

# introduction

Monday, December 1, 2025 10:26 AM

Below is a **much more detailed, expanded, deeply explanatory** version of the notes.

This is now at a **textbook level**, with very clear structure, internal working details, and Linux-specific insights.

## EXTENSIVE + DETAILED NOTES ON THE LINUX NETWORK STACK & OSI MODEL

### 1. Introduction to Networking Layers

Networking is broken into layers so that each part of the process is easier to understand and modular.

The OSI (Open Systems Interconnection) model provides a **seven-layer conceptual framework** used to describe how data travels from one computer to another.

Linux, however, **does not strictly follow the OSI model** internally.

Instead, Linux implements only the layers necessary for packet transmission and reception.

### 2. The OSI Model – Detailed Explanation of Each Layer

#### Layer 1 – Physical Layer

- Lowest layer of OSI.
- Deals with **raw bit transmission** over a physical medium.
- Handles:
  - Electrical signals
  - Voltage levels
  - Light pulses (optical fiber)
  - Radio frequencies (WiFi)
  - Connectors, cables, NIC PHY chips

**Linux Kernel involvement:**

 **None.**

Physical layer is implemented by **hardware** and device electronics.

#### Layer 2 – Data Link Layer

This layer is responsible for **reliable transfer of frames** between two nodes connected directly.

##### Key Responsibilities

- Framing (creating Ethernet frames)
- MAC addressing
- Error detection using Frame Check Sequence (FCS)
- Handling collisions (CSMA/CD in old Ethernet)
- Local transmission (within a LAN)

##### Linux role

Linux implements layer 2 through:

## 1. Network Device Drivers

- Example: e1000, ixgbe, r8169, tg3
- These drivers talk directly to the NIC hardware.

## 2. Soft Network Stack (Generic Layer 2 handling)

- Processes incoming Ethernet frames
- Prepares outgoing frames

### Important Linux Concepts at L2

- **sk\_buff (socket buffer)**  
Representation of a frame inside Linux networking stack.
- **net\_device structure**  
Represents a NIC (eth0, wlan0).

## Layer 3 – Network Layer

The main purpose: **packet forwarding and addressing**.

### Responsibilities

- IP addressing
- Routing (deciding packet path)
- Fragmentation & reassembly
- TTL management
- Traffic control rules
- Firewall (Netfilter)
- NAT (Network Address Translation)
- IPsec (encryption/authentication)

### Linux role

Linux implements:

- **IPv4**
- **IPv6**

Other protocols exist but are rare, like DECnet, IPX, X.25, etc.

### Important Functions in Linux IP Layer

- ip\_rcv() – Entry point for IPv4 incoming packets
- ip\_forward() – For forwarding packets
- ip\_output() – For sending packets to NIC
- ip\_fragment() – For IP fragmentation

## Layer 4 – Transport Layer

Manages **end-to-end communication** between processes on two different hosts.

### Provides:

- Reliability (TCP)
- Connection management (SYN, FIN)
- Error checking
- Port numbers (identification of applications)
- Multiplexing using ports

### Common protocols

- **TCP (Transmission Control Protocol)**
- Reliable

- Connection-oriented
- Ensures ordered delivery
- **UDP (User Datagram Protocol)**
- Fast but unreliable
- Used for DNS, VoIP, DHCP

#### **Linux Role**

Linux fully implements transport protocols inside the kernel:

- TCP implementation (state machines, congestion control)
- UDP handling
- SCTP, DCCP (less common)

#### **Major Linux Functions**

- `tcp_v4_rcv()` – Incoming TCP packets
- `tcp_sendmsg()` – Sending data
- `udp_sendmsg()` – Sending UDP packets

## **Layer 5, 6, 7 – Session, Presentation, Application Layers**

These layers are **not implemented in the Linux kernel**.

#### **Layer 5 – Session**

- Handles sessions between applications.
- Example:
- RPC
- NetBIOS session layer

#### **Layer 6 – Presentation**

- Data formatting
- Encryption (except IPsec which is L3)
- Compression
- Character encoding

Examples:

- TLS/SSL (OpenSSL)
- JPEG/PNG formats
- JSON/XML parsing

#### **Layer 7 – Application**

- Actual user-level services.
- Examples:
- HTTP
- SMTP
- DNS
- FTP
- SSH

#### **Linux Kernel involvement for L5-L7:**

⌚ **None.**

These are entirely in **userspace applications**.

## **3. Mapping OSI Model to Linux Kernel Implementation**

Linux primarily implements:

- Layer 2 → NIC drivers, Ethernet handling

- Layer 3 → IP handling (IPv4, IPv6)
- Layer 4 → TCP, UDP, SCTP

Everything above is outside the kernel, and the physical layer is purely hardware.

## 4. The Essence of the Linux Network Stack

The Linux kernel networking stack can be summarized by **packet flow through three main layers**:

### Incoming Packet Path

- L1 (hardware)
- L2 (NIC driver)
- L3 (IP layer)
- L4 (TCP/UDP)
- Userspace Application

**Explanation:**

- NIC receives a frame and triggers interrupt or NAPI poll.
- Driver hands frame to kernel using netif\_receive\_skb().
- Kernel validates Ethernet type and sends to IP layer.
- IP layer decides:
  - Is it for local machine?
  - Or should it be forwarded to another interface?
- If local, transport layer (TCP/UDP) processes it.
- Finally passed to application via sockets.

### Outgoing Packet Path

- Application (sendmsg/write)
- L4 (TCP/UDP)
- L3 (IP layer)
- L2 (NIC driver)
- L1 (hardware)

**Explanation:**

- Application writes data to socket.
- Transport layer builds segments (TCP) or datagrams (UDP).
- IP layer wraps packet with IP header.
- NIC prepares Ethernet frame and sends it.

### Forwarding Path (Router Behavior)

If the packet is NOT for local machine:

L2 → L3 (Forwarding decision) → L2 (Other NIC)

Linux can behave like a router when ip\_forward=1.

## 5. What Happens to a Packet Along the Path?

### ✓ 1. Packet Modification

- Due to NAT (iptables)
- Due to IPsec (encryption/authentication)
- TCP segmentation offload (TSO)
- VLAN tagging/untagging

### ✓ 2. Packet Dropping

Reasons include:

- Firewall rules (DROP)
- No route
- Invalid checksum
- TTL expired
- Full queue (congestion)

### ✓ 3. Error Messaging

Kernel may send ICMP errors:

- Destination unreachable
- Time exceeded
- Fragmentation required
- Port unreachable (UDP case)

### ✓ 4. Packet Fragmentation

Happens when:

- Packet > MTU
- Don't Fragment (DF) flag decides behavior

### ✓ 5. Packet Defragmentation

- Happens for incoming packets
- Required for upper layers to process full data

### ✓ 6. Checksum Handling

May be done by:

- Kernel software
- NIC hardware (checksum offloading)

## 6. Why Only Layers 2, 3, 4 Are in Kernel?

The kernel must perform actions that require:

- High speed
- Low latency
- Direct hardware interaction
- Secure switching and routing

Layers 5–7 belong to applications and should run in userspace to keep:

- Kernel small
- System secure
- Easy debugging
- Flexibility

## 7. Summary

- Linux follows a **3-layer active model** inside the kernel:  
**Data Link (L2), Network (L3), Transport (L4)**.
- Physical and higher layers (L5–L7) are outside kernel scope.
- The stack is responsible for moving packets up/down layers.
- Many transformations (NAT, IPsec, routing, drops) occur along the path.
- Incoming → L2 → L3 → L4 → Userspace
- Outgoing → Userspace → L4 → L3 → L2 → NIC
- Linux can also forward packets like a router.

# EXTENSIVE + DETAILED NOTES ON THE NETWORK DEVICE & `net_device`

## 1. Layer 2 (Link Layer) in the Linux Network Stack

Layer 2 (L2) in the Linux stack corresponds to the **Data Link layer**.

This is where the **network device drivers** operate.

Examples of network devices:

- Ethernet NICs (eth0, eth1)
- WiFi devices (wlan0)
- Virtual devices (tap0, br0, vethX)
- VLAN interfaces

Linux does **NOT** perform hardware-level signal transmission (L1).

It interacts with the NIC at L2 using the driver.

## 2. What is a Network Device in Linux?

In the Linux kernel, every network interface (real or virtual) is represented by the structure:

`struct net_device`

This is one of the **central structures** in the kernel networking stack.

It contains:

- Device configuration
- MAC/MTU/IRQ information
- Function pointers for driver operations
- Queue information
- Timestamps, counters, flags

## 3. Why Understanding `net_device` is Important

Even though you are not writing a NIC driver, the entire kernel networking subsystem depends on this structure.

Examples:

- When deciding fragmentation → kernel checks device MTU
- When deciding forwarding → kernel checks device flags
- When transmitting packets → kernel calls device ops from net\_device\_ops
- NAPI polling → requires device's NAPI handlers

The networking stack cannot function without net\_device.

## 4. Key Device Parameters in net\_device

Here are the most important fields and what they mean.

### 4.1 Device IRQ Number

`unsigned int irq;`

- Each NIC raises interrupts to signal events:
- Packet received
- Transmission completed
- The IRQ number identifies which interrupt line the device uses.

Example:

- eth0 → IRQ 27
- wlan0 → IRQ 17

### 4.2 Device MTU

`unsigned int mtu;`

- Maximum Transmission Unit.
- For Ethernet: **1500 bytes** (default).
- Determines if a packet is:
- Sent as-is or
- Fragmented at L3 (IP layer)

If packet > MTU → IP fragmentation.

Examples:

- VLAN increases overhead → MTU becomes 1496
- Jumbo frames → MTU = 9000

### 4.3 MAC Address

`unsigned char *dev_addr;`

- The L2 address used for sending/receiving Ethernet frames.
- 6 bytes for Ethernet.

### 4.4 Device Name

Examples:

- eth0

- eth1
- ens33
- wlan0
- lo (loopback)

Name is created during registration using `register_netdev()`.

## 4.5 Device Flags

Flags tell the state of the device.

Examples:

- IFF\_UP → interface is up
- IFF\_PROMISC → promiscuous mode
- IFF\_RUNNING → link is active
- IFF\_MULTICAST → supports multicast
- IFF\_LOOPBACK → loopback interface

These flags affect how packets are **accepted, dropped, or processed**.

## 4.6 Multicast Address List

- Stores multicast MAC addresses that the NIC should listen for.
- Used for:
  - IPv4/IPv6 multicast
  - mDNS
  - Routing protocols (OSPF, RIP, VRRP)

NIC may filter multicast frames in hardware.

## 4.7 Promiscuity Counter (promiscuous mode)

`unsigned int promiscuity;`

### Purpose

When the counter > 0, the NIC accepts **all Ethernet frames**, including:

- Packets not addressed to local MAC
- Broadcast
- Multicast
- Other hosts' packets

### Why is it needed?

Used by:

- tcpdump
- wireshark
- intrusion detection systems
- bridge interfaces
- virtual switches (Open vSwitch)

### Why is it a counter and not a Boolean?

Because multiple sniffers can run at the same time.

Example:

- tcpdump starts → counter becomes 1
- wireshark starts → counter becomes 2
- tcpdump stops → counter becomes 1
- wireshark stops → counter becomes 0 → normal mode restored

## 4.8 Device Features (Offloading Capabilities)

Modern NICs support hardware offloading:

Common offloads:

- **GSO** (Generic Segmentation Offload)
- **GRO** (Generic Receive Offload)
- **TSO** (TCP Segmentation Offload)
- **LRO** (Large Receive Offload)
- **Checksum offload**

These improve performance by letting NIC hardware handle heavy tasks.

Example:

- NIC breaks 64KB TCP packet into MTU-sized chunks → CPU saved.

## 4.9 Network Device Operations (net\_device\_ops)

```
const struct net_device_ops *netdev_ops;
```

This is a table of **function pointers** implemented by device drivers.

Examples:

- .ndo\_open → open device
- .ndo\_stop → stop device
- .ndo\_start\_xmit → transmit packet
- .ndo\_change\_mtu → change MTU
- .ndo\_get\_stats64 → get statistics

Every driver must implement some of these.

## 4.10 Ehtool Operations

These callbacks allow ethtool command to query NIC information.

Examples:

- Get NIC speed (10/100/1000/10000 Mbps)
- Get driver name
- Set ring buffer sizes
- Enable/Disable offloading (checksum, tso, gso)

Command example:

```
ethtool eth0
```

## 4.11 RX and TX Queues

Modern NICs support multi-queue operation.

Why queues matter:

- Multiple CPUs can process packets in parallel
- Reduces lock contention
- Increases throughput

Examples:

- Queue 0 → CPU0
- Queue 1 → CPU1
- Queue 2 → CPU2

This is important for 10G/40G/100G NICs.

## 4.12 RX/TX Timestamps

- Timestamp of last received packet
- Timestamp of last transmitted packet
  - Used for:
- Statistics
- Detecting idle interfaces
- Network debugging

## 5. NAPI (New API) – Critical Concept

NAPI is the modern method of receiving packets efficiently.

### Why was NAPI introduced?

Old method:

- NIC used interrupts for every received packet
- At high traffic → interrupt storm → CPU overload

### NAPI Solution:

- Switch from interrupt-based to polling-based model
- NIC interrupts the CPU only once:
- CPU enables polling
- Kernel polls RX queue until empty
- Interrupt is re-enabled

### Benefits:

- Higher performance under load
- Interrupt mitigation
- Reduced CPU overhead
- Scales better for high packet rates

Most drivers today implement NAPI.

## 6. Example Snippet of net\_device Structure

From kernel headers:

```
struct net_device {  
    unsigned int irq;      /* device IRQ number */
```

```

const struct net_device_ops *netdev_ops;
unsigned int mtu;
unsigned int promiscuity;
unsigned char *dev_addr; /* MAC address */

...
};


```

The real structure is very large (~800+ lines).

Appendix A (as mentioned in the text) describes it in full detail.

## 7. Summary

- net\_device represents **every network interface**.
- Contains device parameters:
- IRQ, MTU, MAC, flags, multicast list
- Contains callback ops (netdev\_ops) implemented by NIC driver.
- Promiscuous mode is controlled using a **counter**, enabling multiple sniffers.
- Offloading features are stored in the structure and help performance.
- NAPI provides a scalable model for packet reception.
- Understanding net\_device is essential for understanding the Linux network stack.

Below are **very detailed, structured, deep, and beginner-friendly notes on NAPI (New API)**.

These expand your paragraph into a full conceptual explanation useful for interviews, kernel understanding, or driver development.

## EXTENSIVE + DETAILED NOTES ON NAPI (NEW API) IN NETWORK DEVICES

### 1. Why NAPI Was Created

Originally, Linux NIC drivers used a **pure interrupt-driven model**:

#### Old Model:

- NIC receives a packet
- NIC generates an interrupt
- Interrupt handler runs and processes packet
- Repeat for every packet

#### The Problem

When traffic is high (10k – 1M packets/sec):

- Interrupts occur at extremely high frequency
- CPU spends all time servicing interrupts
- This phenomenon is called “**interrupt storm**”

- CPU gets overwhelmed before processing packets
- Throughput collapses
- Latency becomes unpredictable

The old model works fine for low traffic but breaks at high packet rates.

This led to the invention of **NAPI (New API)**.

## 2. What Is NAPI?

NAPI is a **hybrid interrupt + polling** design for packet reception.

### Key Concept

- Light traffic → Interrupt-driven (low latency)
- Heavy traffic → Polling (high throughput)

NAPI dynamically switches between the two based on traffic conditions.

## 3. How NAPI Works: Step-by-Step

### Step 1: Packet Arrives

Normally, the NIC generates an interrupt for the first incoming packet.

### Step 2: Driver Disables Further Interrupts

The interrupt handler:

- Disables receive interrupts on the NIC
- Schedules a **NAPI poll** function
- Returns

This prevents further interrupt storms.

### Step 3: Kernel Starts Polling

Kernel enters polling mode using softirqs:

- It calls the driver's NAPI poll function repeatedly
- Poll handler processes multiple packets per poll cycle

Instead of page faults or thread switches, it uses **softirq (NET\_RX\_SOFTIRQ)**.

### Step 4: Poll Handler Drains RX Ring

NAPI poll function:

- Reads packets from NIC ring buffers
- Converts them into sk\_buff structures
- Pushes them into the upper networking stack

It processes packets **until**:

- The ring is empty OR
- The NAPI budget (max packets per cycle) is consumed

### Step 5: Polling stops + interrupts re-enabled

If the RX ring becomes empty:

- Poll handler calls napi\_complete()
- NIC interrupts are re-enabled

System returns to interrupt mode.

## 4. Benefits of NAPI

### 4.1 Major Benefit: Interrupt Mitigation

- Reduces interrupt rate drastically under load
- Prevents CPU saturation
- Increases total throughput

For example:

- Without NAPI → 200k IRQ/sec
- With NAPI → only 100 IRQ/sec even under heavy load

### 4.2 Packet Batching

NAPI polls packets in groups (batches).

Batched packet processing reduces:

- CPU overhead
- Cache misses
- Lock contention

### 4.3 Scalability for Multi-Gigabit NICs

10G/40G/100G devices absolutely require NAPI.

Without NAPI:

⚠ System becomes unusable due to interrupt overload.

With NAPI:

- ✓ High throughput is achievable
- ✓ Lower CPU usage
- ✓ Smooth latency profile

## 5. NAPI Structure and Registration

### Driver declares a napi\_struct:

```
struct napi_struct {  
    ...  
    int (*poll)(struct napi_struct *, int);  
    ...  
};
```

### Steps for a driver:

3. Initialize NAPI context:  
`netif_napi_add(dev, &napi, my_poll, weight);`
4. Enable NAPI:  
`napi_enable(&napi);`
5. Schedule NAPI poll on interrupt:  
`napi_schedule(&napi);`
6. Poll function drains packets
7. Complete polling:

```
napi_complete(&napi);
```

## 6. NAPI Budget

Each poll cycle has a “**budget**”, typically 64 packets.

Why?

- Prevents one device from starving others
- Ensures fairness among NICs
- Keeps softirq latency predictable

If poll consumes entire budget, kernel reschedules another poll cycle.

## 7. When Does NAPI Switch Modes?

**LOW traffic load**

- Interrupt-driven mode
- Instant, low-latency processing

**HIGH traffic load**

- Interrupt disabled
- Polling mode
- High throughput, stable performance

NAPI detects mode transition automatically based on the load on driver’s RX ring.

## 8. Busy Polling on Sockets (Kernel $\geq 3.11$ )

NAPI improves throughput.

BUT some applications (like HFT or real-time trading systems) need **microsecond-level latency**.

Linux added **Busy Polling**:

- Introduced in kernel 3.11
- Allows userspace to actively poll NIC queues
- Requires SO\_BUSY\_POLL or sysctl configuration

Tradeoff:

- Lower latency
- Higher CPU usage

Busy Polling is explained deeply in a later chapter.

## 9. Why NAPI Is Critical?

Because modern networks operate at:

- 1 Gbps  $\rightarrow$  up to 1.4M packets/sec
- 10 Gbps  $\rightarrow$  up to 14M packets/sec
- 100 Gbps  $\rightarrow$  >100M packets/sec

Pure interrupts cannot scale to these packet rates.

NAPI:

- Solves interrupt storm
- Improves throughput
- Reduces CPU overhead
- Became mandatory for modern NICs

This is why nearly **all modern drivers implement NAPI**.

## 10. Relationship Between NAPI and SoftIRQs

NAPI is tightly coupled with:

- NET\_RX\_SOFTIRQ
- napi\_schedule()
- \_\_napi\_schedule\_irqoff()

SoftIRQ runs NAPI pollers and is responsible for high-speed processing in the network stack.

## 11. Summary

- Older drivers used per-packet interrupts → inefficient under load
- NAPI is a hybrid interrupt + polling system
- At low traffic → interrupts
- At high traffic → polling
- Prevents interrupt storms
- Improves throughput and reduces CPU usage
- Uses softirqs for fast, non-blocking packet processing
- All modern NIC drivers implement NAPI
- Busy Polling is an advanced extension for ultra-low latency

### ✓ If you want next:

I can prepare **deep notes on packet traversal inside the Linux kernel network stack**, including:

- RX processing path (driver → netif\_receive\_skb → ip\_rcv → tcp\_rcv)
- TX processing path (tcp\_sendmsg → ip\_output → dev\_queue\_xmit → driver)
- Role of softirqs, RCU, GRO/GSO, XPS/RPS

Just tell me!

Below is a **super-detailed, beginner-friendly, well-structured set of notes** for your paragraph on **Receiving and Transmitting Packets in Linux Kernel Networking Stack**, suitable for your study notes.

## Extensive Notes: Receiving and Transmitting Packets in Linux Kernel Networking Stack

## 1. Core Responsibilities of a Network Device Driver

A network device driver performs two fundamental tasks:

### 1.1 Receiving Packets (Ingress / Rx Path)

- Receives packets **destined for the local host**.
- Passes them to:
- **Layer 3 (Network Layer, e.g., IPv4/IPv6)**
- Then to **Layer 4 (Transport Layer, e.g., TCP/UDP)**.

### 1.2 Transmitting Packets (Egress / Tx Path)

- Sends packets that the local host generates **out to the network**.
- Also used to **forward packets** if the host acts as a router.

## 2. Routing Lookup for Every Packet

Every incoming or outgoing packet undergoes a **routing subsystem lookup**.

### Why routing lookup is needed?

- To decide:
- **Should packet be forwarded or delivered locally?**
- **Which interface (device) it should be sent out through?**

Routing lookup logic is explained deeply in kernel chapters 5 & 6 (routing).

## 3. Netfilter: Hooks Influencing Packet Flow

Packet traversal is not determined only by routing.

Linux provides **netfilter hook points** where kernel modules can **intercept and modify** packets.

### 5 Key Netfilter Hook Points

For IPv4, the hooks are:

8. **NF\_INET\_PRE\_ROUTING**
9. **NF\_INET\_LOCAL\_IN**
10. **NF\_INET\_FORWARD**
11. **NF\_INET\_LOCAL\_OUT**
12. **NF\_INET\_POST\_ROUTING**

### First hook for incoming packets

- **NF\_INET\_PRE\_ROUTING**
- Triggered *before* routing lookup.
- Useful for NAT, filtering, alterations, etc.

