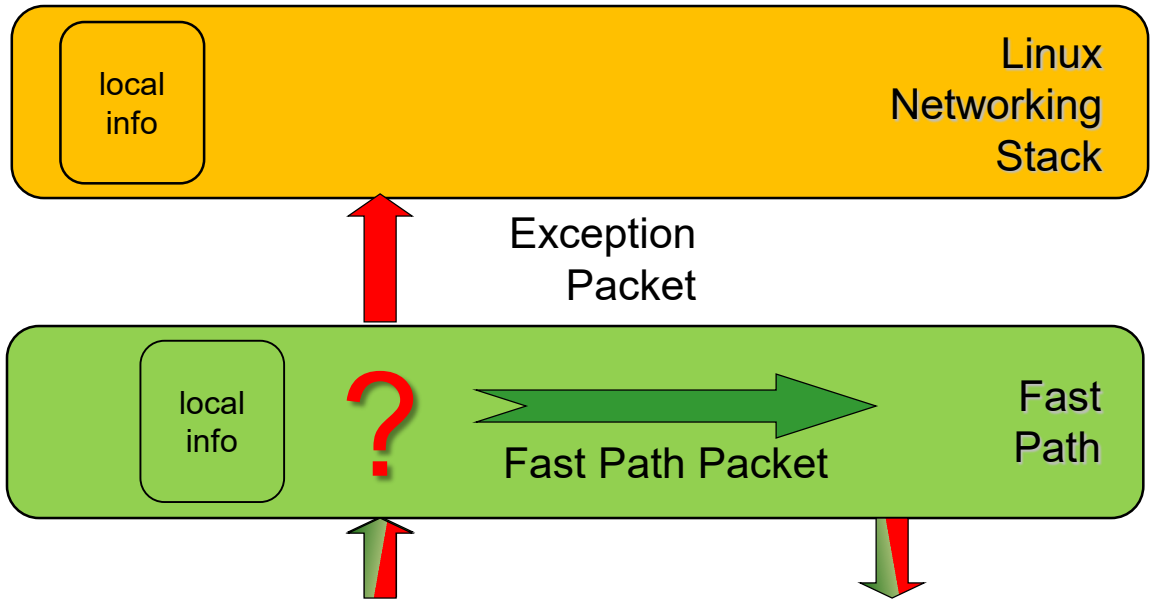


Exception Strategy

- **All packets are received by the Fast Path, but some are delegated to Linux**
  - Local destination
  - Missing processing information in Shared Memory (ARP, IPsec SA, etc.)
  - Unaccelerated protocol
- **Exceptions are sent to Linux**
  - Basic exceptions for standard processing are sent to a TUN/TAP Linux driver
  - Special exceptions for packets that have been preprocessed by the Fast Path are injected at the right place into the Linux Networking Stack thanks to an eBPF program
- **Packets are then processed by the Linux Networking Stack**
  - Missing information (ARP, IPsec SA, etc.) is resolved by Linux and will be synchronized to the Fast Path (see next slides)
- **Benefits**
  - Complete networking stack, relying on Linux for unaccelerated protocols
  - Fast Path benefits from rich Linux Control Plane, no need to develop or change Control Plane daemons
  - No change to Linux

Exception Cases

- **Packets intended at Control Plane**
  - ICMP echo requests
  - Control Plane daemons (BGP, OSPF, IKE, etc.)
  - …
- **Missing info to process packet**
  - No L3 route available
  - No L2 address available for destination/gateway
  - No IPsec info (SP/SA)
  - Missing conntrack info
  - …
- **Protocols delegated to Linux**
  - ARP/NDP
  - ICMP stack (TTL expiration)
  - …