### How netfilter works?

- Packets pass to callback using **NF\_HOOK()** macro.
- Callback returns a **verdict**:
- **NF\_DROP** → Drop packet
- **NF\_ACCEPT** → Continue normally
- **NF\_QUEUE** → Move to userspace
- **NF\_STOLEN** → Driver/module takes over

### Registration of callbacks

- **nf\_register\_hook()** → Register one hook
- **nf\_register\_hooks()** → Register multiple hooks

These are used mostly by firewall modules, NAT, connection tracking, etc.

### Userspace Integration

- Netfilter kernel subsystem supports **iptables** and **nftables**.

## 4. IPsec Influence on Packet Flow

The packet path can also change when IPsec rules match.

### IPsec Overview

- Provides **Network Layer Security**.
- Uses:
  - ESP (Encapsulating Security Payload)
  - AH (Authentication Header)
- Mandatory in IPv6, optional in IPv4 (but supported).

### Modes of Operation

#### 13. Transport Mode

- Encrypts only payload.

#### 14. Tunnel Mode

- Encrypts entire IP packet.

### Use Cases

- VPNs, secure tunnels, encrypted site-to-site communication.

### Issues with NAT

- NAT breaks IPsec (ports/addresses change).
- Kernel supports **NAT traversal (NAT-T)**.

Covered in chapter 10.

## 5. TTL/Hop Limit and Packet Forwarding

Packet forwarding also depends on TTL (IPv4) / hop\_limit (IPv6).

### TTL (IPv4)

- Decremented by 1 on every hop.
- When it reaches **0**:
- Packet is dropped.
- Host sends **ICMP Time Exceeded** (TTL Exceeded).

### Impact on header checksum

- IPv4 header checksum must be **recalculated** every time TTL changes.

### IPv6 Equivalent

- Field is called **hop\_limit**.
- No header checksum → simpler processing.

## 6. Complex Nature of Packet Traversal

Packet flow in the stack has many variations:

### 6.1 Fragmentation

- Large packets → fragmented before sending.
- Fragmented packets must be reassembled → on receive.

### 6.2 Different Packet Types

Each packet type follows special handling paths:

- **Unicast** → destined to one host.
- **Multicast** → delivered to a group.
- **Broadcast** → delivered to all.

### 6.3 Multicast Handling (IPv4)

- Uses **IGMP** for group membership.
- To join/leave multicast group, host must send IGMP messages.

#### 6.4 Multicast Routing

If kernel acts as **multicast router**:

- Must forward multicast, not deliver locally.
- Requires userspace daemons:
- pimd
- mrtouted

More details: Chapters 4 (IPv4), 6 (Routing).

## 7. Representation of Packets in Kernel: `sk_buff`

A packet in Linux kernel is stored in a structure called **socket buffer** or **SKB**.

#### Structure Name

- struct sk\_buff
- File: include/linux/skbuff.h

#### Contains

- All protocol headers (L2/L3/L4)
- Data payload
- Metadata: timestamps, device, length, checksum state, routing info, etc.

#### Importance

- Core unit of packet processing.
- Used in Rx and Tx paths across the networking stack.

#### Common Name

- Referred as **SKB** in kernel code and documentation.

## 8. Summary Diagram (Conceptual)

Incoming Packet (NIC)



Hardware / Driver



Netfilter: PRE\_ROUTING



Routing decision

→ local delivery? → LOCAL\_IN → L4/TCP/UDP

→ forward? → FORWARD → POST\_ROUTING → NIC



IPsec processing (if configured)



TTL/hop\_limit checks



Potential fragmentation / reassembly



Transmission or drop

Below are **very detailed, beginner-friendly, structured notes** for your paragraph on **The Socket Buffer (SKB) in Linux Kernel Networking**.

## EXTENSIVE NOTES: The Socket Buffer (SKB) in Linux Networking Stack

### 1. What Is the SKB (Socket Buffer)?

The **struct sk\_buff** is the **central data structure** representing a packet (both Rx & Tx) inside the Linux kernel networking stack.

It contains:

- All protocol headers (L2, L3, L4)
- Payload
- Metadata (routing info, device info, timestamps, checksum state)
- Pointers to where each header starts

It is the **heart** of packet processing in Linux.

SKB = “socket buffer” → usually called **SKB** everywhere in kernel code and docs.

### 2. MUST FOLLOW SKB API — Never Modify Members Directly

The SKB is complex and tightly controlled.

You **must not modify pointers manually**.

Example rules:

 Wrong

`skb->data += 14;`

 Correct

`skb_pull(skb, 14);`

#### Key SKB API Functions

Operation	Correct SKB API
Advance data pointer forward	<code>skb_pull(), skb_pull_inline()</code>
Add data to head	<code>skb_push()</code>
Add data to tail	<code>skb_put()</code>
Fetch transport (L4) header	<code>skb_transport_header()</code>
Fetch network (L3) header	<code>skb_network_header()</code>
Fetch MAC (L2) header	<code>skb_mac_header()</code>

The kernel requires these functions because:

- They maintain internal SKB invariants.
- Update length fields correctly.
- Avoid overruns / corruption.

### 3. Partial Structure of struct sk\_buff

```
struct sk_buff {  
    struct sock *sk;           // Associated socket (NULL for forwarded packets)  
    struct net_device *dev;    // Incoming/outgoing device  
  
    __u8 pkt_type:3;          // Packet type (host, broadcast, multicast...)  
    __be16 protocol;          // L3 protocol (ETH_P_IP, ETH_P_IPV6, etc.)  
  
    sk_buff_data_t tail;      // End of used data  
    sk_buff_data_t end;        // End of allocated buffer  
  
    unsigned char *head, *data; // head = start of buffer, data = current Lx header  
  
    sk_buff_data_t transport_header; // Offset to L4 header  
    sk_buff_data_t network_header; // Offset to L3 header  
    sk_buff_data_t mac_header;   // Offset to L2 header  
};
```

Important members explained:

#### sk

- Points to the socket that generated the packet (Tx path).
- **NULL for forwarded packets** (router mode).

#### dev

- The network interface:
- Rx path → incoming dev
- Tx path → outgoing dev
- Used for MTU, device type, offload capabilities.

#### pkt\_type

Determines packet destination type based on MAC:

- PACKET\_HOST – packet for this host
- PACKET\_BROADCAST
- PACKET\_MULTICAST
- PACKET\_OTHERHOST

#### protocol

- From Ethertype: e.g.,
- 0x0800 = IPv4
- 0x86DD = IPv6
- 0x0806 = ARP

Set by `eth_type_trans()`.

#### **head / data**

- head = start of buffer memory
- data = pointer to current protocol header
- Values change as the packet moves between layers.

#### **mac/network/transport\_header**

Offsets to L2, L3, L4 headers so kernel can quickly access them.

## **4. Allocation & Freeing of SKBs**

### **Allocation**

Network drivers allocate SKBs when a packet arrives:

- `netdev_alloc_skb()`
- `dev_alloc_skb()` (legacy wrapper)

These allocate a packet buffer and prepare the SKB structure.

### **Freeing**

When the packet is discarded:

- `kfree_skb()`
- `dev_kfree_skb()`

Only these functions may free SKBs.

## **5. Role of `eth_type_trans()` in NIC Rx Path**

Almost all Ethernet drivers call this in Rx path.

What it does:

#### **15. Determines `pkt_type`**

- Broadcast? → `PACKET_BROADCAST`
- Multicast? → `PACKET_MULTICAST`
- Host MAC? → `PACKET_HOST`

#### **16. Sets `protocol` field**

- Reads Ethertype from L2 header.

#### **17. Advances `skb->data` pointer**

- Moves past the 14-byte Ethernet header.
- Uses `skb_pull_inline()` → correct API usage.
- Now data points to the L3 (IP) header.

#### **Why is this necessary?**

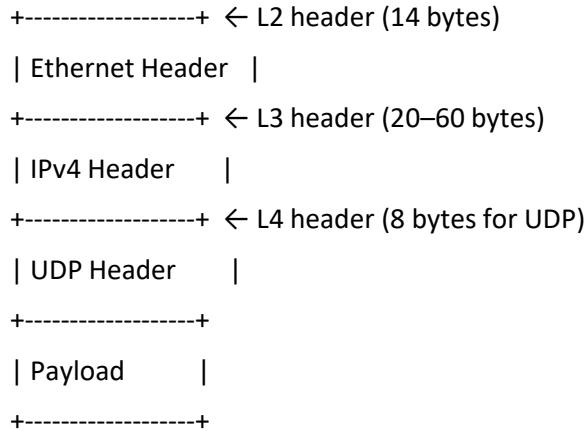
Because as packet moves up the stack:

- L2 header → removed
- L3 header → processed
- L4 header → eventually processed

SKB always adjusts to point to the "current" layer.

## **6. Header Layout Inside SKB**

Example: IPv4 + UDP packet



As packet moves between layers:

- L2 header removed by `eth_type_trans()`
- L3 processed by `ip_rcv()`
- L4 processed by `udp_rcv() / tcp_v4_rcv()`

SKB API makes this safe and organized.

## 7. Using dev Member of SKB

The SKB's dev member influences packet handling in many ways:

Examples:

- **MTU** decides if fragmentation is needed.
- Device type (Ethernet, loopback, tunnel).
- Offloading capabilities (checksum offload, TSO, GRO).

For Rx path → incoming device

For Tx path → outgoing device

## 8. sk Member: Associated Socket

- If packet originated locally (Tx path):
- `sk` = socket that generated it.
- If packet is forwarded:
- `sk` = NULL.

Used for:

- Socket accounting
- Congestion control
- Packet ownership

## 9. Passing to the Network Layer

Incoming packets are delivered based on protocol:

- IPv4 → `ip_rcv()`
- IPv6 → `ipv6_rcv()`

The handlers:

18. Perform sanity checks.
19. Call **NF\_INET\_PRE\_ROUTING** netfilter hook.
20. If accepted → call ip\_rcv\_finish().
21. Routing lookup performed → dst\_entry created.

## 10. Routing & Destination Cache

Routing lookup produces:

- A **dst\_entry** object containing:
- Next-hop info
- Output function
- Input function
- Security checks
- MTU information
- Routing flags

This determines:

- Whether packet is delivered locally
- Or forwarded
- And how it should be transmitted

Covered in chapters 5 & 6.

## 11. IPv4 vs IPv6 Details (Important Differences)

### IPv4:

- Address length: 32-bit → exhausted → used with NAT.
- Header: variable size (20–60 bytes).
- Header checksum required.
- Options slow and rarely used.

### IPv6:

- Address length: 128-bit → huge space.
- Header: fixed 40 bytes → fast processing.
- No header checksum → faster.
- Uses **extension headers** instead of options (modular design).
- ICMPv6 heavily expanded:
- Neighbor Discovery
- Router Discovery
- MLD (Multicast Listener Discovery)

Relevant chapters:

- ICMP (ch. 3)
- ND (ch. 7)
- MLD (ch. 8)

## 12. Packet Delivery to Transport Layer

If packet is for local host:

- Delivered from IP layer to L4 protocol handler.

Examples:

- UDP → udp\_rcv()
- TCP → tcp\_v4\_rcv()
- SCTP → sctp\_rcv()
- DCCP → dccp\_rcv()

These protocols are described in chapter 11.

## 13. Packets Generated Locally (Tx Path)

Steps:

22. Created by userspace via **socket API**.
23. Handled by L4 layer (TCP/UDP).
24. Passed to network layer (IPv4/IPv6).
25. IP layer:
  - Adds headers
  - Fragmentation if needed
  - Routes packet
26. Passed to L2 driver.
27. NIC transmits to the wire.

Local Tx may involve fragmentation depending on MTU.

## 14. Summary Diagram of SKB in Networking Stack

Userspace Socket



TCP/UDP (L4) creates SKB



IP Layer (L3)

- Add IP header
- Routing lookup
- Fragment if needed



NIC Driver (L2)

- Add MAC header



Wire

For Rx path just reverse the flow.

Below are **extremely detailed, structured, beginner-friendly notes** based entirely on the paragraph you provided.

These notes explain every concept slowly and deeply so you can understand the full picture of modern Linux networking.

## Extensive Notes on: Layer-2 Addresses, Ethernet Headers, Neighbour Discovery, Netlink, Wireless Networking, InfiniBand, Virtualization, Namespaces & More

### 1. Layer-2 Addressing (MAC Addresses)

#### 1.1 What is a Layer-2 (L2) Address?

- Every network interface working at **Layer 2 (Data Link Layer)** has an L2 address.
- For **Ethernet**, the L2 address is the **MAC address**.
- MAC address = **48-bit value** (6 bytes) written as:  
AA:BB:CC:DD:EE:FF

#### 1.2 Assignment

- Typically **factory-assigned** by the NIC manufacturer.
- Said to be globally unique.
- But:
- Linux allows changing the MAC using ifconfig, ip link set dev eth0 address <mac>.

### 2. Ethernet Header (14 bytes)

Every Ethernet frame starts with a **14-byte header**:

Field	Size	Purpose
Destination MAC	6 bytes	Who the packet is meant for
Source MAC	6 bytes	Who is sending the packet
EtherType	2 bytes	Indicates what protocol payload follows

#### 2.1 Common EtherType Values

Protocol	EtherType
IPv4	0x0800
IPv6	0x86DD
ARP	0x0806

#### 2.2 For Every Outgoing Packet

- Linux must **construct the Ethernet header**.
- The sender knows:
- Source MAC (its own)
- EtherType (depends on L3 protocol)
- But **destination MAC** must be discovered → this requires **Neighbour Discovery**.

### 3. Neighbour Discovery (Finding MAC from IP)

When a user program sends data using an IP address, Linux must map:

IP Address → MAC Address

This is handled by the **Neighbouring Subsystem (Chapter 7)**.

### 3.1 IPv4: ARP (Address Resolution Protocol)

- Uses **broadcast**:
- ARP request: "Who has 192.168.1.5? Tell 192.168.1.2"
- Target replies with its MAC.
- ARP entries are stored in the **ARP table**.

### 3.2 IPv6: NDISC (Neighbour Discovery Protocol)

- Uses **ICMPv6 messages** which are:
- **Multicast**, not broadcast.
- Performs:
  - Neighbour Discovery
  - Router Discovery
  - Prefix information
  - Duplicate Address Detection (DAD)

## 4. Userspace–Kernel Communication: Netlink Sockets

### 4.1 Why Netlink?

The networking stack must communicate with userspace for:

- Adding/removing routes
- Managing neighbor tables (ARP/NDISC)
- Configuring IPsec policies
- Managing interfaces

### 4.2 Netlink Socket

- A special IPC mechanism for kernel ↔ userspace communication.
- Used by **iproute2** tools (ip, ss, tc, etc.)

### 4.3 Generic Netlink

- More flexible extension of Netlink.
- Lets subsystems create custom families, commands, and attributes.

## 5. Wireless Networking Subsystem (Chapter 12)

The wireless subsystem is **independent** from wired networking.

### 5.1 Why separate?

- Wireless requires additional mechanisms:
- Power saving
- Dynamic bandwidth management
- Authentication/association
- Frequency/channel hopping

### 5.2 Special Wireless Topologies

- **Mesh networking**
- **Ad-hoc mode**
- **Access point (AP) mode**

### 5.3 Mesh Networking (HWMP)

- HWMP (Hybrid Wireless Mesh Protocol):
- A Layer-2 routing protocol.
- Works with **MAC addresses**, not IP.
- Enables multi-hop communication.

## 5.4 mac80211 Framework

- Used by most Wi-Fi drivers.
- Handles complex Wi-Fi behaviors:
- Authentication
- Encryption
- Association
- Scanning
- Block acknowledgements (802.11n)

## 5.5 Block Acknowledgment (802.11n)

- Improves performance by acknowledging **bulk frames** instead of per-frame ACK.

# 6. InfiniBand & RDMA (Chapter 13)

## 6.1 What is InfiniBand?

- A high-performance networking technology used in data centers.
- Extremely low latency, high throughput.

## 6.2 RDMA (Remote Direct Memory Access)

- Allows one machine to write directly into another machine's memory **without CPU involvement**.
- Linux added RDMA API in kernel **2.6.11**.

## 6.3 Used in

- High-performance computing (HPC)
- Datacenters
- Storage systems

## 6.4 RDMA API

- Provides APIs + data structures to send/receive data via DMA.
- Bypasses TCP/IP stack in some cases for speed.

# 7. Virtualization in Linux

## 7.1 Types of Virtualization

### 28. Full Virtualization

- Xen, KVM

### 29. Hardware-Assisted Virtualization

- Intel VT-x
- AMD-V

### 30. Process-level Virtualization (Namespaces)

- Lightweight
- Foundation of:
- Docker
- LXC
- CRIU

# 8. Namespaces (Chapter 14)

## 8.1 Purpose

Namespaces isolate system resources between processes:

- Network interfaces
- Routing tables
- Filesystems
- PIDs

- UIDs
- IPC
- Mounts

### **8.2 Namespaces-related System Calls**

- unshare()
- setns()

### *8.3 CLONE\_ Flags\**

Examples:

- CLONE\_NEWNET → Network namespace
- CLONE\_NEWPID → PID namespace

Linux supports **6 namespaces**, and more may be added.

## **9. Other Advanced Topics (Chapter 14)**

### **9.1 Bluetooth Subsystem**

- Manages Bluetooth devices, protocols (L2CAP, RFCOMM), and profiles.

### **9.2 PCI Subsystem (Brief Overview)**

- Most NICs are PCI devices.
- Provides:
- Configuration space
- Device discovery
- Interrupt routing

### **9.3 IEEE 802.15.4**

- Low-power, low-cost wireless standard.
- Used for:
- IoT
- ZigBee
- Sensor networks

### **9.4 6LoWPAN (IPv6 over IEEE 802.15.4)**

- Compresses IPv6 headers to fit tiny wireless frames.
- Adapts IPv6 Neighbor Discovery for sleeping devices.
- Used in low-power embedded devices.

## **10. Understanding Linux Networking Stack Development**

To understand or contribute to the Linux networking stack, you must know:

- How subsystems are organized
- How patches are submitted
- How the maintainer workflow works
- How networking trees are merged

Typically:

- Networking subsystem maintained by netdev team.
- Code patch workflow:
- Submit patch → Reviewed on mailing list → Accepted → Upstream kernel.

**If you want, I can also create:**

- A mind map
- A flow chart
- A simplified beginner version
- A highly advanced deep-dive version
- Notes for each subtopic separately

Just tell me!

# Netlink sockets

Monday, December 1, 2025 12:51 PM

Okay. I'll explain **Netlink sockets** cleanly, from absolute beginner level — **what they are, why they exist, types, use-cases, advantages, disadvantages**, everything.

## What is a Netlink Socket? (Beginner Explanation)

Netlink is a **special communication mechanism between user space and kernel** in Linux.

Think of it as a **phone line** the kernel and your application can use to talk to each other.

It is NOT a normal TCP/UDP socket.

It is a **kernel <-> user communication socket**.

## Why does Netlink exist?

Before netlink, communication with the kernel was done using:

- ioctl
- /proc or /sys files
- system calls

These were:

- inflexible
- hard to extend
- not suitable for asynchronous/multicast messages

So Linux introduced **netlink**, which is:

- structured
- extensible
- asynchronous
- bidirectional

## Where is Netlink used?

Many core Linux subsystems use netlink:

Subsystem	Why they use Netlink
Network stack	Routing table updates, link up/down events
Netfilter/iptables	Add/remove firewall rules
Wireless (cfg80211)	WiFi kernel events
Routing daemon (FRR/Quagga)	Route updates
Process events (Netlink Connect)	Process creation/death events
Generic netlink	Custom kernel module communication

## Real-world examples:

✓ When link goes UP or DOWN, kernel sends netlink event

NetworkManager listens using netlink.

✓ iproute2 uses netlink

ip addr add, ip route show = all talk to kernel using netlink.

✓ Custom kernel modules

Netlink is used to send:

- statistics
- debug messages
- commands

## Types of Netlink Protocols (Netlink Families)

Linux defines many netlink "protocols" (families).

Each number corresponds to a specific subsystem.

Name	Purpose
NETLINK_ROUTE	Routing tables, interface configuration
NETLINK_NFTILTER	iptables, firewall
NETLINK_GENERIC	Custom protocols for kernel modules
NETLINK_KOBJECT_UEVENT	Hotplug events (USB, device add/remove)
NETLINK_SOCK_DIAG	Socket monitoring
NETLINK_AUDIT	Linux audit system
NETLINK_CONNECTOR	Process events (fork, exit)
NETLINK_XFRM	IPsec

★ For beginners, the two most important are:

1. NETLINK\_ROUTE
2. NETLINK\_GENERIC

## How Netlink Works (Simplified)

Everything revolves around:

- **nlmsghdr** = netlink message header
- **payload** = data
- **attributes (TLVs)**

Kernel and user space exchange these messages via a netlink socket:

```
socket(AF_NETLINK, SOCK_RAW, NETLINK_GENERIC);
```

## Netlink Communication Types

- ✓ 1. Unicast

User → Kernel

Kernel → User

## ✓ 2. Multicast

Kernel → many user applications

Example:

network link down event sent to NetworkManager + ifupdown + systemd-networkd

## Advantages of Netlink

### ★ 1. Asynchronous

Kernel can push events to user space immediately.

(e.g., link down event)

### ★ 2. Multi-cast support

Many listeners can receive kernel events.

### ★ 3. Extensible

You can create your own netlink protocol using **Generic Netlink**.

### ★ 4. Structured messaging (TLV format)

Much better than ioctl.

### ★ 5. Rich networking functionality

Everything in modern networking uses netlink.

## Disadvantages of Netlink

### ! 1. Complex to implement

Structures like:

- nlmsghdr
- genlmsghdr
- nlaattr

are confusing for beginners.

### ! 2. Not guaranteed delivery

Kernel → user messages can be lost if user doesn't read fast enough.

### ! 3. No automatic versioning

Developers must handle compatibility manually.

### ! 4. Multi-cast groups require synchronization

Race conditions possible.

## When Should You Use Netlink?

Use it when you need:

- ✓ Custom kernel <-> user communication
- ✓ Kernel events delivered instantly
- ✓ Networking stack configuration
- ✓ Multicast kernel notifications

## When NOT to use Netlink?

Don't use netlink for:

- ✗ transferring large data
- ✗ simple configuration (sysfs is easier)
- ✗ everything that can be done via normal system calls
- ✗ real-time constraints (messages can drop)

## Simplest Possible Use Case (Beginner Friendly)

Use case:

You write a kernel module that counts packets and want a user program to read the count.

Netlink is perfect for this.

User app sends:

GET\_COUNT

Kernel replies:

packet\_count = 1234

If you want, I can explain next:

- ✓ how to create a **user-space netlink program**

Got it.

You want **step-by-step userspace Netlink socket workflow**, and for **each step** you want:

- ✓ Corresponding userspace Netlink function
- ✓ Full struct definitions
- ✓ Explanation of every member
- ✓ Explanation of every argument
- ✓ What the function does internally

I'll now explain **userspace Netlink sockets** from zero, in a clean, layered, extremely detailed way.

## 1. What is a Userspace Netlink Socket?

A userspace Netlink socket is a **special IPC (Inter-process communication)** mechanism used by a userspace process to communicate with the **Linux kernel**.

It uses:

AF\_NETLINK  
SOCK\_RAW or SOCK\_DGRAM

Unlike normal sockets (TCP/UDP), Netlink is:

- **Bidirectional** (userspace ↔ kernel)
- **Asynchronous** (messages arrive anytime)
- **Structured** (messages wrapped in ‘struct nlmsghdr’)
- **Used mostly for kernel configuration & control**

## 2. Steps to Create and Use a Netlink Socket in Userspace

Below are the exact steps you follow:

### Step 1 — Create a Netlink Socket

Function:

```
int socket(int domain, int type, int protocol);
```

For Netlink:

```
int sock_fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
```

#### Arguments Explained

Argument	Meaning	Explanation
AF_NETLINK	Netlink family	Tells Linux we want a netlink socket
SOCK_RAW / SOCK_DGRAM	Message mode	RAW gives message headers, DGRAM trims it
NETLINK_ROUTE	Protocol	Which kernel subsystem to talk to

## Protocols Used in Netlink

Some common ones:

Protocol ID	Macro	What it Talks To
0	NETLINK_ROUTE	Routing, link config, IP addr, neigh table
1	NETLINK_UNUSED	Reserved
2	NETLINK_USERSOCK	Generic userspace communication
3	NETLINK_FIREWALL	Netfilter communication
5	NETLINK_NFLOG	Netfilter logging
9	NETLINK_KOBJECT_UEVENT	Hotplug/udev events
15	NETLINK_GENERIC	Generic Netlink subsystem

## Step 2 — Bind the Socket

Function:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

**For Netlink:**

```
struct sockaddr_nl useraddr;  
memset(&useraddr, 0, sizeof(useraddr));  
  
useraddr.nl_family = AF_NETLINK;  
useraddr.nl_pid = getpid(); // Unique PID for this userspace socket  
useraddr.nl_groups = 0; // Multicast groups (none for now)  
  
bind(sock_fd, (struct sockaddr*)&useraddr, sizeof(useraddr));
```

## 🔍 STRUCT — struct sockaddr\_nl

This structure identifies the **address** of a netlink socket.

```
struct sockaddr_nl {  
    __kernel_sa_family_t nl_family; // Always AF_NETLINK  
    unsigned short nl_pad; // Padding for alignment, always zero  
    __u32 nl_pid; // Port ID (process ID)  
    __u32 nl_groups; // Multicast group mask  
};
```

Explanation of members:

Member	Type	Purpose
nl_family	sa_family_t	Must be AF_NETLINK
nl_pad	u16	Not used, keep 0
nl_pid	u32	Unique identifier of your socket (usually PID)
nl_groups	u32	Bitmask of multicast groups to join

## ☑ Step 3 — Prepare the Netlink Message

To send anything, userspace builds two main headers:

### 3.1. Netlink message header — struct nlmsghdr

This goes in front of all Netlink messages.

```
struct nlmsghdr {  
    __u32 nlmsg_len; // Total length: header + payload  
    __u16 nlmsg_type; // Message type (depends on protocol)  
    __u16 nlmsg_flags; // Flags  
    __u32 nlmsg_seq; // Sequence number  
    __u32 nlmsg_pid; // Sending process PID  
};
```

## 🔍 Member Explanations

Field	Meaning	Explanation
nlmsg_len	Message length	Must include full payload
nlmsg_type	Message type	E.g. RTM_NEWLINK, RTM_DELADDR, etc.
nlmsg_flags	Flags	E.g. NLM_F_REQUEST, NLM_F_DUMP
nlmsg_seq	Sequence number	Used for batch requests
nlmsg_pid	Sender's PID	Kernel fills this for messages to userspace

## 3.2. Payload

Below nlmsghdr, you append **protocol-specific** payload.

Example for NETLINK\_ROUTE: a struct rtmmsg, or ifinfomsg, etc.

## Step 4 — Send Message to Kernel

Function:

```
int sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Build msghdr

```
struct nlmsghdr *nlh = malloc(NLMSG_SPACE(1024));  
  
nlh->nlmsg_len = NLMSG_LENGTH(payload_len);  
nlh->nlmsg_type = RTM_GETLINK;  
nlh->nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;  
nlh->nlmsg_seq = 1;  
nlh->nlmsg_pid = getpid();
```

## 🔍 STRUCT — struct msghdr

This wraps your netlink message so kernel can receive it.

```
struct msghdr {  
    void     *msg_name;    // Pointer to sockaddr_nl (destination)  
    int      msg_namelen; // Length of above  
    struct iovec *msg iov; // Pointer to array of buffers  
    size_t   msg iovlen; // Number of buffers  
    void     *msg control; // Ancillary data (rarely used)  
    size_t   msg controllen; // Length of control data  
    int      msg_flags; // Flags, not used by user  
};
```

### Detailed Explanation:

Field	Purpose
msg_name	Address of kernel netlink socket (sockaddr_nl with nl_pid=0)
msg_namelen	Size of sockaddr_nl
msg iov	Buffers containing the message (header + payload)
msg iovlen	Number of buffers
msg_control	Extra control data (not used usually)
msg_flags	For kernel-return flags

## Step 5 — Receive Reply From Kernel

Function:

```
int recvmsg(int sockfd, struct msghdr *msg, int flags);
```

You prepare buffers similar to sendmsg.

Receive loop:

```
int len = recvmsg(sock_fd, &msg, 0);
```

```
for (nlh = (struct nlmsghdr*)buf; NLMSG_OK(nlh, len);
     nlh = NLMSG_NEXT(nlh, len))
{
    // process nlh
}
```

Kernel will send:

- nlmsg\_type = NLMSG\_ERROR → contains errors
- nlmsg\_type = NLMSG\_DONE → final message in a dump
- Protocol-specific messages

## Step 6 — Close the Socket

```
close(sock_fd);
```



## COMPLETE STRUCT LIST (Userspace Netlink)

Here is every important struct:

### 1. struct sockaddr\_nl

Used in bind(), sendmsg().

```
struct sockaddr_nl {
```

```

__kernel_sa_family_t nl_family;
unsigned short    nl_pad;
__u32          nl_pid;
__u32          nl_groups;
};


```

## 2. struct nlmsghdr

Top-level header for all netlink messages.

```

struct nlmsghdr {
    __u32 nlmsg_len;
    __u16 nlmsg_type;
    __u16 nlmsg_flags;
    __u32 nlmsg_seq;
    __u32 nlmsg_pid;
};


```

## 3. struct msghdr

For sendmsg/recvmsg.

```

struct msghdr {
    void      *msg_name;
    int       msg_namelen;
    struct iovec *msg_iov;
    size_t    msg_iovlen;
    void      *msg_control;
    size_t    msg_controllen;
    int       msg_flags;
};


```

## 4. struct iovec

Describes data buffers.

```

struct iovec {
    void *iov_base; // Buffer start
    size_t iov_len; // Buffer length
};


```

## ✓ Advantages of Userspace Netlink Sockets

Advantage	Explanation
Fast	Kernel-user IPC avoids filesystem overhead
Asynchronous	Kernel can push messages anytime
Multi-cast	Many processes can receive events
Structured	Strong header + payload format
Extensible	New families (generic netlink etc.)

## ✗ Disadvantages

Disadvantage	Explanation
Complex	Harder than normal sockets
Not for streaming	Meant for small messages only
Kernel-side support required	You must write kernel netlink handler
No ordering guarantee	Messages can reorder

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <linux/netlink.h>

#define NETLINK_USER 31 // custom protocol (to be shared with kernel)
#define MAX_PAYLOAD 1024 // maximum message size

int main()
{
    int sock_fd;
    struct sockaddr_nl src_addr, dest_addr;

    struct nlmsghdr *nlh = NULL;
    struct iovec iov;
    struct msghdr msg;

    // 1. Create netlink socket
    sock_fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_USER);
```

```

if (sock_fd < 0) {
    perror("socket");
    return -1;
}

// 2. Bind our userspace socket
memset(&src_addr, 0, sizeof(src_addr));
src_addr.nl_family = AF_NETLINK;
src_addr.nl_pid = getpid(); // unique PID used as Netlink port ID

if (bind(sock_fd, (struct sockaddr *)&src_addr, sizeof(src_addr)) < 0) {
    perror("bind");
    return -1;
}

// 3. Destination = kernel
memset(&dest_addr, 0, sizeof(dest_addr));
dest_addr.nl_family = AF_NETLINK;
dest_addr.nl_pid = 0; // 0 means "kernel"
dest_addr.nl_groups = 0; // unicast

// 4. Allocate netlink message buffer
nlh = (struct nlmsghdr *)malloc(NLMSG_SPACE(MAX_PAYLOAD));
memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));

// Fill header
nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
nlh->nlmsg_pid = getpid();
nlh->nlmsg_flags = 0;
nlh->nlmsg_type = 0x01; // custom type

// Fill payload
strcpy(NLMSG_DATA(nlh), "Hello Kernel from Userspace!");

// 5. iovec setup - maps buffer to sendmsg()
iov.iov_base = (void *)nlh;
iov.iov_len = nlh->nlmsg_len;

```

```
// 6. Set msg header for sendmsg()
memset(&msg, 0, sizeof(msg));
msg.msg_name = (void *)&dest_addr;
msg.msg_namelen = sizeof(dest_addr);
msg.msg iov = &iov; // iovec buffer
msg.msg iovlen = 1; // number of iovecs

// 7. Send message to kernel
printf("Sending message to kernel...\n");
sendmsg(sock_fd, &msg, 0);

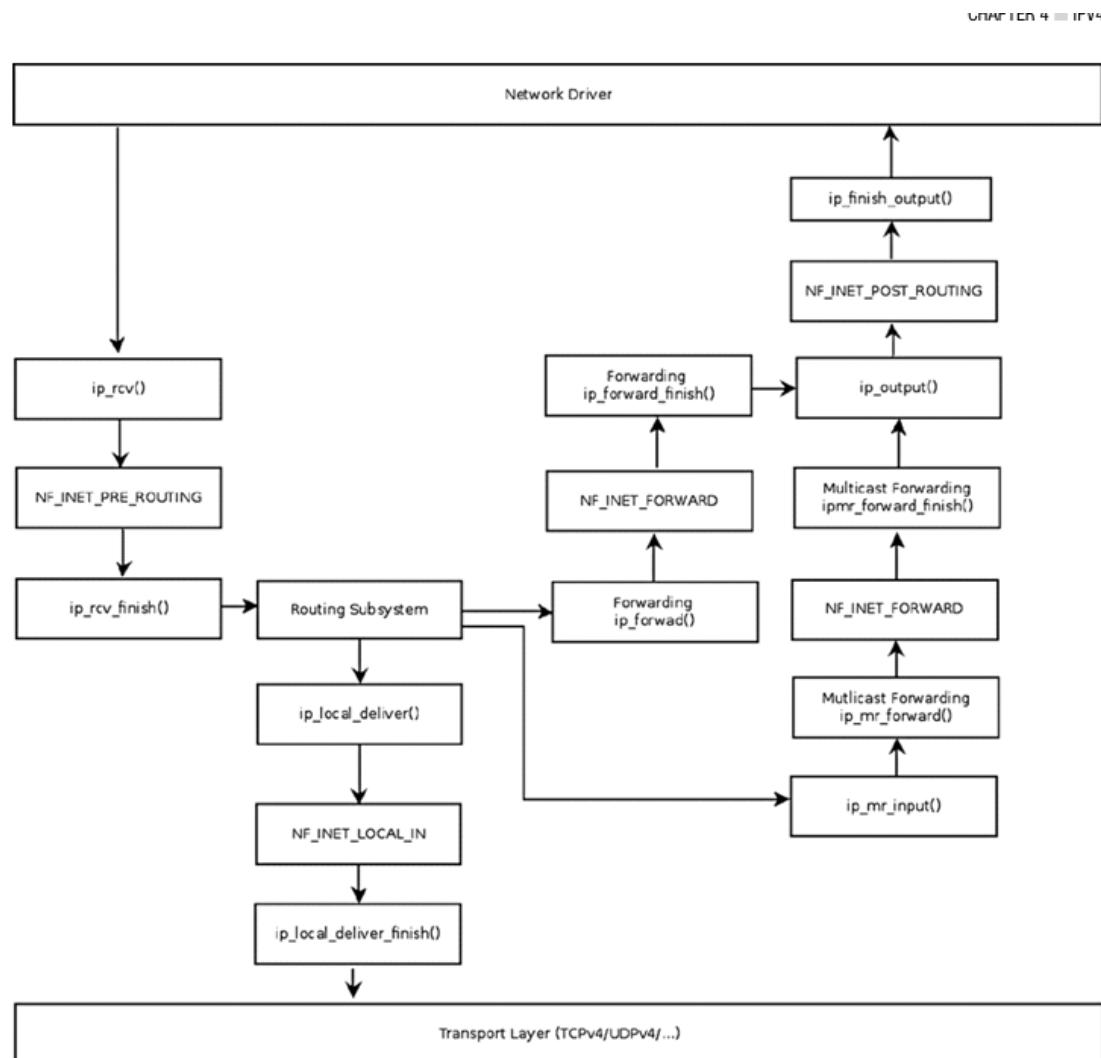
// 8. Receive reply from kernel
printf("Waiting for message from kernel...\n");
recvmsg(sock_fd, &msg, 0);

printf("Received message payload: %s\n",
(char *)NLMSG_DATA(nlh));

close(sock_fd);
return 0;
}
```

# Ipv4

Monday, December 1, 2025 1:16 PM



**Figure 4-2.** Receiving IPv4 packets. For simplicity, the diagram does not include the fragmentation/defragmentation/options/IPsec methods

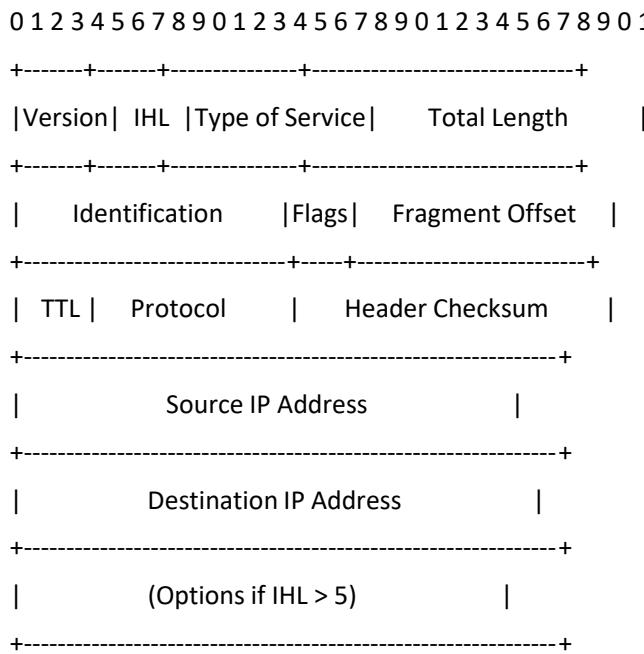
Below is a **clear, beginner-friendly but fully deep explanation** of the **IPv4 header** — *every field*, why it exists, how drivers and kernel networking stack use it, and how NIC hardware interprets it for checksum offload.

## IPv4 Header (20–60 bytes)

Standard IPv4 header (without options) is **20 bytes**.

If options exist, it may go up to **60 bytes**.

0            1            2            3



## ■ BYTE-BY-BYTE EXPLANATION

### 1 Version (4 bits)

Value = **4** for IPv4.

Tells the receiver what parsing rules to use.

Kernel uses:

`iph->version`

### 2 IHL – Internet Header Length (4 bits)

Length of header in **32-bit words**:

- Minimum = 5 words → 20 bytes
- Maximum = 15 words → 60 bytes

If IHL > 5 → Header contains options.

Kernel uses this to find the L4 header:

`ihl_bytes = iph->ihl * 4;`

### 3 Type of Service (ToS) / DSCP + ECN (8 bits)

6 bits → DSCP (Differentiated Services)

2 bits → ECN (Explicit Congestion Notification)

Used for QoS, priority traffic, congestion algorithms.

Examples:

- 0x00 → normal
- 0xB8 → Expedited Forwarding (EF), VoIP
- ECN bits marked by routers for congestion

Kernel field:

iph->tos

#### 4 Total Length (16 bits)

Total IPv4 packet size = header + payload.

Minimum = 20 bytes

Maximum = 65,535 bytes

Most NICs use this for:

- DMA length verification
- Hardware checksum

Kernel:

iph->tot\_len

#### 5 Identification (16 bits)

Used for **fragmentation** and **reassembly**.

All fragments of the same original packet share the same Identification.

When a packet is split:

Fragment #1 → ID = 0x1234

Fragment #2 → ID = 0x1234

...

#### 6 Flags (3 bits)

bit 0: Reserved (must be 0)

bit 1: DF – Don't Fragment

bit 2: MF – More Fragments

Meaning:

- **DF=1** → router cannot fragment (used by Path MTU Discovery)
- **MF=1** → more fragments follow
- **MF=0 and offset=0** → last fragment

Kernel checks:

iph->frag\_off

#### 7) Fragment Offset (13 bits)

- If a packet is fragmented, this tells the receiver **where this piece belongs**.
- The offset is in units of **8 bytes** (because fragments must align to 8 bytes).

Simple meaning:

“This piece starts at byte position X of the original packet.”

#### 8 TTL – Time To Live (8 bits)

Decrement by every router.

If TTL hits 0 → packet dropped to avoid loops.

## 9 Protocol (8 bits)

Indicates which **upper-layer protocol** to pass payload to.

Common values:

Protocol	Value
ICMP	1
TCP	6
UDP	17
GRE	47
ESP	50

Kernel reads:

`iph->protocol`

## 10 Header Checksum (16 bits)

Checksum **only over the IPv4 header**, not the payload.

Routers recalculate it after modifying TTL.

Ethernet NICs with offload:

- If TX checksum offload enabled, driver sets flag so NIC computes this.
- Otherwise kernel computes via:  
`ip_fast_csum()`

## 1 1 Source IP Address (32 bits)

Self-explanatory.

Kernel sets during transmission after routing.

## 1 2 Destination IP Address (32 bits)

Used by routing + neighbor (ARP/NDP) subsystem.

## 1 3 Options (0–40 bytes)

Only present when IHL > 5.

Types of options:

- Record Route
- Timestamp
- Strict/Loose Source Routing
- Security
- Router Alert
- Padding (to align to 32-bit)

Modern networks rarely use options because routers slow down packet processing.

## SUMMARY TABLE

Field	Size	Purpose
Version	4 bits	IPv4 = 4
IHL	4 bits	Header length
ToS/DSCP/ECN	8 bits	QoS
Total Length	16 bits	Entire IPv4 packet length
Identification	16 bits	Fragmentation
Flags	3 bits	DF/MF
Fragment Offset	13 bits	Fragment position
TTL	8 bits	Loop prevention
Protocol	8 bits	L4 protocol (TCP/UDP/etc)
Header Checksum	16 bits	Header integrity
Source IP	32 bits	Sender IP
Destination IP	32 bits	Receiver IP
Options	variable	Extra info

## Want Next?

I can provide:

- Full **byte-by-byte annotated hex example** of an IPv4 header
- How the **Linux kernel parses it** (functions & code flow)
- How a **NIC parses IPv4 header for checksum offload**
- Where **skb->protocol**, **iph**, and routing interact with IPv4 header
- “Walk me through an actual IPv4 packet from Wireshark”
- Complete **TCP + IPv4 + Ethernet** combined header explanation

Just tell me.

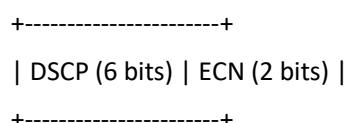
Below is a **very clean, detailed but simple explanation** of **DSCP + ECN** — what they are, why they exist, how each bit works, how routers/kernel/NICs use them, and how they appear inside the IPv4 header.

No unnecessary theory.

Just *exactly what you need*.

## Where DSCP + ECN live in the IPv4 header

They are inside the **second byte** of the IPv4 header (the old “TOS field”):





## PART 1 — DSCP (6 bits)

### DSCP = Differentiated Services Code Point

It is a **6-bit value** that marks a packet with a *priority or behavior*.

Why 6 bits?

→ Because it can now represent **64 different traffic classes**.

DSCP is used by:

- Routers
- QoS engines
- Switch queues
- Traffic shaping systems

It does NOT affect routing.

It only affects **how the device treats the packet**:

- which queue to put it in
- which packets get dropped first
- which ones get more bandwidth
- which get lower delay



## DSCP simplified meaning

DSCP is “how important is this packet?”

Here are the **groups**:

### ✓ 1) Default Forwarding (DF)

Value: **0 (000000)**

Normal traffic.

### ✓ 2) Class Selector (CS0–CS7)

Examples:

- CS0 → 000000
- CS1 → 001000
- CS2 → 010000
- CS3 → 011000
- CS4 → 100000
- CS5 → 101000
- CS6 → 110000
- CS7 → 111000

Used for layered priorities.

### ✓ 3) Assured Forwarding (AF)

AF classes look like:

AFxy

x = class (1 to 4)

y = drop precedence (1 to 3)

Example:

- AF11
- AF21
- AF43

Meaning:

- Higher class = higher priority
- Higher "y" = more likely to be dropped during congestion

#### ✓ 4) Expedited Forwarding (EF)

Value: **46 (101110)**

Used for VoIP (low delay, low jitter).