Exception Types

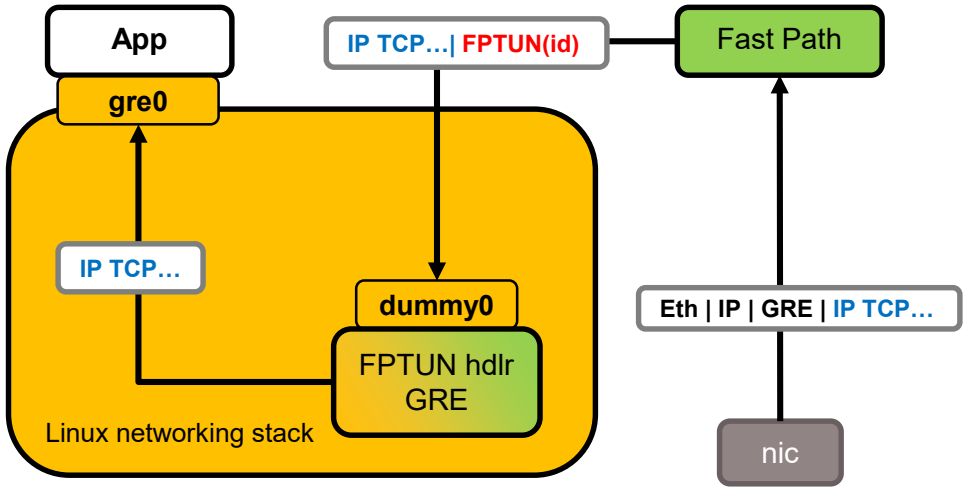
- **Basic exceptions**
  - Default case
  - Original packet sent to the Linux Networking Stack
    - Restore IPv4/IPv6 headers, L2 headers
  - Examples
    - ARP resolution is missing after route lookup during forwarding
    - Local delivery: packet destination is local host e.g. a SSH packet
- **Special exceptions**
  - Original packet cannot be restored: Fast Path already processed some headers
  - Inner packet goes through an eBPF program in Linux to inject it at the right place in the Linux Networking Stack
  - Specific FPTUN trailer is added to the packet to indicate where to inject the packet
  - Examples
    - Local delivery of OSPF packet to the control the plane routing after GRE decapsulation done in Fast path
    - Missing conntrack after VLAN decapsulation don in Fast path
    - Request to give packets to “tcpdump”

What Is eBPF?

- **History**
  - BPF: Berkeley Packet Filter
    - Assembly-like language initially developed for BSD systems
    - Filter packets in the kernel to avoid useless copies to user-space (e.g. tcpdump)
  - eBPF: extension of BPF for Linux with new points of attachment, function calls and performance improvements
- **Usage**
  - Originally used for kernel tracing and event tracing
  - Extended for network filtering (Anti-DDos), hardware modeling, using XDP hooks
- **Programming**
  - Written in C-like
  - Kernel uapi/linux/bpf.h includes API to manipulate the packet
  - Translated into eBPF assembly instructions by LLVM compiler
  - Loaded, verified, JIT compiled, and executed in kernel

Example Of eBPF Usage In 6WINDGate: Inner GRE Local Delivery

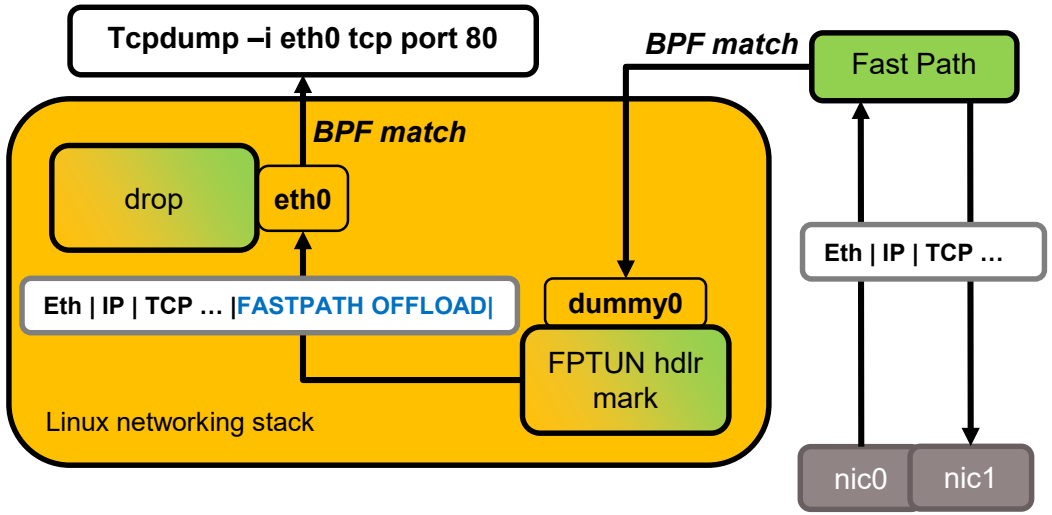
- **Initialization**
  - FPTUN handler eBPF program loaded and attached to dummy interface
- **Fast Path**
  - Ethernet / IP / GRE processing
  - Local delivery: adding FPTUN trailer with “gre0” index and sending it over dummy interface
- **eBPF**
  - Packet goes through TC egress hook
  - After parsing, trailer removal, packet is redirected to gre0 interface




```
__section("tc_fptun_ebpf")
int _tc_fptun_ebpf(struct __sk_buff *skb)
{
        bpf_skb_load_bytes(skb, off, &fptunebpf, sizeof(struct fptunebpf))
        ifindex = _ntohl(fptunebpf.fptunebpf_ifid);
        bpf_skb_change_tail(skb, off, 0)
        return bpf_redirect(ifindex, BPF_F_INGRESS);
}
```