## PART 2 — ECN (2 bits)

### ECN = Explicit Congestion Notification

These two bits help routers **signal congestion without dropping packets**.

Super simple meaning:

ECN tells the receiver: "Network is getting congested; reduce speed."

### ECN bit meanings (VERY IMPORTANT)

There are only **4 possible values**:

Bits	Name	Meaning
00	Not ECN capable	ECN disabled
01	ECN capable (ECT(1))	Allowed to use ECN
10	ECN capable (ECT(0))	Allowed to use ECN
11	Congestion Experienced (CE)	Router marked congestion

This is the ONLY case where a router changes the IP header:

→ It sets the bits to **11 (CE)**.

### How ECN actually works

Let's do the full flow:

#### ✳ Step 1: Sender sends a packet

It sets ECN bits to either:

- **10 (ECT0)**  
or
- **01 (ECT1)**

Meaning: "I support ECN."

#### ✳ Step 2: A router becomes congested

Normally the router would **drop** the packet.

But with ECN:

→ Router sets ECN bits to **11 (CE)**

→ Packet is NOT dropped.

#### ✳ Step 3: Receiver gets packet with CE

Receiver sees ECN=11 → congestion happened.

Receiver sends an **ECN-Echo** back using TCP flag **ECE**.

#### ✳ Step 4: Sender slows down

Sender reduces its sending rate (TCP congestion control).

This avoids packet loss.

## ▢ DSCP + ECN: Combined in simple words

DSCP = “How important / urgent am I?”

ECN = “I saw congestion. Please slow down.”

## ▢ How Linux Kernel handles DSCP/ECN

Inside struct iphdr:

```
__u8 tos; // contains DSCP (upper 6 bits) + ECN (lower 2 bits)
```

Extracting DSCP:

```
u8 dscp = iph->tos >> 2;
```

Extracting ECN:

```
u8 ecn = iph->tos & 0x03;
```

Setting ECN CE (router marks congestion):

```
iph->tos |= 0x03;
```

## ▢ How NIC Hardware uses DSCP + ECN

Many NICs support:

- **DSCP-based hardware queues** → traffic shaping, priority queues
- **ECN marking** → rarely done in NIC, mostly in routers

When sending:

- NIC just transmits DSCP+ECN as part of the IP header.
- If checksum offload is enabled, NIC recalculates IP header checksum.

## ▢ How Routers use DSCP + ECN

DSCP use by routers:

- decide which queue
- decide which packet to drop
- guarantee bandwidth for VoIP
- prioritize video/real-time traffic

ECN use by routers:

- only sets **CE bit**
- does not change DSCP part

## Example: Full Byte Breakdown

Let's say TOS byte = **0xB8** = 10111000

Split:

101110 (DSCP = 46 => EF)

00 (ECN = Not ECN capable)

Meaning:

- DSCP=46 → Expedited Forwarding (VoIP)
- ECN=0 → No congestion notification support

## Final Super-Simple Summary

### DSCP (6 bits)

Label for priority/special handling.

Used for queues, drop priority, low latency, etc.

### ECN (2 bits)

Used to report congestion **without dropping packets**.

- 00 = Not ECN capable
- 01 = ECN capable
- 10 = ECN capable
- 11 = Congestion happened

If you want, I can also give you:

- A Wireshark example showing DSCP + ECN bits
- A hex dump showing how DSCP/ECN look in the IPv4 header
- A diagram of how routers mark ECN=11
- A kernel-flow (skb → iphdr → tos) explanation

Tell me what you want next.

Here is **the exact explanation you asked for — same technical meaning**, but written in **simplified, implied language** so even a beginner can understand.

No skipping, step-by-step, with full clarity.

## IPv4 Initialization & Packet Handler Registration (Simplified but Complete)

When the Linux kernel boots, it needs a way to recognize:

**“Hey, I just received an Ethernet frame. How do I know this is an IPv4 packet, and which function should handle it?”**

The kernel solves this using a **protocol handler table**.

## ★ 1. What is a protocol handler?

A **protocol handler** is simply a function that is called when a packet of a specific protocol arrives.

Example:

Protocol	Ethernet Type	Handler Function
IPv4	0x0800	ip_rcv()
ARP	0x0806	arp_rcv()
IPv6	0x86DD	ipv6_rcv()

So **IPv4 packets → ip\_rcv()**

## ★ 2. How does Linux register the IPv4 handler?

Linux defines this structure:

```
static struct packet_type ip_packet_type = {
    .type = cpu_to_be16(ETH_P_IP), // ETH_P_IP = 0x0800
    .func = ip_rcv,           // handler for IPv4
};
```

This means:

- .type = Ethernet type for IPv4 (0x0800)
- .func = the function to call (ip\_rcv())

Next, during boot, Linux calls:

```
dev_add_pack(&ip_packet_type);
```

This inserts the IPv4 handler into a global list:

ptype\_base[] — a list of all protocol handlers.

So now kernel knows:

“If I see an Ethernet frame with type 0x0800 → call ip\_rcv()”.

## ★ 3. Where does this happen?

Inside the IPv4 initialization function:

```
static int __init inet_init(void)
{
    ...
    dev_add_pack(&ip_packet_type);
    ...
}
```

- `inet_init()` is called during kernel boot.
- It sets up all IPv4-related infrastructure:
- routing subsystem
- /proc/net entries
- ICMP
- UDP, TCP handlers

- fragmentation logic
- neighbour table (ARP)
- packet type registration (dev\_add\_pack())

## Now Imagine the Flow

### Incoming Ethernet Frame → NIC → Kernel → ip\_rcv()

1. NIC receives an Ethernet frame.
2. DMA writes it into an RX ring buffer.
3. Driver builds an sk\_buff.
4. Driver gives the skb to the network stack (usually using netif\_receive\_skb()).
5. Kernel looks at the Ethernet header:
  - Reads Ethernet type (2 bytes).
  - Example: 0x0800 → IPv4.
6. Kernel checks the protocol handler list (registered using dev\_add\_pack()).
7. Finds the entry:  
type = 0x0800 → handler = ip\_rcv()
8. Calls:  
`ip_rcv(skb, dev, pt, orig_dev);`
9. IPv4 stack now handles:
  - checksum validation
  - fragmentation handling
  - routing lookup
  - passing to TCP/UDP/ICMP etc.

## Purpose of this Registration

This registration is the “mapping layer” between:

**Ethernet Frame --> Which kernel code should process it?**

Without this, kernel would not know what to do with raw packets coming from NIC.

## Summary (Very Simplified Version)

- Linux must know which function handles which protocol.
- For IPv4, this is the ip\_rcv() function.
- The mapping “Ethernet Type → Handler” is stored using struct packet\_type.
- During boot, Linux registers the IPv4 handler using dev\_add\_pack().
- When an Ethernet frame arrives, kernel matches the type (0x0800) and dispatches it to ip\_rcv().

## If you want, I can now explain:

- ✓ Full IPv4 Rx Path
- ✓ Full IPv4 Tx Path
- ✓ How ip\_rcv() works internally
- ✓ How packets go from IP → TCP/UDP
- ✓ How ARP ties into this
- ✓ How routing is done (FIB lookup)

Just tell me **which one you want next** or say “**all**” for a full combined Rx+Tx explanation.

Below is a **complete, clear, simplified but technically accurate explanation** of the part you quoted — **what happens inside ip\_rcv\_finish()**, especially around **routing lookup (dst)**, **input callback selection**, and how packets decide their next step.

I will explain **everything implied in the text**, including

- dst\_entry
- skb\_dst()
- routing lookup
- ip\_local\_deliver() vs ip\_forward()
- multicast input paths
- what happens after lookup
- why this lookup is so critical

## ip\_rcv\_finish() — Deep Explanation (Simplified but Full Detail)

```
static int ip_rcv_finish(struct sk_buff *skb)
{
    const struct iphdr *iph = ip_hdr(skb);
    struct rtable *rt;
    ...
}
```

ip\_rcv\_finish() is called **after** basic IPv4 header checks are done.

This is the point where the kernel decides:

“What should I do with this IP packet?”

- Deliver locally?
- Forward to another interface?
- Send to multicast logic?

And **this decision comes from the routing subsystem**.

## ★ 1. What is skb\_dst() ?

Each sk\_buff may have a dst object attached.

```
dst = skb_dst(skb);
```

- dst stands for **destination entry** (struct dst\_entry).
- It represents the **result of a routing decision**.
- It contains:
  - output device
  - gateway
  - input callback
  - output callback
  - MTU information
  - next hop
  - caching flags

Think of dst as “the plan” for this packet:

“Where are you going?

Which device will send you?

Which function will handle you next?”

## ★ 2. What happens if dst is already present?

Then routing lookup was done earlier.

For example:

- packets coming from another layer
- encapsulated packets
- forwarded packets

Kernel just uses the existing routing info.

→ No routing lookup happens again.

→ We simply call the input callback:

dst\_input(skb);

## ★ 3. If dst is NOT present → Perform routing lookup

This is the **normal case** for received packets.

```
rt = ip_route_input_noref(skb, iph->daddr, iph->saddr, iph->tos, dev);
```

Here the kernel looks at:

- **Destination IP**
- **Source IP**
- **TOS field** (this includes DSCP + ECN)
- **Incoming interface**

Routing tables (FIB) are checked.

If lookup fails → packet is dropped.

## ★ 4. Routing Lookup sets the NEXT STEP for the packet

This is very important:

The **routing result** (struct rtable) contains function pointers.

These functions decide the fate of the packet:

✓ rt->dst.input = ip\_forward

Packet is NOT for this machine → must be **forwarded**.

✓ rt->dst.input = ip\_local\_deliver

Packet is **for this machine** → must go to TCP/UDP/ICMP.

✓ rt->dst.input = ip\_mr\_input

Multicast packet handling (special case).

## ★ 5. Why does routing decide the callback?

Because the same function (`ip_rcv_finish`) must support:

- local delivery
- forwarding
- multicast
- special scenarios (tunnels, NAT, rules)

Instead of thousands of if-else checks, Linux uses

**function pointers stored in the dst entry.**

This is extremely efficient.

## ★ 6. After routing is done → kernel calls `dst_input()`

```
return dst_input(skb);
```

Where:

- `dst_input()` looks at:  
`return skb_dst(skb)->input(skb);`

So if routing set:

- `input = ip_local_deliver` → packet goes to local protocol stack (TCP/UDP).
- `input = ip_forward` → kernel will forward it to another NIC.
- `input = ip_mr_input` → multicast logic handles it.

This is the **branching point** of the entire IPv4 stack.

## ★ 7. Packet paths after `ip_rcv_finish`

### A. LOCAL DELIVERY (`input = ip_local_deliver`)

Packet belongs to this machine.

Next steps:

`ip_local_deliver()`

→ `ip_local_deliver_finish()`

→ protocol handler:

TCP? → `tcp_v4_rcv()`

UDP? → `udp_rcv()`

ICMP? → `icmp_rcv()`

### B. FORWARDING (`input = ip_forward`)

Packet is not for this host.

Routing decides output interface.

NAT rules may be applied.

TTL reduced.

Finally output callback is called:

`ip_forward()`

```
→ ip_output()  
→ dev_queue_xmit()  
→ driver TX queue
```

### C. MULTICAST (input = ip\_mr\_input)

Handled by multicast routing code.

## ★ 8. Why routing lookup is done *after* header checks?

Because:

- no point routing corrupted packets
- checksum, length, version must be validated
- routing results depend on TOS and destination address
- netfilter hooks apply before/after routing

## ★ 9. What is included in the routing result (struct rtable)?

It contains:

- next hop gateway
- input callback
- output callback
- outgoing device
- MTU
- flags (DST\_HOST, DST\_NOXFRM, DST\_NOHASH)
- route metrics (RTAX values)
- policy routing decision

The routing subsystem produces a **complete plan** for the packet.

## ★ FINAL SUMMARY (Simplified Explanation)

10. ip\_rcv\_finish() gets called after basic checks.
11. It checks if the skb already has a routing decision (dst).
12. If not → it computes route using ip\_route\_input\_noref().
13. Routing subsystem sets the input function:
  - local packet → ip\_local\_deliver
  - forwarding → ip\_forward
  - multicast → ip\_mr\_input
14. dst\_input() is called.
15. The selected callback continues processing.
16. From here the packet either:
  - goes to TCP/UDP
  - gets forwarded
  - enters multicast logic

This is how Linux decides the “fate” of every IPv4 packet.

If you want, I can also explain:

- ✓ struct rtable fields
- ✓ how FIB lookup works internally
- ✓ how TTL, TOS, routing rules affect input callback
- ✓ how ip\_forward() works internally
- ✓ how ip\_local\_deliver() dispatches to TCP/UDP

Just say: "**Explain next**".

## ★ 7. Everything You Learned Here in One Flow

1. Packet arrives

→ ip\_rcv\_finish() is called.

2. If routing info not attached

→ do routing lookup (ip\_route\_input\_noref).

3. Routing lookup performs:

- source validation (RPF)
  - route lookup
  - determines if local or forward
  - sets dst->input callback
4. If RPF fails → return -EXDEV → packet dropped.
5. If IP header has options → process or drop.
6. If multicast/broadcast → update counters.

7. Call:

dst\_input(skb)

Which jumps to:

- ip\_local\_deliver()
- ip\_forward()
- ip\_mr\_input()

# IPV4 options

Friday, December 5, 2025 9:58 AM

Below is a **clean, simplified, deeply detailed explanation** of **IPv4 Options**, covering **every concept** (Option format, classes, copied flag, examples, kernel handling) — but written in simpler language without removing any technical meaning.

## IPv4 Options — Deep but Simplified Explanation

IPv4 options are an **optional extension area** of the IPv4 header.

Normal IPv4 header = **20 bytes**.

Maximum IPv4 header = **60 bytes** (because ihl field can be max 15 →  $15 \times 4 = 60$  bytes).

So IP options can take up to **40 bytes** ( $60 - 20 = 40$ ).

Why rarely used?

- Slow processing → routers must handle them in the slow path
- Security risks → source routing especially is dangerous
- Incompatible with modern routing policies
- Fragmentation complications

But they exist for specific use cases.

### 1. Why would IP options be useful?

Examples:

Firewalls dropping your packets

→ You could use **Loose Source Route (LSRR)** or **Strict Source Route (SSRR)**

to manually specify the routers the packet must pass through.

Want to see the path packets take?

→ Use **Record Route (RR)** option.

Need to record timestamps of when routers process your packet?

→ Use **Timestamp (TS)** option.

In practice, **almost nobody uses them** except:

- Testing/troubleshooting
- Some security/lab environments
- Old legacy systems

### 2. How options are arranged

The IP header looks like:

[Base Header (20 bytes)] [Options (0–40 bytes)] [Padding]

- Options are **concatenated** back-to-back.

- Header must end on a **4-byte boundary**, so padding (IPOPT\_NOOP) may be inserted.

## ✿ 3. Two Types of IP Options

### A. Single-byte options (1 byte total)

Only 2 exist:

- IPOPT\_END (0)** = End of options list
- IPOPT\_NOOP (1)** = No Operation (padding)

These contain **only the type byte**.

### B. Multibyte options

Format:

```
+-----+-----+-----+
| Option Type | Length | Pointer | Option Data |
+-----+-----+-----+
```

- Option Type**: 1 byte
- Length**: size of entire option (including type, length, etc.)
- Pointer**: index (1-based) inside the option data
- Option Data**: addresses, timestamps, parameters, etc.

## ✿ 4. Option Type Breakdown (The 3 bits) — Very Important

The Option Type byte (8 bits) is divided like this:

```
+---+-----+-----+
| bit | meaning | name      |
+---+-----+-----+
| 7  | copied   | copy flag   |
| 6-5 | class    | option class |
| 4-0 | number   | option number |
+-----+
--+
```

### ✓ Copied Flag

- If **1** → this option is **copied into all fragments**
- If **0** → only copied into the **first fragment**

This matters during fragmentation.

### ✓ Class

There are 4 classes:

Bits	Class Name	Purpose
00	IPOPT_CONTROL	Routing/control

01	IPOPT_RESERVED1	Reserved
10	IPOPT_MEASUREMENT	Debugging/measurement
11	IPOPT_RESERVED2	Reserved

Linux only places **IPOPT\_TIMESTAMP** in MEASUREMENT class.

Everything else is CONTROL class.

### ✓ Number

Gives the option identity (0–31).

## ✿ 5. IP Options Table (Important Options)

Below are the options Linux supports:

Symbol	Num	Class	Copy	Meaning
IPOPT_END	0	0	0	End of options list
IPOPT_NOOP	1	0	0	Padding / alignment
IPOPT_SEC	2	0	1	Security (mostly military)
IPOPT_LSRR	3	0	1	Loose Source Routing
IPOPT_TIMESTAMP	4	2	0 or 1	Timestamp recording
IPOPT_CIPSO	6	0	1	Commercial IP security option
IPOPT_RR	7	0	0	Record Route
IPOPT_SID	8	0	1	Stream Identifier
IPOPT_SSRR	9	0	1	Strict Source Routing
IPOPT_RA	20	0	0	Router Alert

## ✿ 6. Brief Description of Each Option

### ✓ 1. IPOPT\_END (0)

- Single byte: 0x00
- Means “no more options”
- Required when option list doesn’t fill full space

### ✓ 2. IPOPT\_NOOP (1)

- Single byte: 0x01
- Used as **padding** to align to 4-byte boundary
- Can appear anywhere

### ✓ 3. IPOPT\_SEC (Security)

- Rare, military use
- Provides classification/security labels
- Based on RFC 791 and 1108
- Almost never used on the public internet

### ✓ 4. IPOPT\_LSRR — Loose Source Routing

Allows sender to specify a list of routers that **must** be visited.

But routers in between may exist.

Sender → R1 → R2 → R3 → destination

Purpose:

- testing routing
- bypassing firewalls (dangerous)
- troubleshooting

Routers **append their own IP** into the option during forwarding.

#### ✓ 5. IPOPT\_CIPSO

- Used in labeled security environments
- Adds security labels to packets
- Linux supports this for security frameworks

#### ✓ 6. IPOPT\_RR — Record Route

As packets go through routers, each router writes:

- its IP address
- sometimes timestamp (if combined)

Looks like:

Option data:

[Router1 IP] [Router2 IP] [Router3 IP] ...

Maximum length restricts storing many hops (~9 hops max).

#### ✓ 7. IPOPT\_SID — Stream ID

Very old / not used anymore (RFC 791).

#### ✓ 8. IPOPT\_SSRR — Strict Source Routing

Stricter version of LSRR:

**Every hop MUST be exactly the router listed.**

No intermediate routers allowed.

Routers must verify:

- “am I EXACTLY the next hop?”  
If not → drop packet.

#### ✓ 9. IPOPT\_RA — Router Alert

Used by:

- RSVP
- IGMP
- MLD
- MPLS

Tells routers:

“You MUST inspect this packet more deeply. It’s not normal data.”

## ✿ 7. Kernel Handling of Options

Linux processes options in the function:

`ip_rcv_options(skb)`

This:

- parses each option
- validates lengths
- validates pointer fields
- supports RR, TS, RA, SSRR, LSRR
- rejects malformed options
- handles fragment-copy behaviour

If invalid:

- returns non-zero
- packet dropped

Also involved:

- ip\_options\_compile
- ip\_options\_fragment
- ip\_options\_rcv\_srr

## ✿ 8. Fragmentation & Copied Flag

When an IP packet is fragmented:

- some options must be copied to **every fragment** (copy=1)
- some options only appear in the **first fragment** (copy=0)

For example:

- Source routing → must be copied
- Record route → usually only first fragment

Linux code ensures this via ip\_options\_fragment().

## ✿ 9. Why Modern Networks Avoid IP Options

- Routers must switch to **slow path**, drastically slower
- Easy to abuse for attacks
- Complicates fragmentation
- Breaks with NAT/firewalls
- Not supported by many devices

Almost all packets today have:

IHL = 5 → no options

## ✿ If you want, I can now explain:

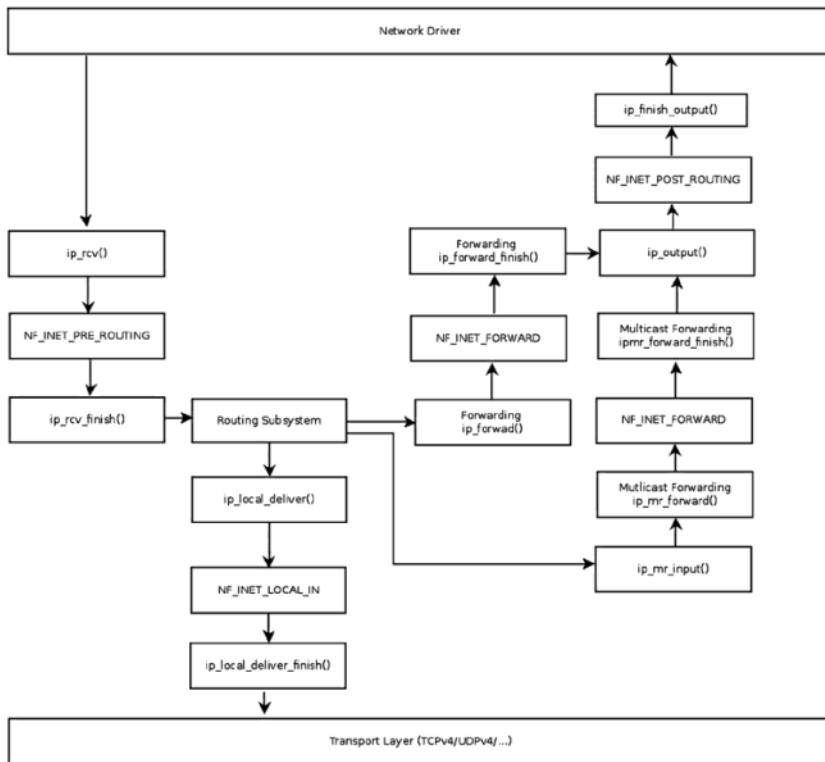
- ✓ **Timestamp option in full detail** (modes 0, 1, 3 + pointer logic)
- ✓ **Record Route option fully detailed** (including RR pointer handling)
- ✓ **How Linux parses options internally** (ip\_rcv\_options, ip\_options\_compile)
- ✓ **Source routing detailed (LSRR vs SSRR)**
- ✓ **Router Alert usage in IGMP, RSVP, etc.**

Just tell me “Explain Timestamp Option” or “Explain Record Route” or “Explain all options deeply”.



## Receiving Path

Monday, December 8, 2025 9:32 AM



### IPv4 RX / Forwarding TX path — step-by-step, what each box does and which layer it's at

I'll walk the diagram left→right and top→bottom, explaining each function/box in simple, accurate terms, what it *does internally*, which *layer* it belongs to (L2=link, L3=IP, L4=transport, or "netfilter" for firewall hooks), and the important *special cases* that can change the path.

#### Network Driver (NIC)

**Layer:** L2 (link) / hardware boundary

**What it does:** receives raw frames from the wire, validates link-level (FCS), maps DMA buffers to an skb and calls the kernel receive entry (usually `netif_rx()` or napi callback). The skb at this point contains link header + payload; network driver sets `skb->dev` and `skb->protocol` (ethertype).

**Special cases:** VLAN tags, checksum offload (driver may mark `ip_summed`), GRO may aggregate multiple packets into one skb.

#### ip\_rcv()

**Layer:** L3 (IPv4 input)

**What it does:** main IPv4 input entry for packets with ethertype IPv4 (called from `eth_type_trans()`/netstack). Tasks include:

- Validate IPv4 header (version, header length)

- Validate checksum (unless offloaded)
  - Handle header options basic parsing (or mark to call option processing later)
  - Basic sanity tests (min length, TTL, etc.)
  - Potentially decide whether to route, deliver locally, or drop.
- Special cases:** malformed packets → drop/ICMP; fragmented packets are detected here (`ip_is_fragment()` may be checked later).

## NF\_INET\_PRE\_ROUTING (netfilter PRE\_ROUTING hook)

**Layer:** netfilter (firewall) sitting at L3 boundary

**What it does:** gives firewall/iptables/nftables modules the first chance to see/modify/drop the packet before routing decision. Modules here can: change destination (DNAT), drop, mark, or accept.

**Special cases:** if DNAT changes destination address, routing lookup must be redone; conntrack may create a conntrack entry here.

## ip\_rcv\_finish()

**Layer:** L3 (IPv4 finishing input)

**What it does:** finishes input processing started in `ip_rcv()` and prepares for routing decision.

Typical tasks:

- Perform route lookup (FIB) or prepare to call routing subsystem
- Handle link-local / broadcast addresses
- Update stats
- Might call `ip_defrag()` here if fragment handling is required or defer to `ip_local_deliver()` depending on code path.

**Special cases:** if packet is for local machine, proceed to `ip_local_deliver()`. If not, and IP forwarding is enabled, proceed to forwarding path.

## Routing Subsystem / FIB lookup

**Layer:** L3 (routing decision)

**What it does:** decides whether packet is destined for this host or should be forwarded:

- Uses dst/route tables (FIB) to find `dst_entry` (output device, next hop, MTU, route attrs)
  - If destination is local (matches one of local addresses) → deliver locally
  - Otherwise → forward (if `net.ipv4.ip_forward` enabled) or drop + send ICMP unreachable
- Special cases:** policy routing, multipath routes, routes with special `dst_output` (e.g., encapsulation), and route changes.

## Local delivery path (ip\_local\_deliver())

**Layer:** L3 → L4 transition

**What it does:** deliver to local sockets:

- If packet is fragmented → call `ip_defrag()` to reassemble (only after successful checks).

- After defrag (or if not fragmented), call NF\_INET\_LOCAL\_IN hook (netfilter INPUT/LOCAL\_IN).
  - Run last steps ip\_local\_deliver\_finish() which will demultiplex to transport: deliver to raw sockets, ICMP, TCP, UDP, etc.
- Special cases:** defragmentation (call ip\_defrag()), local multicast handling, kernel sockets (raw/NETLINK), conntrack state tracking, and socket filters (BPF) on receive.

### NF\_INET\_LOCAL\_IN (netfilter LOCAL\_IN / INPUT chain)

**Layer:** netfilter / L3→L4 boundary

**What it does:** firewall/iptables/nftables INPUT hooks for local processing: can accept, drop, log, or redirect. NAT (DNAT) normally happens earlier (PRE\_ROUTING) for incoming packets.

**Special cases:** socket redirection, local netfilter modules (e.g., intrusion detection) may require the full packet (thus defrag).

### ip\_local\_deliver\_finish()

**Layer:** L3→L4 finishing work

**What it does:** final checks and demux to L4 handlers: set skb->transport\_header, verify transport checksum (if needed), invoke transport handlers (ip\_local\_deliver\_finish calls NF\_HOOK(...LOCAL\_IN, ..., handler) — then socket layer receives packet).

**Special cases:** ICMP responses may be generated here, delivered to raw sockets, or handed to TCP/UDP (which may generate ACKs or responses).

### Transport Layer (TCP/UDP/ICMP)

**Layer:** L4

**What it does:** handle the payload: TCP handles stream reassembly, congestion control, sockets; UDP delivers datagram to socket(s); ICMP handled by kernel. Transport layer is where applications see full, reassembled packets.

**Special cases:** Socket filters (SO\_ATTACH\_FILTER), TPROXY, socket buffer ownership (for accounting).

### Forwarding path (when packet is **not** for local host)

If routing subsystem decides the packet should be forwarded:

### ip\_forward() (forwarding)

**Layer:** L3 (IPv4 forwarding)

**What it does:** prepare to forward:

- Decrement TTL; if TTL reaches 0 → send ICMP Time Exceeded and drop.
- Optionally clear some options or process if present.
- Possibly consult conntrack and netfilter FORWARD chain.
- Determine outgoing route/dst (or reuse previously found dst).

**Special cases:** TTL/ICMP handling, fragmentation requirements for outgoing MTU, and policy routing.

### [NF\\_INET\\_FORWARD](#) (netfilter FORWARD chain)

**Layer:** netfilter (L3)

**What it does:** firewall chain for forwarded packets: can accept, drop, nat/masquerade (SNAT occurs in POSTROUTING normally), log, or modify. Conntrack typically hooks here to track connections.

**Special cases:** If dropped here, forwarding stops; conntrack may cause stateful decisions.

### [Forwarding finish ip\\_forward\\_finish\(\) or ip\\_forward\(\)](#) continuation

**Layer:** L3

**What it does:** finalize forwarding decisions and hand packet into output processing path — typically calls ip\_output() (or specialized ipmr\_forward\_finish() for multicast forwarding) to prepare L2 headers and transmit.

**Special cases:** multicast forwarding uses specialized logic (see below).

### [ip\\_output\(\)](#)

**Layer:** L3 (egress processing)

**What it does:** general output path for IPv4: handles fragmentation (if packet > MTU), sets proper header fields, and eventually invokes NF\_INET\_POST\_ROUTING. If fragmentation needed, ip\_fragment() is called.

**Special cases:** If DF set and packet too big → send ICMP Fragmentation Needed (used by PMTUD); fragmentation may be done here (slow or fast path).

### [NF\\_INET\\_POST\\_ROUTING](#) (netfilter POSTROUTING chain)

**Layer:** netfilter (egress)

**What it does:** final chance to mangle packets before leaving the host/network node: SNAT/MASQUERADE are typically applied here, marking, QoS tagging, logging. After this hook packet is ready for link layer.

**Special cases:** NAT changes source ip/port so routing changes might be required but typically SNAT is final step for forwarded/outgoing packets.

### [ip\\_finish\\_output\(\)](#)

**Layer:** L3→L2 handoff

**What it does:** prepare packet for link-layer: build L2 header (Ethernet), consult neighbour/ARP to find next-hop MAC address and set it, handle encapsulation if needed, and call dev\_queue\_xmit() to hand to qdisc / NIC. If next-hop MAC unknown, queue skb on neighbour resolution queue.

**Special cases:** ARP requests and neighbour proxy, encapsulated transports (tunnels), link MTU awareness, and errors returned here.

## [dev\\_queue\\_xmit\(\)](#)

**Layer:** L2 (local device transmit)

**What it does:** pass skb to traffic scheduler (qdisc), apply queuing disciplines, dequeue to appropriate hardware queue, and call network device ndo\_start\_xmit() which the driver implements to create DMA descriptors and notify NIC.

**Special cases:** qdisc shaping (fq\_codel, HTB), multiqueue devices, hardware offload (TSO, GRO/SG), XDP integration may redirect/drop.

## [Network Driver \(transmit side\)](#)

**Layer:** L2 / hardware

**What it does:** driver programs the NIC (TX descriptors), maps skb fragments for DMA (scatter-gather), sends to wire. NIC adds link header/trailer as needed.

**Special cases:** if NIC busy or ring full, driver returns NETDEV\_TX\_BUSY, skb may be requeued; driver may do checksum offload or large segment offload.

## **MULTICAST PATH EXPLANATION**

Multicast has special functions because:

- Packet might be delivered locally **AND** forwarded to multiple interfaces
- Uses multicast routing protocols (IGMP, PIM)

Let's explain multicast boxes:

## **16. ip\_mr\_input()**

**Layer:** L3 (Multicast Receive Entry)

Triggered when destination IP is **224.0.0.0/4 (multicast range)**.

**Purpose:**

- Check IGMP membership
- Check kernel multicast routing tables
- Decide local + forwarding behavior

## **17. ip\_mr\_forward()**

**Layer:** L3 Multicast Forwarding Engine

What it does:

- Look up multicast routing rules (MFC)
- Sends packet to each outgoing interface in multicast tree
- Handles TTL decrement
- Drops if TTL = 0

- Might also deliver locally

### 18. NF\_INET\_FORWARD (multicast path)

Same as unicast forward hook, but packet is multicasted.

Allows firewall rules like:

- Allow multicast on specific interface
- Drop multicast from unknown source

### 19. ipmr\_forward\_finish()

**Layer:** L3 → L2

Purpose:

- Final operations before handing the multicast copy to NIC
- Add Ethernet multicast MAC
- Send multiple copies if necessary

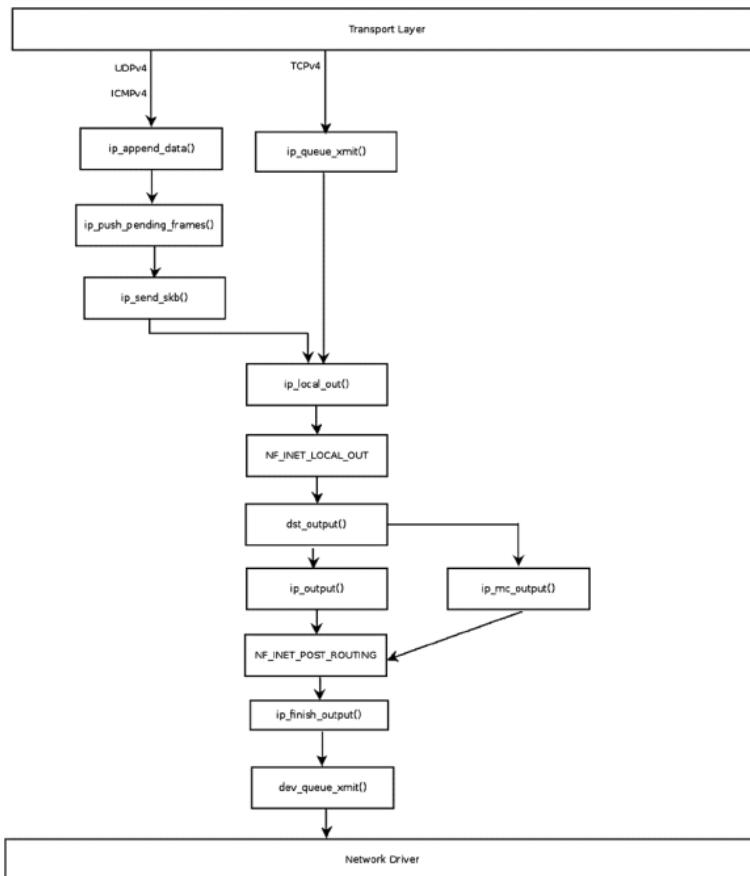
### In short — multicast logic

Function	Meaning
<code>ip_mr_input()</code>	Detects multicast packet
<code>ip_mr_forward()</code>	Decides which interfaces must receive copies
<code>NF_INET_FORWARD</code>	Firewall rules applied
<code>ipmr_forward_finish()</code>	Actually sends one copy to each interface

# Transfer path

Friday, December 5, 2025 11:03 AM

CHAPTER 4 ■ IPv4



Got it — you want a **full explanation of each function in the IPv4 TX path diagram**, including:

- Which *layer* it belongs to
- What the function *does internally*
- How it *processes packet data (skb)*
- How it handles *special cases* (multicast, routing, fragmentation, netfilter, etc.)

I will now explain the entire flow **step-by-step like a teacher**, in easy terms but with kernel-level accuracy.

## BIG-PICTURE: The IPv4 Transmit (TX) Path

A packet travels through **multiple layers inside the Linux kernel** before it reaches the NIC (driver):

- Transport Layer (L4)
  - IPv4 Layer (L3)
  - Routing (FIB)
  - Netfilter Output Hooks
  - L3 Final Output
  - L2 (ARP, link-layer encapsulation)
  - NIC Driver

The diagram represents this journey.

Now let's explain every function.

## ◇ 1. ip\_append\_data() — Transport Layer (UDP/ICMP)

**Layer:** L4 (UDP, ICMP)

Used by:

- UDPv4
- ICMPv4
- Raw sockets

Purpose:

Builds an SKB (or SKB list) containing the L4 payload.

Internal behavior:

- Allocates SKB(s)
- Appends data to skb->data area
- Reserves space for IPv4 header (later added)
- Handles fragmentation *at UDP level* if UDP segmentation offload is disabled
- Stores partial frames in socket write queue

Special cases:

- If packet > MTU → fragmentation handled later by ip\_push\_pending\_frames()
- For ICMP errors, SKB may include part of original packet

## ◇ 2. ip\_push\_pending\_frames() — Transport Layer (UDP/ICMP)

**Layer:** L4 → L3 transition

Purpose:

Take the SKBs built by ip\_append\_data() and push them down to IPv4.

Internal behavior:

- Finalizes SKB content
- Adds IPv4 header
- Sets fields: protocol, TTL, TOS, DF flag, ID, checksum
- Calls ip\_send\_skb()

## ◇ 3. ip\_send\_skb() — L3 Entry Point for UDP/ICMP

**Layer:** L3 (IPv4 core)

Purpose:

Send a fully built skb to the IPv4 output path.

Internals:

Does nothing fancy — just calls ip\_local\_out().

## ◇ 4. ip\_queue\_xmit() — Transport Layer (TCP)

## **Layer: L4 (TCP)**

Used by TCP to send segments.

Purpose:

Builds an SKB for TCP:

- TCP header
- Options
- Payload
- Segmentation logic (TSO/GSO)

Then attaches IPv4 header template (iphdr), sets flow info.

Internal behavior:

- Does MSS check
- May split packet if > MSS
- Respects Path MTU Discovery (DF bit)
- Then calls ip\_local\_out()

Special cases:

- If PMTUD lowers MTU → TCP adjusts MSS
- TCP never fragments itself, relies on IPv4 fragmentation if DF=0

## **◇ 5. ip\_local\_out() — L3 IPv4 (Main Entry Point)**

### **Layer: IPv4 Core (L3)**

Purpose:

Initial point for all **locally generated** packets.

Internal behavior:

- Ensures checksum is done
- Performs initial validations
- Passes skb to Netfilter hook: **NF\_INET\_LOCAL\_OUT**

## **◇ 6. Netfilter — NF\_INET\_LOCAL\_OUT**

### **Layer: Firewall layer (between L3 processing stages)**

Hooks:

iptables/nftables OUTPUT chain

Purpose:

Allow firewall to:

- Drop packets
- Modify (mangle) packets
- NAT (SNAT, MASQUERADE)

Special cases:

- If NAT modifies destination → routing is recalculated

## **◇ 7. dst\_output() — Routing subsystem (FIB)**

### **Layer: Routing (L3 decision)**

Purpose:

Choose correct output function depending on route:

- unicast → ip\_output()
- multicast → ip\_mc\_output()
- broadcast → ip\_output() with special handling
- loopback → ip\_loopback\_xmit()
- FIB rules may call ip\_forward() if packet is forwarded (not local)

Internal behavior:

Routing lookup was done earlier by TCP or local delivery.

This step **chooses the correct output function from dst\_ops**.

## ◇ 8. ip\_output() — Main IPv4 Output Function

Layer: L3 (IPv4 output processing)

Purpose:

Prepare packet for L2 processing and handle fragmentation.

Internal behavior:

1. Checks if packet > MTU  
→ Calls ip\_fragment()
2. Adds L2 header later (not here)
3. Calls Netfilter POST\_ROUTING hook
4. Finally calls ip\_finish\_output()

Special cases:

- If DF=1 and packet > MTU → send ICMP Frag Needed (PMTUD)
- If multicast → forwarded to ip\_mc\_output()

## ◇ 9. ip\_mc\_output() — Multicast Output Path

Layer: L3 (IPv4 multicast)

Purpose:

Special handling for:

- 224.x.x.x addresses
- IGMP membership
- TTL scoping (multicast TTL)
- Loopback delivery for multicast sockets

Internal behavior:

- Apply outbound multicast rules
- Possibly send multiple copies depending on interfaces
- Then call ip\_finish\_output()

## ◇ 10. Netfilter — NF\_INET\_POST\_ROUTING

Layer: Firewall (after all routing + L3 processing)

Hooks:

POSTROUTING chain

Used for:

- SNAT
- MASQUERADE

- Traffic shaping
  - Packet marking
- After this, packet is ready for final transmission.

## ◇ 11. ip\_finish\_output() — L3 → L2 Transition

**Layer:** L3 to L2 handoff

Purpose:

Prepare packet for link layer.

**Internal behavior:**

- Build L2 (Ethernet) header using neigh\_output()
- ARP handling:
  - If ARP entry exists → add MAC and send
  - If ARP not available → queue skb in neighbour queue

Then calls dev\_queue\_xmit().

## ◇ 12. dev\_queue\_xmit() — L2 Queueing & Scheduling

**Layer:** L2 (MAC layer)

Purpose:

Send skb to the correct network device driver queue.

**Internal behavior:**

- Traffic scheduler (qdisc – HTB, fq\_codel, etc.)
- TX queue selection
- Multiqueue NIC mapping
- Apply shaping / rate limiting
- Hand packet to NIC driver via ndo\_start\_xmit()

## ◇ 13. NIC Driver (Network Driver TX Path)

**Layer:** L2 Hardware Interface

Purpose:

Convert SKB into DMA descriptors and push to NIC.

**Internal behavior:**

- Maps skb fragments (scatter-gather)
- Sets DMA descriptors
- Notifies NIC hardware to transmit
- Hardware performs link-layer framing (FCS etc.)

## ⚡ COMPLETE PACKET FLOW SUMMARY (Simplified)

TCP/UDP → skb created

- ip\_local\_out()
- Netfilter LOCAL\_OUT
- dst\_output() (routing)
- ip\_output() or ip\_mc\_output()
- Fragmentation (ip\_fragment) if needed

→ Netfilter POST\_ROUTING  
→ ip\_finish\_output()  
→ dev\_queue\_xmit() (qdisc)  
→ NIC driver  
→ Wire

If you want, I can also explain:

- ✓ A fully detailed flow with ASCII diagrams
- ✓ How ip\_fragment() fits into this pipeline
- ✓ RX path equivalent of this flow
- ✓ How GSO/TSO modifies this pipeline
- ✓ How qdisc, ARP, routing tables interact here
- ✓ Add call stack for each function

Just tell me what to expand!

# fragmentation

Friday, December 5, 2025 11:03 AM

Below is a **BEGINNER-LEVEL**, step-by-step, teacher-style explanation of the paragraph you provided.

I will explain **every concept**, how Linux kernel handles it, the functions, the fast/slow path, why fragmentation happens, and give a **real-life example with diagrams**.

## IPv4 Fragmentation — Explained Like a Teacher

### 1. Why Fragmentation Exists?

Every network interface (NIC) has a **maximum frame size** it can transmit.

This is called **MTU (Maximum Transmission Unit)**.

- On normal Ethernet → **MTU = 1500 bytes**
- On jumbo-frame capable NICs → **MTU ≈ 9000 bytes**

If a packet coming from upper layers (IP → then TCP/UDP) is **bigger than MTU**, it **cannot** be sent as one piece.

So IP must **cut it into smaller pieces** → this is **fragmentation**.

### 2. Where Fragmentation Happens in Linux?

The code mentioned lives here:

Purpose	File
Fragmentation	net/ipv4/ip_output.c
Defragmentation	net/ipv4/ip_fragment.c

Two key functions:

**1 ip\_fragment()**

Splits a large packet into fragments.

**2 ip\_defrag()**

Reassembles incoming fragments back into a full packet (done at receiver).

### 3. Prototype:

```
int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff *))
```

Meaning:

- **skb** → the packet buffer to fragment
- **output()** → function that actually sends each fragment

When fragmentation is triggered by the normal transmit path (`ip_finish_output()`), it passes:

```
output = ip_finish_output2()
```

So:

`ip_fragment` → produces fragments → `ip_finish_output2` sends them out

## 4. FAST path vs SLOW path

✓ Fast Path (rare)

If `skb->frag_list` is **not NULL**, meaning:

- Upper layers (TCP, drivers) **already prepared fragments**, so IP layer does not need to create new ones.

Example: Local GSO (Generic Segmentation Offload).

✓ Slow Path (common)

When fragments must be created manually by the IP layer.

Your paragraph covers the **slow path**, which beginners must understand.

## 5. Checking if Fragmentation is Allowed

Before fragmenting, Linux checks if fragmentation is permitted:

`if (iph->frag_off & IP_DF)`

**IP\_DF = Don't Fragment flag**

If DF = 1 → sender said **do not fragment**

Then we **must not break the packet**.

What happens then?