Example Of eBPF Usage In 6WINDGate: tcpdump

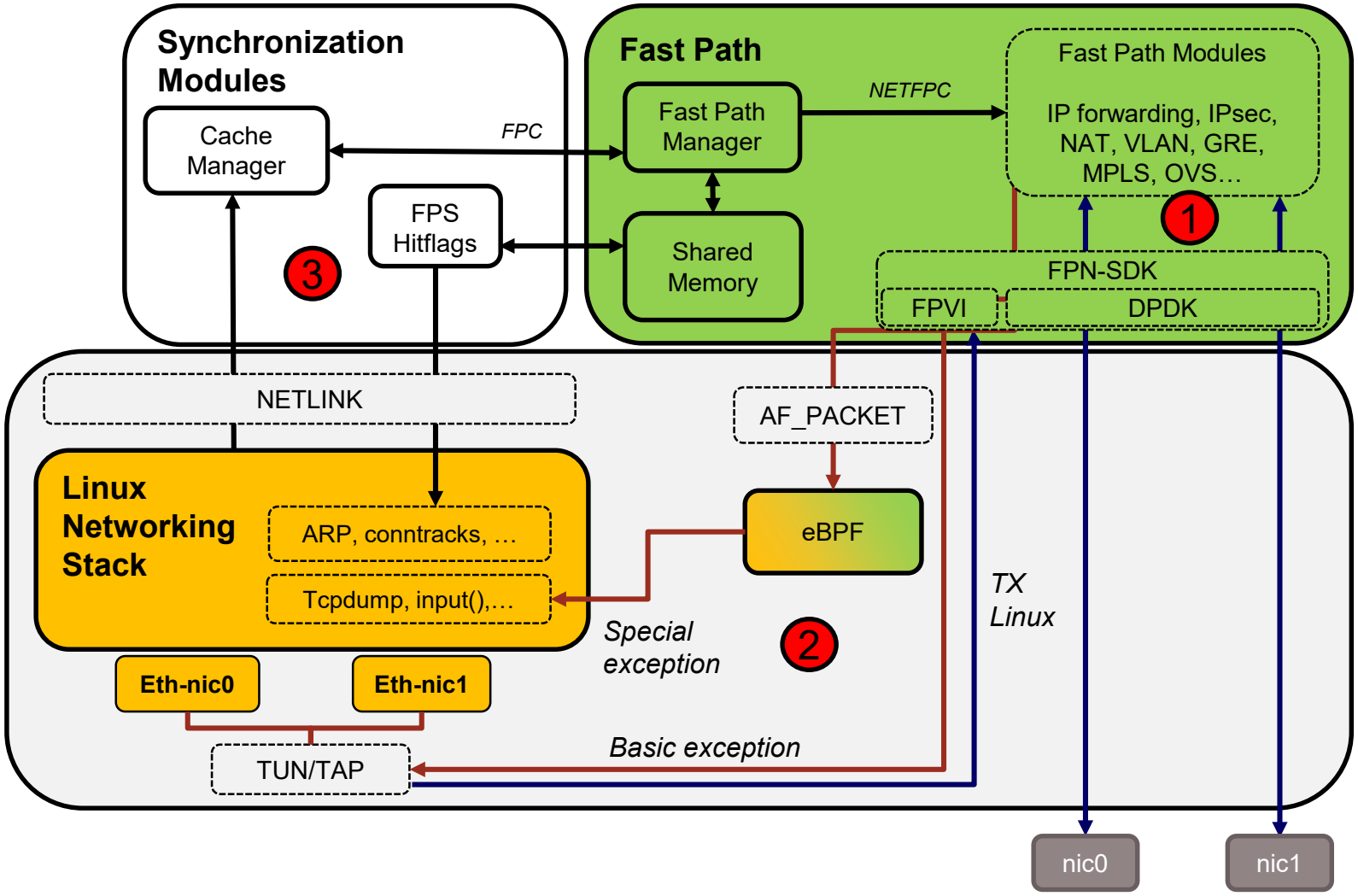
- **On running tcpdump**
  - BPF filter sync'd in Fast Path to filter packets to tap
  - eBPF:
    - FPTUN handler is attached to dummy
    - drop attached to eth0
- **Fast Path**
  - Forwarding
  - On BPF match, send a copy to exception path, but Linux Networking Stack must not process it
- **eBPF**
  - Packet is marked
  - Linux delivers to "tcpdump" socket and to eth0
  - Marked packet is dropped to avoid processing by Linux Networking Stack




```
__section("tc_fptun_ebpf")
int _tc_fptun_ebpf(struct __sk_buff *skb)
{
        mark(skb);
        return bpf_redirect(ifindex, BPF_F_INGRESS);
}
__section("tap_drop")
int _tap_drop(struct __sk_buff *skb)
{
        if is-marked(skb) return TC_ACT_SHOT;
        return TC_ACT_OK;
}
```