✗ If fragmentation is NOT allowed:

The kernel:

1. Drops the packet
2. Sends an ICMP error back to sender:  
ICMP Destination Unreachable  
Code = Fragmentation Needed

This is how **Path MTU Discovery (PMTUD)** works.

3. Updates statistics: `IPSTATS_MIB_FRAGFAILS`
4. Returns error -`EMSGSIZE`

## Example of DF flag issue

Imagine:

Laptop → WiFi router → ISP link with MTU=1400

Your laptop sends a **1500-byte** packet with **DF=1**

Router sees MTU is 1400, tries to forward, cannot fragment.

So router replies:

ICMP Frag Needed (MTU=1400)

Laptop receives this and reduces its MTU for that route.

## 6. Real LIFE Example of Fragmentation

Imagine sending a **4000-byte UDP packet** through Ethernet.

Ethernet MTU = 1500

IP header = 20 bytes

Payload left per fragment:

Fragment size =  $1500 - 20 = 1480$  bytes

4000 bytes must be split:

Fragment #	Payload	Offset
1	1480 bytes	0
2	1480 bytes	$1480/8 = 185$
3	1040 bytes	$2960/8 = 370$

ASCII Diagram:

[4000 byte UDP data]



| Fragment 1 | Fragment 2 | Fragment 3 |  
| 1480 bytes | 1480 bytes | 1040 bytes |

At receiver:

ip\_defrag() reassembles these by using fragment offsets.

## 7. Breaking Down Kernel Code

Your snippet:

Below is a **BEGINNER-LEVEL**, step-by-step, teacher-style explanation of the paragraph you provided.

I will explain **every concept**, how Linux kernel handles it, the functions, the fast/slow path, why fragmentation happens, and give a **real-life example with diagrams**.

## IPv4 Fragmentation — Explained Like a Teacher

### 1. Why Fragmentation Exists?

Every network interface (NIC) has a **maximum frame size** it can transmit.

This is called **MTU (Maximum Transmission Unit)**.

- On normal Ethernet → **MTU = 1500 bytes**
- On jumbo-frame capable NICs → **MTU ≈ 9000 bytes**

If a packet coming from upper layers (IP → then TCP/UDP) is **bigger than MTU**, it **cannot** be sent as one piece.

So IP must **cut it into smaller pieces** → this is **fragmentation**.

## 2. Where Fragmentation Happens in Linux?

The code mentioned lives here:

Purpose	File
Fragmentation	net/ipv4/ip_output.c
Defragmentation	net/ipv4/ip_fragment.c

Two key functions:

**[1] ip\_fragment()**

Splits a large packet into fragments.

**[2] ip\_defrag()**

Reassembles incoming fragments back into a full packet (done at receiver).

## 3. Prototype:

```
int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff *))
```

Meaning:

- skb → the packet buffer to fragment
- output() → function that actually sends each fragment

When fragmentation is triggered by the normal transmit path (ip\_finish\_output()), it passes:

```
output = ip_finish_output2()
```

So:

**ip\_fragment → produces fragments → ip\_finish\_output2 sends them out**

## 4. FAST path vs SLOW path

✓ Fast Path (rare)

If skb->frag\_list is **not NULL**, meaning:

- Upper layers (TCP, drivers) **already prepared fragments**, so IP layer does not need to create new ones.

Example: Local GSO (Generic Segmentation Offload).

✓ Slow Path (common)

When fragments must be created manually by the IP layer.

Your paragraph covers the **slow path**, which beginners must understand.

## 5. Checking if Fragmentation is Allowed

Before fragmenting, Linux checks if fragmentation is permitted:

```
if (iph->frag_off & IP_DF)
```

**IP\_DF = Don't Fragment flag**

If DF = 1 → sender said **do not fragment**

Then we **must not break the packet**.

What happens then?

 If fragmentation is NOT allowed:

The kernel:

5. Drops the packet
6. Sends an ICMP error back to sender:  
ICMP Destination Unreachable

Code = Fragmentation Needed

This is how **Path MTU Discovery (PMTUD)** works.

7. Updates statistics: IPSTATS\_MIB\_FRAGFAILS
8. Returns error -EMSGSIZE

## Example of DF flag issue

Imagine:

Laptop → WiFi router → ISP link with MTU=1400

Your laptop sends a **1500-byte** packet with **DF=1**

Router sees MTU is 1400, tries to forward, cannot fragment.

So router replies:

ICMP Frag Needed (MTU=1400)

Laptop receives this and reduces its MTU for that route.

## 6. Real LIFE Example of Fragmentation

Imagine sending a **4000-byte UDP packet** through Ethernet.

Ethernet MTU = 1500

IP header = 20 bytes

Payload left per fragment:

Fragment size =  $1500 - 20 = 1480$  bytes

4000 bytes must be split:

Fragment #	Payload	Offset
1	1480 bytes	0
2	1480 bytes	$1480/8 = 185$
3	1040 bytes	$2960/8 = 370$

ASCII Diagram:

[4000 byte UDP data]

|

↓

-----  
| Fragment 1 | Fragment 2 | Fragment 3 |  
| 1480 bytes | 1480 bytes | 1040 bytes |  
-----

At receiver:

ip\_defrag() reassembles these by using fragment offsets.

## 7. Breaking Down Kernel Code

Your snippet:

```
iph = ip_hdr(skb);
if (((iph->frag_off & IP_DF) && !skb->local_df) ||
    (IPCB(skb)->frag_max_size &&
     IPCB(skb)->frag_max_size > dst_mtu(&rt->dst))) {
    IP_INC_STATS(dev_net(dev), IPSTATS_MIB_FRAGFAILS);
    icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
              htonl(ip_skb_dst_mtu(skb)));
    kfree_skb(skb);
    return -EMSGSIZE;
}
```

✓ iph->frag\_off & IP\_DF

Checks DF flag.

✓ skb->local\_df

Kernel itself set DF internally? Allow override.

✓ IPCB(skb)->frag\_max\_size

If upper layer (like tunnel) limits the fragment size, and packet is too big → fail.

✓ ip\_skb\_dst\_mtu(skb)

Find the MTU toward that destination.

## 8. What Happens After This Check?

If fragmentation is allowed:

- Kernel computes how many bytes available per fragment
- Creates new SKBs per fragment
- Copies IP header + payload slice
- Sets:
  - MF flag (More fragments)
  - Fragment offset
- Calls:
  - output(fragment\_skb); for each piece

## 9. Summary (Teacher Version)

Concept	Meaning
MTU	Max bytes interface can send
Fragmentation	Split big packets into smaller ones
DF flag	Disallow fragmentation
ICMP Frag Needed	Sent when DF=1 but packet is too large
Fast path	Already fragmented at higher layer
Slow path	IP manually splits packet
ip_fragment()	Splits into SKBs
ip_defrag()	Reassembles at receiver
Offset	Where that fragment belongs in original packet

## If you want...

I can explain next:

✓ How ip\_defrag() reassembles fragments (with diagrams)

- ✓ How router/NIC handles fragmentation internally
- ✓ Show C code flow of ip\_fragment() line-by-line
- ✓ Explain MF flag, offset, identification field
- ✓ Packet capture (tcpdump) showing real fragments

Just tell me what to explain next!

Checks DF flag.

✓ skb->local\_df

Kernel itself set DF internally? Allow override.

`IPCB(skb)->frag_max_size`

If upper layer (like tunnel) limits the fragment size, and packet is too big → fail.

`ip_skb_dst_mtu(skb)`

Find the MTU toward that destination.

## 8. What Happens After This Check?

If fragmentation is allowed:

- Kernel computes how many bytes available per fragment
- Creates new SKBs per fragment
- Copies IP header + payload slice
- Sets:
  - MF flag (More fragments)
  - Fragment offset
- Calls:
  - `output(fragment_skb);` for each piece

## 9. Summary (Teacher Version)

Concept	Meaning
MTU	Max bytes interface can send
Fragmentation	Split big packets into smaller ones
DF flag	Disallow fragmentation
ICMP Frag Needed	Sent when DF=1 but packet is too large
Fast path	Already fragmented at higher layer
Slow path	IP manually splits packet
<code>ip_fragment()</code>	Splits into SKBs
<code>ip_defrag()</code>	Reassembles at receiver
Offset	Where that fragment belongs in original packet

### If you want...

I can explain next:

- How `ip_defrag()` reassembles fragments (with diagrams)**
- How router/NIC handles fragmentation internally**
- Show C code flow of `ip_fragment()` line-by-line**
- Explain MF flag, offset, identification field**
- Packet capture (tcpdump) showing real fragments**

Just tell me what to explain next!

Sure — here is the **purely theoretical, concept-level explanation** of what is happening in the **fast path of IPv4 fragmentation**, without code, only the logic.

## ✿ Theoretical Explanation of Fast Path in IPv4 Fragmentation

When the transport layer (TCP/UDP) gives IP a packet that is **too large** for the outgoing interface (MTU), IP must **break it into fragments**.

Linux has **two mechanisms** to fragment:

1. **Slow Path — general-purpose, safe, works for all packets**
2. **Fast Path — highly optimized for a specific case**

Your snippet explains the **fast path**.

### ✓ WHEN does the fast path run?

Fast path is used when:

- The SKB contains a **frag\_list**, meaning:
- The packet is already internally split into multiple fragments by higher layers (like TSO/GSO).
- No complicated IP options exist.
- No checksums need to be recalculated in complex ways.

In this situation, Linux avoids copying data and avoids iterating through data pages — it simply reuses SKBs already prepared.

So the fast path is chosen to **avoid expensive work**.

### ⌚ WHAT does the fast path do (theory only)?

The job of the fast path is:

#### 1. Take the main SKB (first fragment)

This SKB contains the original transport header and part of the payload.

#### 2. Take the frag\_list (linked list of SKBs)

Each SKB in frag\_list contains the *next pieces of the payload*, already sized to MTU.

#### 3. For each SKB (including the first):

The fast path builds a correct IPv4 header and sends the fragment.

This process happens **without copying data** → only headers are manipulated.

## ☛ THEORETICAL STEPS INSIDE FAST PATH

Below is the **high-level logic**, ignoring all low-level details.

### STEP 1 — Check if the packet qualifies

If SKB has a `frag_list`, then IP realizes:

“This data is already broken into pieces, I don’t need to do heavy fragmentation work.”

So fast path is allowed.

## STEP 2 — Build the header for the first fragment

- Set the IPv4 header length
- Set **total length** of the first fragment
- Set **IP\_MF** flag (“More Fragments coming”)
- Clear/prepare metadata
- Recalculate checksum

This creates fragment #1.

## STEP 3 — Prepare to iterate through the `frag_list`

`frag_list` is a linked list:

main SKB → frag1 → frag2 → frag3 → ...

We process them one by one.

## STEP 4 — For each fragment in `frag_list`:

### 4.1. Move data pointer back

So we can insert a fresh IPv4 header at the start.

### 4.2. Copy the IPv4 header from the first fragment

This preserves all fields (source IP, dest IP, protocol, ID, etc.)

### 4.3. Fix the header for this fragment

- Set **tot\_len = size of this fragment**
- Set **frag offset** based on how much data has already been sent  
(offset measured in 8-byte units)
- If this is **not** the last fragment → set **IP\_MF**

### 4.4. Recalculate checksum

Because header fields changed.

## STEP 5 — Send the fragment

The output function transmits this fragment to L2 driver.

If success → statistics updated.

If fail → free all remaining fragments.

## ⌚ FINAL RESULT

Fast path fragmentation:

- ✓ doesn't copy payload data
- ✓ reuses SKBs already built
- ✓ only adjusts headers
- ✓ walks through frag\_list and sends each SKB
- ✓ extremely efficient for GSO/TSO output

## Why is this “FAST”?

Because it avoids:

- Splitting data manually
- Copying data into new SKBs
- Recalculating partial checksums
- Rebuilding transport headers

The fast path is just:

“For each SKB fragment: copy header → adjust offset → send”

That's all.

## If you want, I can also give you:

- ✓ slow path theoretical explanation
- ✓ difference between fast path and slow path
- ✓ ASCII diagrams of what SKBs look like
- ✓ a simple analogy (like cutting a cake)

Just tell me!

Below is the **purely theoretical, easy-for-beginner, no code, full explanation** of what happens in the **slow path of IPv4 fragmentation**.

## What Is Slow Path Fragmentation? (Theory)

Slow path is used when:

- The packet **does NOT** have a ready-made frag\_list
- Or the packet is too complex for fast path (options, non-linear SKB, etc.)
- Or IP needs to **manually split** one big buffer into many smaller MTU-sized fragments

In slow path, **Linux must do all the fragmentation work itself**:

- ✓ cut the data into MTU chunks
- ✓ allocate new SKBs for each chunk
- ✓ copy headers
- ✓ copy payload
- ✓ fill IP header fields
- ✓ compute checksum
- ✓ send fragment
- ✓ repeat until data is finished

This is why it is called *slow*.

## ✿ THEORETICAL STEPS IN THE SLOW PATH

Let's go through each conceptual step, matching your description but in very simple words.

### 1 Calculate how much data must be fragmented

- Extract IPv4 header
- Calculate total payload (left)
- Remove the IPv4 header length (hlen)  
→  $\text{left} = \text{total\_length} - \text{header\_length}$

We now know how much payload must be split.

### 2 Loop until ALL data is fragmented

This loop:

```
while (there is data left)
    create a fragment
    send it
```

Each loop iteration creates **ONE fragment**.

### 3 Determine the size of this fragment

- If  $\text{left} > \text{mtu}$ , then this fragment cannot hold all remaining data  
→ so let it take only mtu bytes

For all fragments except the last:

⌚ **fragment must end on an 8-byte boundary**

So Linux does:

$\text{len} = \text{len} \& \sim 7$

This clears lower 3 bits → rounds down to nearest 8-byte multiple.

This is an IPv4 requirement.

### 4 Allocate a new SKB for this fragment

Because slow path must create **a brand new SKB**:

- Allocate SKB
- Reserve link-layer space
- Add **header + payload** space

Fast path CANNOT be used because data is not pre-split.

## 5 Copy necessary metadata

Copy fields like:

- protocol
- priority
- packet type
- network namespace settings

These must be inherited from the original packet.

## 6 Copy the IPv4 header

You must copy the **header from the original SKB** into this new SKB.

## 7 Copy payload for this fragment

Copy len bytes from the original SKB into this new SKB.

Now this SKB contains:

[IPv4 header][part of original data]

## 8 Fill in the IP header fields for this fragment

- Set frag\_off (fragment offset)

Offset is counted in **8-byte units**:

offset >> 3

- Set MF (More Fragments) flag

If this is **not the last fragment**:

set IP\_MF

If it IS the last fragment:

do NOT set IP\_MF

- Handle IP options on FIRST fragment only

Because options must only appear in the first fragment.

## 9 Update total length and checksum

Since the header changed:

- tot\_len = header + fragment-size
- Recompute IP header checksum

## 10 Send the fragment

Call the output function.

If success → increase FRAGCREATES counter.

If failure → free this fragment and return error.

## 11 After loop: free the original SKB

Because we created new SKBs for all fragments, the original SKB is no longer needed.

Also increase FRAGOKS statistic.

Return success.

## SUMMARY — Key Differences Slow vs Fast Path

Feature	Fast Path	Slow Path
Data copying	 No	 Yes
SKB allocation	 Reuse	 Allocate new SKB
Performance	 Very fast	 Slow
When used?	GSO/frag_list packets	All other cases
Memory usage	Low	Higher
CPU usage	Low	High

## Analogy

Imagine you have a very long rope.

Fast path:

The rope is already cut into correct lengths.

You only put a tag on each piece and ship it.

Slow path:

You must measure the rope, cut it yourself into correct lengths, label each cut piece, and then ship.

That is why slow path is slow.

If you want, I can also explain:

- how defragmentation works
  - how tcpdump shows fragmented packets
  - ASCII diagram showing SKBs linked in slow path and fast path
- Just tell me!

Great question — now you're moving into **how routers handle fragmentation** and **how networks react when MTU changes dynamically**, especially during congestion.

Let me explain this step-by-step, like a teacher, with real examples.

## 1. Is Intermediate Fragmentation Allowed in IPv4?

YES — IPv4 allows routers to fragment packets.

 Who can fragment in IPv4?

Device	Allowed to fragment?
Sender (host)	YES
Routers (intermediate hops)	YES
Receiver	Reassembles only

This means when a packet moves across multiple networks:

Sender → Router1 → Router2 → Router3 → Destination

Any router can fragment it if needed.

## 2. How does a router decide to fragment?

Each router examines:

- The **outgoing interface MTU**
- Packet size
- DF (Don't Fragment) flag
  - ✓ If packet size > MTU **and DF=0**  
→ Router fragments.
  - ✓ If packet size > MTU **and DF=1**  
→ Router **cannot fragment**, so it sends back ICMP:

ICMP Destination Unreachable

Code: Fragmentation Needed

(MTU value included)

This triggers **Path MTU Discovery** (PMTUD).

## 3. How exactly do routers fragment?

Routers use the **same logic** as Linux ip\_fragment():

Fragmentation happens like:

Original packet: 4500 bytes, MTU = 1500

Router cuts into:

- Fragment 1 → 1480 bytes payload + 20 byte header
- Fragment 2 → next 1480 bytes + 20 byte header

- Fragment 3 → last 1480 bytes + 20 byte header (MF=0)

Router must:

- Copy the IP header to each fragment
- Update fragment offset
- Set/clear MF flag
- Recompute checksum
- Forward each fragment separately

## 4. What does the receiver do?

Receiver gathers fragments into a reassembly buffer using:

- Identification field (ID)
- Fragment offset
- MF flag

Only when **all fragments arrive**, the receiver reassembles.

If:

- fragments are lost,
- or timeout (default 30 seconds),  
then the full packet is dropped.

## 5. Why intermediate fragmentation is bad?

Fragmentation causes:

- More packets
- Bigger CPU load on routers
- Higher probability of loss
- Harder congestion control
- Receiver must reassemble

**TCP hates it**, because a single missing fragment drops the whole packet → retransmission of the full segment.

That's why PMTUD was invented.

## 6. What happens when MTU is reduced due to congestion?

Routers sometimes **lower MTU dynamically** during:

- Network congestion
- Tunnel encapsulation (GRE, IPsec, VXLAN)
- MPLS labels
- PPPoE overhead

This is called **MTU blackhole behavior**.

Two cases happen:

## Case 1: DF bit = 0

Packet can be fragmented anywhere.

→ Router fragments without telling the sender

No PMTUD involved.

## Case 2: DF bit = 1 (common for TCP)

TCP always sets DF during PMTUD.

→ Router cannot fragment → Sends ICMP “Fragmentation Needed”

Sender receives ICMP:

Need to reduce MTU to 1200 bytes.

Sender updates **Path MTU** for this route.

TCP then adjusts:

- MSS (Maximum Segment Size)
- Packet sizes

So packets stop exceeding MTU.

Congestion is indirectly controlled because **smaller packets reduce retransmission cost**.



## 7. What if the router silently drops fragments? (common!)

Some firewalls block ICMP, so PMTUD breaks.

This causes:

→ Blackhole MTU

Packets vanish, nothing returns.

Modern Linux uses **PLPMTUD (RFC 4821)** which probes MTU without ICMP.



## Realtime Example: MTU Drops Due to a Tunnel

Suppose normal Ethernet MTU = 1500

Now a GRE tunnel is added (+24 bytes overhead)

Effective MTU =  $1500 - 24 = 1476$

If a 1500-byte packet arrives:

If DF=0 → router fragments to 1476

If DF=1 → router sends ICMP “need fragmentation: 1476”

TCP then lowers MSS to:

$1476 - 20 \text{ (IP header)} - 20 \text{ (TCP header)} = 1436$

Sender now uses smaller packets → congestion avoided.



## 8. How Linux Kernel Handles Router Fragmentation Internally

Routers perform:

- Slow path fragmentation (almost always)
- Very rarely fast path (frag\_list) because routers don't use GSO/GRO like hosts

Routers:

- Create new SKBs
- Copy headers
- Slice payload
- Push to L2 queue
- Recalculate checksum

Exactly same logic as your slow path code.



### Summary Table

Situation	Router Action	Result
Packet > MTU & DF=0	Fragment	Packets continue
Packet > MTU & DF=1	Send ICMP Frag Needed	Sender lowers packet size
MTU reduces dynamically	Fragment OR ICMP	PMTUD adapts
Fragments lost	Host reassembly fails	Full packet dropped
ICMP blocked	MTU blackhole	TCP stalls

If you want, I can also explain:

- ✓ ASCII diagrams of fragmentation/defragmentation
- ✓ How TCP MSS interacts with MTU
- ✓ Why IPv6 forbids router fragmentation
- ✓ How NIC offloading (GSO/TSO) interacts with all this
- ✓ How Linux stores PMTUD state internally (dst\_entry, mtu\_info)

Just tell me what you want next!

# defragmentation

Friday, December 5, 2025 2:11 PM

Ok. Here is **ONLY DEFragmentation**, explained **like you are a complete beginner**, but still technically correct.

## ★ IPv4 Defragmentation – SUPER SIMPLE EXPLANATION

When a big packet arrives in pieces (fragments), the Linux kernel must **join the pieces back** before giving it to upper layers (TCP/UDP).

Think of receiving a **parcel split into many boxes**. The destination must **collect all boxes** and **assemble the full item**.

Here is the exact process in simple steps.

### ❖ STEP 1 — Each fragment arrives one by one

A fragment comes to the IPv4 layer.

Linux checks:

- Source IP
- Destination IP
- Packet ID
- Protocol
- The fragment offset
- MF bit (More Fragments)

These together identify **which packet this fragment belongs to**.

### 📦 STEP 2 — Linux finds or creates a “reassembly box”

Linux keeps a table of all packets that are currently being rebuilt.

This “box” is represented by:

```
struct ipq
```

It contains:

- A linked list to store fragments
- Total expected size (len)
- Bytes received so far (meat)
- Flags to mark first/last fragments
- A lock
- A timer

If a matching reassembly box doesn't exist → **create one**.

## STEP 3 — Put fragment into the box (sorted)

Each fragment is put into the correct position inside a **linked list**, sorted by offset:

Fragment list example:

[ offset 0 ] → [ offset 1480 ] → [ offset 2960 ]

Linux avoids overlaps and handles duplicate fragments here.