6WINDGate Detailed Architecture

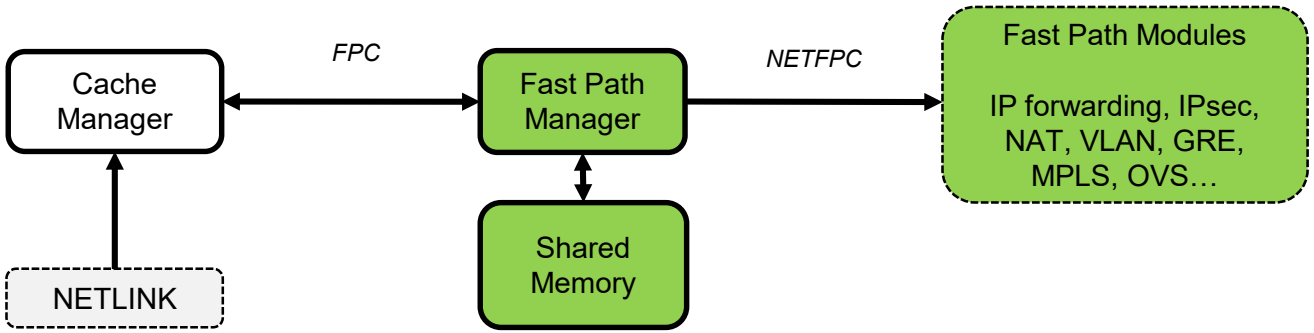
1. **Fast Path**
   - Fast Path modules on top of DPDK
   - Process Linux TX packets
   - Read configuration from shared memory and store usage and statistics
2. **Exception path**
   - Basic RX for packets unmodified by Fast Path
   - Special RX with eBPF for injecting packets modified by Fast Path in Linux Networking Stack
3. **Synchronization**
   - Netlink monitoring to reflect kernel configuration into the shared memory
   - FPS / Hitflags to update kernel states from shared memory



Configuration Synchronization Between Linux and Fast Path

- **Based on two applications**
  - Cache Manager (CM): cmgrd executable
  - Fast Path Manager (FPM): fpmd executable
  - Local or remote communication between CM and FPM is done by the Fast Path Control API (FPC API)

- **Full synchronization path**
  - CM -> FPC -> FPM
  - FPM -> Shared Memory
  - FPM -> NETFPC -> Fast Path



Synchronization: Cache Manager (CM)

- Part of the Linux Fast Path Synchronization module
- Runs as a Linux userland application
- Listens to the Netlink socket, for kernel internal states (Control Plane and configuration updates)
- Transforms Netlink messages into FPC messages
- Control Plane modules (routing, IKE, PPP…) are not modified

Synchronization: FPC API

- **FPC API**
  - Interface between Cache Manager and Fast Path Manager
  - Defines the exchange protocol and the structures of the configuration messages exchanged between the Cache Manager and the Fast Path Manager

- **Dedicated protocol**
  - UNIX or TCP socket
  - client/server
  - Common header
  - Type, sequence number (SN), report, length

Synchronization: Fast Path Manager (FPM)

- **Part of the Linux Fast Path synchronization module**
- **Runs as a Linux userland application**
- **Translates FPC API messages to configure Fast Path modules using**
  - Read / write Shared Memory
  - Send / receive notifications to / from Fast Path through NETFPC

Synchronization: Shared Memory

- **Contains structures for**
  - Physical ports
  - Forwarding table
  - Statistics
  - IPsec databases
  - etc.

- **Read/write access for**
  - FPM: writes local information received from CM through FPC messages
  - Fast Path: reads local information used for packet processing (L2/L3 entries, IPsec SAs, etc.) and writes statistics
  - FPS: reads statistics

- **Allocation is specific to processor architecture, contents are generic**
  - POSIX shmem implementation

Synchronization: NETFPC

- **RPC-like API to trigger an event from Linux to Fast Path**
  - Useful to get a function being called in Fast Path execution environment, typically to change NIC settings via the Fast Path drivers
  - Examples
    - Set the MTU on an interface (the Fast Path owns the drivers)
    - Configure MAC filtering
    - Enable promiscuous mode
- **Implemented as a communication socket between FPM and Fast Path**
  - Point to point communication with socket API: open(), recv(), send(), close()
  - Default is UNIX socket transport, or IPv6 RAW for non-userspace Fast Path

Synchronization: VRF

- **Virtual Routing and Forwarding (VRF): IP technology that allows multiple instances of a routing table to work simultaneously within the same router**
- **6WINDGate provides support for Virtual Routing and Forwarding (VRF) in all Fast Path modules**
- **In Linux, VRFs are configured using network namespaces (netns)**
- **The Linux / Fast Path Synchronization - VRF module implements synchronization of Linux netns to Fast Path VRFs**
  - Userland API: libvrf
    - This library allows to manage and monitor 6WINDGate VRFs from any Linux userland process
  - Cache Manager makes use of libvrf to synchronize netns-VRF in Fast Path VRF

Synchronization: Fast Path Statistics (FPS)

- **Reports Fast Path statistics into the Linux Networking Stack**
  - Fast Path modules update the Shared Memory with statistics
  - FPS daemon reads Shared Memory statistics periodically
  - Statistics are updated in Linux Networking Stack via NETLINK e.g. XFRM family for IPsec
- **Unfortunately the coverage is limited by the userspace API**
  - Currently no way to update per interface statistics, or IP MIB
- **Helper to fetch statistics**
  - Well-known tools like iproute2, SNMP, bmon use NETLINK to get statistics
  - FPS provides a library transparently catching NETLINK requests for statistics (IFLA_STATS attribute) and updating the answer with Fast Path statistics

Synchronization: Hitflags

- **When packets go through the Fast Path, the kernel object states are not updated**
  - Fast path relies on Linux slow path for ARP, conntracks
  - When not used, these entries will expire and get removed by Linux
  - The synchronization will raise periodic waves of packet exceptions
- **The Fast Path Hitflags daemon is in charge of updating the kernel states**
  - Fast Path module writes the hitflag field into the Shared Memory when the entry is used
  - Hitflags daemon scans the Shared Memory entries (ARP for example) periodically
  - Entries marked by Fast Path are updated in Linux Networking Stack via NETLINK
- **Packets flow now keeps steady in Fast Path**
  - In-use entries in Fast Path remain alive in Linux