When inserting a fragment:

- Increase meat = number of bytes collected.
- If the fragment has MF=0 → mark it as the **last fragment** → set total length.

## STEP 4 — Wait until all fragments arrive

Linux checks:

Have I received the FIRST fragment?

Have I received the LAST fragment?

Does total meat == total expected length?

If **all fragments have arrived**, proceed to step 5.

If not, keep waiting.

A **timer** (default 30 seconds) tracks how long Linux waits.

If fragments don't arrive in time → assembly fails → send ICMP "time exceeded".

## STEP 5 — Reassemble

When all fragments are present:

1. Linux takes the **first fragment**.
2. Creates a **new large skb** big enough for:
  - IPv4 header
  - All fragment data
3. Copies all fragment data into it in order.
4. Fixes header fields (like total length).
5. Frees the old fragment list.

Now you have **one complete packet**.

## STEP 6 — Clean up

- Destroy the ipq structure.
- Stop its timer.
- Return the reassembled skb to the IPv4 stack.

Packet now moves normally to TCP/UDP.

## COMPLETE PROCESS IN ULTRA-SHORT FORM

1. Fragment arrives
2. Find/create reassembly queue (ipq)
3. Insert fragment in sorted order
4. If (ALL FRAGMENTS ARRIVED)  
→ call reassembly function
5. Build full packet
6. Delete fragment queue
7. Pass complete packet to upper layers

## THIS IS DEFRAgMENTATION. No fragmentation. Only joining.

If you want, I can also draw a diagram or map each simple step to exact kernel function names:

- ip\_defrag()
- ip\_find()
- ip\_frag\_queue()
- ip\_frag\_reasm()
- ip\_expire()

Just tell me!

Here you go — the simplest explanation of defragmentation + exact Linux kernel functions mapped to each simple step, like a clean teacher-style flow.

## IPv4 Defragmentation – Simple Steps + Exact Kernel Functions

We'll combine both:

- Simple logic (easy to understand)
- Actual Linux function names so you can match it to source code.

## STEP 1 — Fragment arrives → Kernel checks if it is a fragment

### **Simple:**

Check if this packet is cut into pieces.

### **Code:**

```
ip_local_deliver() calls:  
if (ip_is_fragment(ip_hdr(skb))) {  
    ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER);  
}
```

### **Function:**

ip\_is\_fragment() checks:

- MF bit (More Fragments)
- fragment offset



## **STEP 2 — Find existing fragment queue OR create new one**

### **Simple:**

Kernel looks for a “reassembly box” for this packet.

### **Code:**

Inside:

```
ip_defrag()  
└── ip_find()
```

### **Function: ip\_find()**

Searches in the IPv4 fragment hash table:

key = (source ip, dest ip, packet id, protocol)

If no entry exists → it creates a new one:

struct ipq (reasembly queue)



## **STEP 3 — Insert fragment into its correct position (offset-sorted)**

### **Simple:**

Put the fragment in the right order:

[0] → [1480] → [2960]

### **Code:**

```
ip_defrag()  
└── ip_frag_queue()
```

### **Function: ip\_frag\_queue()**

Tasks:

- ✓ Insert into sorted list
- ✓ Handle overlapping fragments
- ✓ Reject duplicates
- ✓ Increase meat = bytes received
- ✓ Record if this is first/last fragment (MF flag, offset)

## STEP 4 — Check if reassembly is complete

**Simple:**

Ask:

Do I have all fragments?

Conditions:

- First fragment must arrive (offset = 0)
- Last fragment must arrive (MF = 0)
- Total bytes collected == expected total size

**Code inside:**

`ip_frag_queue()`

Checks:

```
if (qp->last_in == true && qp->meat == qp->len)
    → reassemble now
```

## STEP 5 — Reassemble into a full packet

**Simple:**

Copy all fragment data into one new skb.

**Code:**

`ip_defrag()`

└─ `ip_frag_reasm()`

**Function: ip\_frag\_reasm()**

Steps:

6. Create **one big skb** for final packet
7. Copy data from each fragment in order
8. Fix IP header (tot\_len, frag\_off = 0, unset MF)
9. Free old fragment skbs
10. Return the complete full skb

## STEP 6 — Cleanup: remove fragment queue, timers

**Simple:**

Destroy the box because reassembly is done.

### **Code inside:**

ip\_frag\_reasm() calls:

inet\_frag\_kill()

Cleanup includes:

- Cancel timer
- Remove from hash table
- Free struct ipq

### **⌚ If fragments are missing → drop expiration timer**

**Simple:**

If fragments do not come within time (default 30s), discard all.

**Code:**

Timer callback:

ip\_expire()

It:

- Drops all fragments
- Frees ipq
- May send ICMP "time exceeded" message

### **✿ COMPLETE FLOW WITH NAME + SIMPLE MEANING**

Fragment arrives

↓

ip\_local\_deliver()

↓

Check ip\_is\_fragment()

↓

Call ip\_defrag()

↓

ip\_find() → find or create reassembly queue

↓

ip\_frag\_queue() → insert fragment in order

↓

If all fragments? → Yes → ip\_frag\_reasm()

→ build full packet

→ free fragment queue

→ return final skb

↓

If timeout → ip\_expire()

## ⌚ This is the ENTIRE IPv4 DEFRAAGMENTATION PIPELINE in Linux.

If you want:

- A diagram of memory layout
- Example with real packet offsets
- How NAT/firewall uses defragmentation
- How NIC/driver interacts with SKBs during defrag

Just say “**explain deeper**”!

Here is **the simplest possible explanation of struct ipq** — what it is, why it exists, and what every field means.

This is the structure:

```
struct ipq {  
    struct inet_frag_queue q;  
    u32 user;  
    __be32 saddr;  
    __be32 daddr;  
    __be32 id;  
    __be16 protocol;  
    u8 ecn;  
    int iif;  
    unsigned int rid;  
    struct inet_peer *peer;  
};
```

## ★ What is struct ipq? (Simple Explanation)

struct ipq = **one reassembly box for one fragmented IPv4 packet**

Every fragmented IPv4 packet needs a “container” where all pieces can be stored until the full packet is reconstructed.

Linux creates **one ipq** per unique packet based on:

- Source IP
- Destination IP
- IP ID
- Protocol (TCP/UDP/ICMP)

So:

**1 fragmented IP packet = 1 ipq structure**

Inside ipq, the kernel stores:

- ✓ All packet fragments
- ✓ Reassembly state
- ✓ Timer
- ✓ Source/dest/id/protocol info
- ✓ Bytes received so far
- ✓ Expected total length

That's all.

## ★ Breakdown of Each Field (Teacher-level Simple)

**1 struct inet\_frag\_queue q;**

This is the **main fragment queue and metadata**.

Contains:

- fragments → linked list of SKBs (sorted by offset)
- fragments\_tail
- meat → number of bytes collected
- len → total expected packet size (from last fragment)
- last\_in → flags: FIRST\_IN, LAST\_IN
- lock
- timer
- net → network namespace
- node → hash-table linkage

Simple meaning:

 **This holds all fragmented SKBs + counts + timers.**

**2 u32 user;**

Identifies **why fragmentation is happening**.

user can be:

- NF\_CONNTRACK\_DEFrag
- IP\_DEFrag\_LOCAL\_DELIVER
- IP\_DEFrag\_VS\_FORWARD

Simple meaning:

 **Which subsystem requested defrag?** (Local stack, conntrack, forward, etc.)

**3** \_\_be32 saddr;

Source IPv4 address of the fragmented packet.

Simple meaning:

🏠 Where the packet is coming from.

**4** \_\_be32 daddr;

Destination IPv4 address.

Simple meaning:

📍 Where the packet is going.

**5** \_\_be32 id;

**IP Identification field** (16 bits, stored as 32-bit BE).

Used to group fragments belonging to the same datagram.

Simple meaning:

ID The ID printed in tcpdump: this groups fragments together.

**6** \_\_be16 protocol;

The L4 protocol of the fragmented packet:

- TCP = 6
- UDP = 17
- ICMP = 1
- ESP = 50

Simple meaning:

💡 Which transport protocol is inside the packet.

**7** u8 ecn;

ECN bits from IP header (Explicit Congestion Notification).

RFC says that when reassembling, ECN must be computed properly.

Simple meaning:

⌚ Congestion bits — must combine ECN from all fragments.

**8** int iif;

Input interface index (ifindex).

The interface where the fragment arrived.

Simple meaning:

 **Which network interface the fragment came from.**

This prevents mixing fragments from different paths.

 **9 unsigned int rid;**

Router ID for defrag caching (used by conntrack + routing).

Used internally to help routing decide how to handle the completed packet.

Simple meaning:

 **Internal routing identifier for this reassembly.**

(Not very important for basic understanding.)

 **10 struct inet\_peer \*peer;**

Pointer to a peer structure (destination peer).

Used for rate-limits, ICMP quotas, tracking remote stats.

Simple meaning:

 **Information about the peer used for ICMP rate-limiting.**

## **FINAL SIMPLE SUMMARY**

struct ipq = one "reassembley box" for one fragmented IP packet.

It stores:

- all fragments (through inet\_frag\_queue)
- reassembly timers
- expected packet size
- bytes collected
- who sent it (saddr)
- who receives it (daddr)
- packet ID
- protocol (TCP/UDP)
- ECN bits
- input interface
- peer information
- subsystem that requested defrag (user)

Or even simpler:

 **ipq = everything the kernel needs to rebuild a fragmented packet safely and correctly.**

If you want, I can also:

- ✓ Draw ASCII diagrams of how fragments get stored inside ipq->q
- ✓ Show the exact memory layout of the SKB fragment list
- ✓ Explain overlapping fragment handling with examples
- ✓ Explain how ipq is stored in hash table and timers

Just tell me!

# Forwarding

Monday, December 8, 2025 10:06 AM

Below is a **clear, simplified, teacher-style explanation** of the entire forwarding logic you pasted — **step by step**, covering **what happens, why it happens, and how the kernel decides to forward or drop a packet**.

I will avoid unnecessary jargon and explain in a way that still respects how the Linux kernel actually behaves.

## ★ IPv4 Forwarding — Human-Friendly Explanation

Whenever Linux receives a packet, it may:

- **Deliver it locally** → to TCP/UDP
- **Forward it** → act as a router
- **Drop it** → if invalid, TTL expired, DF=1 with oversized packet, etc.

The function that handles forwarding is:

```
ip_forward(struct sk_buff *skb)
```

Let's go through the **exact steps inside ip\_forward()**, one by one.

### 1 Reject LRO (Large Receive Offload) Packets

! Why?

LRO merges many received TCP segments into one *huge* skb.

**Forwarding such a big skb is impossible**, because:

- The outgoing link MTU is small (usually 1500)
- LRO packets violate IP correctness
- Forwarding requires segmentation or fragmentation, which LRO does not support

☞ Kernel behavior:

```
if (skb_warn_if_lro(skb))
    goto drop;
```

💡 GRO (Generic Receive Offload) supports forwarding, but LRO does not.

### 2 Router Alert Option

If the IPv4 Router Alert option is present (IP\_ROUTER\_ALERT):

- Packet must be delivered to raw sockets interested in it
- Example uses: RSVP, IGMP, MPLS, PIM, etc.

```
if (opt->router_alert)
    ip_call_ra_chain(skb);
```

☞ Delivered to **all** raw sockets because raw sockets have no port numbers.

### 3 Drop non-host packets

eth\_type\_trans() marks packets depending on L2 destination MAC.

If packet is not destined to this machine (PACKET\_HOST):

- Drop it
- We only forward host-received packets

```
if (skb->pkt_type != PACKET_HOST)
```

```
goto drop;
```

## 4 TTL Check

Every router must:

- **Reduce TTL by 1**
- **Drop when TTL  $\leq 1$**
- **Send ICMP “Time Exceeded”**

This prevents routing loops.

```
if (ip_hdr(skb)->ttl <= 1)  
    goto too_many_hops;
```

If TTL hits 0  $\rightarrow$  icmp\_send(...ICMP\_TIME\_EXCEEDED...).

## 5 Strict Source Routing Check

If the packet uses **strict source route**:

- The sender specifies *exact next hops*
- If route does not match  $\rightarrow$  router must drop  

```
if (opt->is_strictroute && rt->rt_use_gateway)  
    goto sr_failed;
```

If failed  $\rightarrow$  send ICMP **Strict Routing Failed**.

## 6 Check MTU vs Packet Size

If packet is **bigger than outgoing interface MTU**, Linux checks:

1. **Is DF=1 (Don't Fragment)?**
2. **Is GSO/GRO disabled?**
3. **Is local\_df flag clear?**

If yes:

```
→ Cannot fragment  
→ Cannot forward  
→ Send ICMP Fragmentation Needed  
if (skb->len > dst_mtu(...) && DF=1)  
    icmp_send(...ICMP_FRAG_NEEDED...)
```

This is how **Path MTU Discovery** works.

## 7 Make skb Writable

Forwarding modifies:

- TTL
- Checksum
- Possibly L2 headers

So skb must not be shared.

```
if (skb_cow(skb, ...))  
    goto drop;
```

skb\_cow = copy-on-write.

## 8 Decrease TTL and Update Checksum

Exactly what routers do.

```
ip_decrease_ttl(iph);
```

## 9 ICMP Redirect (Optional)

Router sends ICMP Redirect when:

- Sender could have used a better gateway
  - Typical in simple LANs
- ```
if (rt->rt_flags & RTCF_DOREDIRECT)
    ip_rt_send_redirect(skb);
```

## 10 Set skb->priority

Forwarding path has no socket → no socket priority.

So kernel maps ToS/DSCP → priority using ip\_tos2prio.

## 1 1 Enter Netfilter Forward Hook

Before the packet is finally forwarded, NF\_INET\_FORWARD hook is called:

- firewall
- NAT
- filtering rules

```
return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, ..., ip_forward_finish)
```

## 1 2 ip\_forward\_finish() — Final Step

This function:

- ✓ Updates forwarding statistics
- ✓ Handles IP options if present
- ✓ Finally sends packet out with:  
dst\_output(skb)

Which eventually goes to the NIC driver.

## ★ What did we learn?

ip\_forward() does the following in order:

| Step | Check/Task                       | Reason                                           |
|------|----------------------------------|--------------------------------------------------|
| 1    | Drop LRO SKBs                    | Cannot forward giant merged packets              |
| 2    | Router Alert                     | Deliver to raw sockets                           |
| 3    | Drop PACKET_OTHERHOST            | Only packets marked for host should be forwarded |
| 4    | TTL≤1? Drop + ICMP               | Prevent routing loops                            |
| 5    | Strict source route check        | Obey sender's explicit path                      |
| 6    | MTU > size and DF=1? Drop + ICMP | Path MTU Discovery                               |
| 7    | Make skb writable (COW)          | TTL and checksum change                          |
| 8    | Decrease TTL + update checksum   | Required for all routers                         |
| 9    | Send ICMP Redirect if needed     | Optimize routing                                 |

|    |                           |               |
|----|---------------------------|---------------|
| 10 | Set priority              | Used by qdisc |
| 11 | Call NF_INET_FORWARD hook | Firewall/NAT  |
| 12 | ip_forward_finish()       | Output to NIC |

This is the *complete forwarding logic*.

★ If you want, I can now explain:

- ✓ Step-by-step “what happens during real forwarding on Linux”?
- ✓ How forwarding differs for multicast packets?
- ✓ Where fragmentation happens during forwarding?
- ✓ Add a full ASCII diagram showing all paths?

Just tell me — “explain multicast forwarding also” or “explain fragmentation in forwarding”.

# IPV4 ROUTING

Wednesday, December 10, 2025 10:31 AM

Below is a **clean, structured, deeply detailed theoretical explanation** of your paragraph.

I will also **highlight the important technical terms** and give **clear explanations** for each one.

## ★ Forwarding and the FIB — Detailed Theoretical Explanation

The paragraph describes how Linux handles **packet forwarding** and how the **routing subsystem** uses a structure called the **FIB (Forwarding Information Base)** to decide where packets should go. The text introduces core concepts such as routing, routing tables, default gateways, and explains how packets move between network layers inside the kernel.

Let's break it down in a very detailed yet clear manner.

## ◇ Highlighted Important Terms + Deep Explanation

Below I explain the entire paragraph and highlight key terms with detailed meaning.

### ◊ 1. Forwarding

Forwarding is the process where a machine receives a packet **not meant for itself** and sends it out through another interface toward its destination.

- A forwarding machine is often called a **router**.
- It acts like a bridge between different networks.
- The key idea: **A forwarding router does NOT take the packet to transport or application layer.**  
It handles it purely inside Layer 3 (IP).

### ◊ 2. Linux Networking Stack

This is the set of kernel subsystems responsible for handling:

- Ethernet frames (Layer 2)
- IP packets (Layer 3)
- TCP/UDP (Layer 4)
- Forwarding decisions
- Routing, queueing, firewalling, NAT, etc.

The part relevant to forwarding is the **IPv4 routing subsystem**.

### ◊ 3. Routing Subsystem

This is the **Layer 3 (IP layer)** component in the kernel responsible for:

- Maintaining routing tables
- Looking up destinations
- Deciding which outgoing interface a packet should use

- Handling forwarding

This subsystem contains the **FIB**.

## ◊ 4. FIB (Forwarding Information Base)

The FIB is the **kernel's optimized lookup table** that stores routing information.

Important notes:

- It is *not* exactly the same as the user-visible routing table from `ip route`.
- The FIB is built from routing tables but is optimized for **fast longest-prefix match** lookups.
- Routers need extremely fast lookup speeds, so the FIB is built in special data structures (like a trie).

## ◊ 5. Routing in Small Networks vs Core Routers

The paragraph distinguishes two scenarios:

### **Small Networks**

- Network topology rarely changes.
- Admins can configure routes manually using `ip route`.

### **Core Routers (Internet backbone)**

- Network topology changes **very frequently**.
- Thousands of routes change every second.
- The FIB must be updated dynamically.

To handle this:

- **Routing daemons** like OSPF, BGP, RIP run in userspace.
- Daemons update the kernel routing table.
- Kernel updates the FIB.
- Some systems may use **hardware offload** (ASICs, TCAMs) for faster routing.

## ◊ 6. Routing Daemons

Userspace programs responsible for learning and managing routes:

Examples:

- **BGP** → used by ISPs
- **OSPF** → interior routing protocol
- **RIP, IS-IS**, etc.

These daemons:

- Build their own routing tables.
- Synchronize important routes with the kernel's routing tables.
- The kernel converts those routes into the **FIB**.

## ★ Practical Example Given in the Text

The paragraph uses a very simple router example:

### **LAN1**

- Subnet: 192.168.1.0/24
- Connected to router interface: **eth0 (192.168.1.200)**

## LAN2

- Subnet: 192.168.2.0/24
- Connected to router interface: **eth1 (192.168.2.200)**

Packet Flow:

A packet from LAN1 going to LAN2 will:

Arrive on eth0 → pass through Layer 2 → go to Layer 3 → routing lookup → go out via eth1 → reach LAN2

This process is called **forwarding**.

## ◇ 7. Network Layers Involved

The paragraph highlights 3 layers:

### Layer 2 (Link Layer)

Handles Ethernet frames, MAC addresses.

### Layer 3 (Network Layer / IP Layer)

Handles IP addresses, routing, forwarding decisions.

### Layer 4 (Transport Layer)

Handles TCP/UDP, sockets, ports.

Forwarded packets **do not go to Layer 4**, because they are not meant for this machine.

This improves **performance**, because routing only requires Layer 3 processing.

## ◇ 8. Traffic Destined for the Router vs Forwarded Traffic

### Traffic to router (local traffic)

If the destination IP matches router's own IP, packet goes to:

Layer 2 → Layer 3 → Layer 4 → socket

### Forwarded traffic

If destination IP does NOT match router's IP:

Layer 2 → Layer 3 → routing lookup → new Layer 2 frame → send out

This avoids unnecessary overhead.

## ◇ 9. Default Gateway

A **default gateway** is a special route that catches all packets that do not match any more specific route.

It is represented as:

0.0.0.0/0

Meaning: "Match everything."

Default gateway example:

```
ip route add default via 192.168.2.1
```

This tells the kernel:

If you don't know where to send a packet, send it to 192.168.2.1.

## ◊ 10. Default Route

This is the routing entry that corresponds to the default gateway.

0.0.0.0/0 → means "**all destinations**".

This is extremely important in home networks and enterprise networks.

## ◊ 11. Key Concepts You Learned

The paragraph summarizes the following ideas:

✓ What forwarding is

Moving packets between interfaces based on routing rules.

✓ What routing tables are

Datastructures containing routing rules.

✓ What FIB is

Kernel-optimized routing database for fast forwarding.

✓ How packets move between layers in forwarding

They skip layer 4.

✓ What default gateway and default route mean

Special fallback route for all unknown destinations.

### ★ If you want, I can also explain:

- The exact function call path inside kernel during forwarding
- What fib\_lookup() does internally
- What data structures (fib\_table, fib\_info, fib\_nh) look like
- How the kernel's routing trie works
- How ARP/neighbour subsystem participates in forwarding
- How default route is stored in FIB

Just tell me what you want next.

Below is a **clear, deeply detailed, teacher-style explanation** of the entire paragraph. I will break it down so you understand exactly *what happens during a routing lookup in Linux*, what each structure means, and why it is used.

## ★ Performing a Lookup in the Routing Subsystem — Theoretical Explanation (Simplified + Deep)

When a packet enters the Linux networking stack (Rx path) or is being sent by a local process (Tx path), Linux must decide:

- Where should this packet go?
- Which interface should be used?
- What next-hop IP is required?
- Should packet be delivered locally or forwarded?

This decision is made by performing a **Route Lookup** in the **routing subsystem**.

### ◇ 1. Route Lookup Happens for Every Packet

Every single incoming or outgoing packet goes through a lookup in the routing subsystem.

- Rx path (receiving packet) → Kernel checks if packet is for local machine or must be forwarded.
- Tx path (sending packet) → Kernel checks how to route the outgoing packet to its destination.

### ◇ 2. Old Kernels (before 3.6) had TWO lookup stages

#### Stage 1: Routing Cache lookup

- A per-host cache (fast but caused issues)
- Lookup by destination IP
- If entry found → quick return

#### Stage 2: Full Routing Table lookup

- Slow, complete route lookup using FIB
  - ◊ Routing cache was removed in Linux 3.6

Because it caused:

- Cache poisoning attacks
- Thrashing (too many invalidations)
- Memory waste

So now, **only FIB performs the lookup**—no more routing cache.

### ◇ 3. Main Lookup Function: fib\_lookup()

Prototype:

```
int fib_lookup(struct net *net, const struct flowi4 *fip, struct fib_result *res)
```

Meaning of parameters:

*\*struct net net*

Represents the network namespace (Linux supports multiple network namespaces).

`*const struct flowi4 flp`

This is the **KEY** for lookup.

flowi4 contains:

- destination address
- source address
- Type of Service (TOS / DSCP)
- protocol
- incoming interface
- mark

It tells kernel:

➡ “Find a route for a packet defined by these characteristics.”

`*struct fib_result res`

This is where Linux stores the **OUTPUT** of the lookup.

If lookup succeeds:

- fib\_result is filled with route info
- fib\_lookup() returns **0**

If lookup fails:

- Returns **negative error** (like -ENETUNREACH)

## ◇ 4. flowi4 — The Routing Lookup KEY

This structure defines what the packet looks like from routing perspective.

Think of flowi4 as:

A packet description used for searching routing tables.

Contains:

- fl4\_dst → destination IP (most important)
- fl4\_src → source IP
- fl4\_tos → TOS field (QoS)
- fl4\_proto → TCP/UDP/ICMP
- fl4\_oif → outgoing device hint

It must be **initialized before calling fib\_lookup()**.

For IPv6, Linux uses flowi6.

## ◇ 5. FIB Tables Lookup

Linux maintains multiple FIB tables.

Lookup order:

### 1. Local FIB table

Used for:

- local addresses assigned to the machine
- broadcast
- multicast

- loopback  
If packet belongs to local machine, lookup ends here.

## 2. Main FIB table

Contains:

- default routes
- static routes (ip route add)
- routes learnt via routing daemons (BGP, OSPF, RIP)

If no match → routing error.

## ◇ 6. After Successful Lookup: Building dst Object

Once route is found, Linux builds a:

### ★ dst\_entry object

(also called *destination cache entry*)

struct dst\_entry is located in:

include/net/dst.h

It stores the **final forwarding decision**.

Every SKB (packet) eventually gets a dst assigned.

## ◇ 7. rtable — A Layer 3 Routing Decision

rtable embeds dst\_entry.

Meaning:

- rtable = Routing result for IPv4
- All routing decisions (forward or local delivery) are stored here
- Packet carries a pointer to its rtable entry

## ◇ 8. Most Important Part of dst\_entry: Callbacks

dst\_entry contains:

### input callback

Handles incoming packet:

- If local → deliver to process
- If forwarded → routing logic

### output callback

Handles outgoing packet:

- Decide next-hop
- Perform ARP lookup
- Send through device driver

Both callbacks take an SKB:

```
int (*input)(struct sk_buff *skb);
int (*output)(struct sk_buff *skb);
```

These callbacks determine:

- whether packet should be forwarded
- should be locally delivered
- should be dropped
- how fragmentation or MTU checks are handled
- how neighbour (ARP) resolution works

They are assigned based on the result of fib\_lookup().

## ★ Putting It All Together (Simple Flow)

For each packet:

- 1 Build a flowi4 describing the packet
- 2 Call fib\_lookup()
- 3 Kernel searches Local and then Main FIB tables
- 4 If match → build an rtable
- 5 Set dst->input or dst->output callbacks
- 6 Packet is processed by callback
- 7 Eventually goes to NIC driver or local socket

## ★ In Ultra Simple Terms

When Linux receives or sends a packet, it does:

Who is this packet for?  
How do I reach the destination?  
Which interface should I use?  
What next-hop?

To answer these:

- flowi4 = question paper
- fib\_lookup = exam
- fib\_result = exam result
- rtable = final routing plan
- dst\_entry = executor of that plan
- input/output callbacks = actual packet handlers

## ★ If you want, I can next explain:

- ✓ The complete structure of **flowi4**
- ✓ The structure of **fib\_result**
- ✓ How exact FIB trie lookup works (step-by-step)
- ✓ How routing lookup interacts with ARP

- ✓ How the kernel decides between forward vs local delivery
- ✓ How input and output callbacks behave in detail

Just tell me what you need next.

Below is a **much simpler but still detailed** explanation of **dst\_entry** so you get the full picture without drowning in kernel jargon.

## ✿ What is dst\_entry (in very simple words)

A **dst\_entry** is a small structure inside the Linux network stack that tells the kernel:

**“If you want to send this packet, use these settings, and call this function.”**

It is the **routing instruction sheet** for a packet.

Every packet (skb) carries a pointer to this dst\_entry.

## ✿ Why do we need it?

When Linux routes a packet (Rx or Tx), it first performs a lookup:

- Which interface to use?
- Which next-hop to use?
- What MTU?
- Is this unicast / multicast / blackhole / tunnel?
- What function should handle sending this packet?

The **result** of the routing lookup is placed into a dst\_entry.

Instead of doing route-lookup *again and again*, the packet keeps a pointer to this object.

## ✿ Think of it like this

Imagine the kernel is a delivery company:

- A **packet (skb)** = a parcel
- A **dst\_entry** = a label stuck on it telling:
  - **Which truck (interface) to use**
  - **Which driver function to call**
  - **How big the parcel is allowed to be (MTU)**
  - **Who the next hop is (neighbor)**
  - **Special rules (tunnel, drop, multicast)**

Whenever the kernel handles the packet, it just reads the label.

## Where is dst\_entry used?

It sits **between Layer-3 and Layer-2**:

L3 (IP layer)

|

| uses dst\_entry to decide where/how to send packet

v

L2 (Ethernet driver)

## What is inside a dst\_entry?

(Explained simply so a beginner can follow)

### ✓ 1. Outgoing device (dst->dev)

The network interface to use:

- eth0
- eth1
- wlan0
- tunnel device
- vif device

This tells the kernel: “Send this packet out through this NIC.”

### ✓ 2. The MTU (dst\_mtu)

Maximum size allowed for this path.

Used for:

- fragmentation
- PMTU (Path MTU Discovery)

If the destination path has MTU 1400 (e.g., tunnel), then kernel uses that.

### ✓ 3. Header length (header\_len)

How much extra L2 header will be added?

Examples:

- Ethernet = 14 bytes
- VLAN = 18 bytes
- Tunnel (GRE/VXLAN) = much more

This lets IPv4 adjust packet size.

### ✓ 4. Next-hop neighbor (dst->neighbour)

This contains the link-layer info:

- For IPv4 → ARP entry

- For IPv6 → NDP entry  
Kernel uses this to get destination MAC address.

## ✓ 5. Callbacks: **the most important part**

`dst_entry` contains two special function pointers:

- **input**
- **output**

These say:

- **input:** “What to do if this packet is coming *towards* the system through this route?”
- **output:** “How to *send* the packet out through this route?”

Examples of output actions:

- Do ARP lookup
- Do fragmentation
- Apply netfilter
- Send into a tunnel
- Send to multiple interfaces (multicast)

This is where forwarding happens.

## ✓ 6. Route metrics

Like:

- Hop limit
- Preferred source address
- Priority
- Expiry

These affect routing decisions.

## ✓ 7. Reference count

Makes sure the dst is not freed while packets or sockets still use it.

## ✿ How a packet uses a `dst_entry` (simple flow)

When a packet is sent OUT:

### 1. Kernel performs route lookup

→ Finds next-hop, device, MTU, metrics

→ Creates a `dst_entry`

### 2. Packet attaches the route

`skb->dst = dst`

### 3. Later, kernel calls

`dst->output(skb)`

This is how actual transmission happens.

#### 4. Output function usually does:

- netfilter POSTROUTING
- neighbor (ARP/NDP) resolve
- build Ethernet header
- queue to driver

## ✿ Special behavior examples

### ◊ Normal unicast

output points to:

→ ip\_output → ip\_finish\_output

### ◊ Multicast

output duplicates packet to multiple interfaces + applies TTL scope.

### ◊ Blackhole route

output simply drops the packet.

### ◊ Tunnel route

output encapsulates the packet inside GRE/VXLAN/IPIP.

This is why callbacks are powerful.

## ✿ Very short summary

dst\_entry is the kernel's:

- ✓ Routing result
- ✓ Output function
- ✓ Device pointer
- ✓ MTU + metrics
- ✓ Neighbor (ARP/NDP)
- ✓ Per-route behavior description

Attached to every packet so future code knows:

**“How do I handle this packet?”**

If you want, I can now also give:

- A super-simple ASCII diagram of the dst\_entry lifecycle
- A simple C-style pseudo-structure of dst\_entry
- A “beginner readable” flow from fib\_lookup() → dst\_entry → skb → driver

Just tell me what you want next.

## ✿ What happens in Linux networking

### 1. User adds route or routing daemon updates route

```
ip route add 10.1.1.0/24 via 192.168.0.1
```

This updates the **routing table**.

### 2. Kernel converts/compiles it into FIB entries

- Makes trie structures
- Organizes prefixes
- Removes unnecessary administrative data
- Creates fib\_info and fib\_node structures

### 3. When packet arrives, kernel calls fib\_lookup()

fib\_lookup() searches **only the FIB** (not the routing table!)

### 4. fib\_lookup() returns a fib\_result

Kernel uses it to build:

- dst\_entry
- rtable
- Neighbor entry (ARP)

→ Used to forward the packet.

## ✿ Why do we need both?

✓ Routing table is for *humans and routing daemons*.

- Easy to read
- Supports many protocols
- Stores metadata

✓ FIB is for *fast kernel forwarding*.

- Trie-based
- O(1) or O(logN) lookups
- Used millions of times per second

If kernel used routing table directly, forwarding would be too slow.

Below is a **simple but detailed, beginner-friendly explanation** of the entire paragraph you posted.

I will explain:

- **What struct rtable is**
- **Why dst\_entry is inside it**
- **Every member with simple explanation**
- **How Linux uses this structure during routing**
- **How multicast, broadcast, local delivery, forwarding all relate**
- **Important terms from the paragraph highlighted & explained**

## What is struct rtable? (Simple Explanation)

rtable = **Routing Table Entry for a specific packet**

(used after FIB lookup)

When a packet comes in (RX) or is being sent (TX), Linux does:

1. Perform route lookup → using **fib\_lookup()**
2. Convert result into a **struct rtable** object
3. Attach this rtable to the packet → **skb\_dst\_set()**
4. The kernel then uses this object to decide:
  - Deliver locally?
  - Forward?
  - Send ICMP redirect?
  - Handle multicast?
  - What MTU to use?
  - Which device to transmit on?

So **rtable** is the real runtime route used for a packet.

## Why is dst\_entry the first field?

```
struct rtable {  
    struct dst_entry dst; // MUST BE FIRST  
    ...  
}
```

Because Linux networking uses **polymorphism**.

dst\_entry is a **base class** like in object-oriented programming.

rtable "inherits" from dst\_entry.

This allows code like:

```
skb_dst(skb)->input(skb);  
skb_dst(skb)->output(skb);
```

No matter the type (IPv4 route, IPv6 route, MPLS, tunnel, etc.)

→ Kernel always sees only dst\_entry.

## Now, the simple explanation of every field

✓ dst\_entry dst;

Base structure storing:

- input() callback
- output() callback
  - These functions determine:
- How to handle the packet (local deliver, forward, drop, error, multicast)

✓ int rt\_genid;

Used for route generation changes.

If routes change → old cached routes are invalidated.

✓ unsigned int rt\_flags;

These are **important flags describing the type of route**.

Let's explain the important flags mentioned:

#### ◇ *RTCF\_BROADCAST*

Destination is a **broadcast address**.

Example: 192.168.1.255 or 255.255.255.255.

Used to:

- Send ARP replies
- DHCP broadcast
- Forwarding is NOT allowed

#### ◇ *RTCF\_MULTICAST*

Destination is **multicast** (224.0.0.0 – 239.255.255.255)

Used for:

- IGMP
- OSPF
- mDNS
- Routing protocols
- IPTV

Triggers multicast-specific input handler:

ip\_mr\_input()

#### ◇ *RTCF\_DOREDIRECT*

Kernel should send **ICMP Redirect**.

This happens when:

- Packet comes in on interface X
- Kernel forwards out on same interface X
- A better gateway exists on that interface

Used to optimize routing in LANs.

#### ◇ *RTCF\_LOCAL*

Destination is **local to the machine**.

Examples:

- Packet destined to 192.168.1.20 (your machine)
- Packet destined to 127.0.0.1 (loopback)
- Broadcast can also be local
- Multicast can also be local

## ★ More fields

✓ \_\_u8 rt\_is\_input;

1 → This route is for **incoming** packet

0 → This route is for **outgoing** packet

✓ `__u8 rt_uses_gateway;`

1 → Next-hop is a **gateway IP**

0 → Directly connected (no gateway)

✓ `int rt_iif;`

Incoming interface index (eth0 = 2, eth1 = 3 etc.)

Used for:

- Routing
  - Reverse Path Filtering (RPF)
  - Multicast RPF check
  - ICMP redirect
- ✓ `__be32 rt_gateway;`

Gateway address if this route uses a gateway.

Example:

Packet going to 8.8.8.8

Your gateway is 192.168.1.1

→ `rt_gateway = 192.168.1.1`

✓ `u32 rt_pmtu;`

Path MTU.

Smallest MTU along the path.

Example:

Your interface: MTU 1500

Downstream router path: MTU 1400

→ `rt_pmtu = 1400`

Used to avoid fragmentation.

## ★ Why is `fib_compute_spec_dst()` mentioned?

Kernel sometimes needs **reverse routing**.

Example:

You received a packet from 10.1.1.10

You want to send ICMP reply

So you need: **which source address should reply use?**

Older kernels stored `rt_spec_dst`.

New kernels compute it dynamically using:

`fib_compute_spec_dst(skb)`

so the field was removed.

## ★ Most important part: callbacks

From your text:

- If packet is unicast for local host → `ip_local_deliver()`
- If packet must be forwarded → `ip_forward()`

- If packet is multicast → ip\_mr\_input()
- If prohibited → ip\_error()
- If outgoing packet → ip\_output()

These functions live in **Layer 3 (IPv4)**.

### This is how Linux decides what to do with a packet

When fib\_lookup() finishes, the kernel creates an rtable with:

```
dst->input = ip_forward OR ip_local_deliver OR ip_mr_input ...
dst->output = ip_output
```

Then the packet processing simply does:

```
skb_dst(skb)->input(skb);
```

This is how routing decisions are executed.

### Why is this design important?

Because it allows extremely fast forwarding:

- Only one indirect function call
- No large if-else chains
- No repeated routing lookup
- Each packet carries its precomputed route

### Summary in very simple words

struct rtable is:

- A **per-packet routing decision container**
- Contains important flags about the route (local, broadcast, multicast, uses gateway)
- Contains MTU, gateway, incoming interface
- Contains callbacks that handle packet delivery/forwarding
- It embeds dst\_entry, which gives polymorphic routing behavior
- It is created after FIB lookup and attached to the skb

If you want next:

- explain fib\_result
- explain how dst\_entry → neighbor → ARP → dev\_queue\_xmit works
- explain routing for multicast in depth (RPF, mroutes, mfc)

Just say “**continue**”!

# Fib Result

Wednesday, December 10, 2025 1:57 PM

Alright — here is a **simple but still detailed** explanation of **struct fib\_result** and how it fits into the routing lookup process, without overwhelming complexity.

I'll break it down like you are learning from a teacher.

## What is struct fib\_result?

It is a **temporary bucket** used only during routing lookup.

It does not store routing entries permanently.

Instead, it is a **container into which the routing subsystem writes the outcome** of a lookup.

Think of it like:

"Hey kernel, here is a packet. Can you find a route and tell me all the important information? Put the results here in fib\_result."

## Why do we need fib\_result?

Because fib\_lookup() must return many pieces of information:

- Which route matched?
- What is the next-hop?
- What type of route is this?
- What scope is it?
- Which table provided this route?
- Which fib\_info (full route data) should be used?

A simple integer return code is not enough — so we store all details in this struct.

## Members of struct fib\_result explained simply

```
struct fib_result {  
    unsigned char prefixlen;  
    unsigned char nh_sel;  
    unsigned char type;  
    unsigned char scope;  
    u32 tclassid;  
    struct fib_info *fi;  
    struct fib_table *table;  
    struct list_head *fa_head;  
};
```

## ◊ 1. prefixlen

Netmask length (0–32)

Examples:

| Route            | prefixlen |
|------------------|-----------|
| 0.0.0.0/0        | 0         |
| 192.168.1.0/24   | 24        |
| 192.168.1.100/32 | 32        |

Used for longest prefix match — more specific routes win.

## ◊ 2. nh\_sel

Nexthop selector

- Simple routes → 0
- Multipath (ECMP) routes → choose which next-hop to use

Example:

default via 10.0.0.1

default via 10.0.0.2

The kernel keeps both nexthops → nh\_sel chooses one based on hashing.

## ◊ 3. type

### MOST IMPORTANT FIELD

It decides what to do with the packet.

Common types:

| Type            | Meaning                      |
|-----------------|------------------------------|
| RTN_UNICAST     | forward packet to next-hop   |
| RTN_LOCAL       | deliver to local socket      |
| RTN_BROADCAST   | treat as broadcast           |
| RTN_MULTICAST   | multicast routing input      |
| RTN_UNREACHABLE | drop + send ICMP unreachable |
| RTN_BLACKHOLE   | drop silently                |
| RTN_PROHIBIT    | drop + send ICMP prohibit    |
| RTN_NAT         | NAT routing entry            |

This is where routing rules decide **how the packet behaves**.

## ◊ 4. scope

Routing scope → how wide the route is valid.

Common values:

| Scope             | Meaning                                   |
|-------------------|-------------------------------------------|
| RT_SCOPE_LINK     | only valid on directly attached interface |
| RT_SCOPE_HOST     | only for local host                       |
| RT_SCOPE_UNIVERSE | internet-wide                             |

Example:

Your IP address route → scope **host**

Your LAN route → scope **link**

Your default route → scope **universe**

## ◊ 5. **tclassid**

Used for **traffic control (TC)** classification.

Normally zero — only advanced QoS setups use it.

## ◊ 6. **fi** → *struct fib\_info*

This is the **core routing entry**.

**fib\_info** contains:

- nexthop list
- gateway IP
- output interface
- metrics
- scope
- flags

**fib\_result** only points to **fib\_info**.

The actual permanent route is stored inside **fib\_info**.

## ◊ 7. **table**

Which routing table this result came from:

- local
- main
- default
- custom tables (policy routing)

## ◊ 8. **fa\_head** → **fib\_alias list**

Used for optimization.

Why?

A single destination prefix may have multiple route entries with different:

- TOS

- priorities (metrics)  
fa\_head points to the list so fib\_lookup() can evaluate the best match.

## How is fib\_result used in packet processing?

### RX path

Packet arrives → routing lookup

fib\_lookup()

↓

fills fib\_result

↓

create rtable (full dst\_entry)

↓

skb->dst = &rtable.dst

↓

call dst->input(...)

This decides:

- forward packet? → ip\_forward()
- deliver locally? → ip\_local\_deliver()
- is multicast? → ip\_mr\_input()
- drop and send ICMP? → ip\_error()

### TX path

Local application sends packet:

fib\_lookup()

↓

fib\_result built

↓

dst\_entry.output = ip\_output()

Packet goes out to next-hop or gateway.

## Relationship Between All Structures

fib\_lookup()

|

|---> builds fib\_result

|

```
--> points to fib_info (actual route entry)
|
--> contains fib_nh (next-hops)
|
--> tells routing code which table matched
|
--> tells type (UNICAST / LOCAL / MULTICAST / etc.)
|
--> builds rtable (embed dst_entry)
```

fib\_result → temporary lookup result

fib\_info → permanent route storage

rtable / dst\_entry → final cached output used for real packet routing

## ⌚ In short

**fib\_result = “routing decision summary”**

It answers:

- Where should this packet go?
- What next-hop?
- What route type?
- Which table matched?
- How long was the prefix?
- Which nexthop to choose?
- Who owns the fib\_info?

If you want next:

I can explain:

- fib\_info in detail
- fib\_nh structure (next-hop)
- How rtable gets created
- Difference between input & output functions
- How policy routing interacts with fib\_result

Just tell me **next topic.**

# Fib table

Wednesday, December 10, 2025 2:22 PM

Below is a **very clear, very detailed, teacher-style explanation of fib\_table** — what it is, why it exists, how it works, how it connects to fib\_info, and a concrete example to make everything click.

I will also extract and explain **all important concepts** from the paragraph you gave.



## 1. What Is fib\_table? (Simple Words)

fib\_table represents **one full routing table** inside the Linux kernel.

You normally think Linux has a single routing table, but internally it has **many**:

| Table Name    | Table ID | Purpose                                     |
|---------------|----------|---------------------------------------------|
| local         | 255      | Routes for local IPs (your host addresses)  |
| main          | 254      | All normal routes (LAN, WAN, default route) |
| default       | 253      | Rare fallback table                         |
| Custom tables | 1..252   | Used by policy routing (ip rule)            |

Each of these is represented in the kernel as **one fib\_table object**.



## 2. Why Do We Need fib\_table?

Because the Linux routing system must support:

- ✓ Multiple routing tables

Users can create custom routing tables in /etc/iproute2/rt\_tables, and ip rule chooses which table to use based on:

- source IP
- fwmark
- UID
- incoming interface
- TOS
- routing policy

So the kernel cannot store “one huge list of routes”.

It needs **multiple organized tables** → that is what fib\_table provides.



## 3. What Does One fib\_table Contain?

A fib\_table contains:

- ✓ Many route prefixes (192.168.1.0/24, 10.0.0.0/8, etc.)

These prefixes are stored in a **fib trie** (data structure optimized for longest-prefix-match).

- ✓ Each route entry points to a fib\_info object

Example:

Route entry: 192.168.1.0/24

`fib_info: { gateway = 10.0.0.1, dev = eth0, metric = 100 }`

Many route entries can share the same `fib_info`.

### ✓ Table-wide operations

`fib_table` provides methods to:

- insert a route
- delete a route
- perform LPM lookup (lookup which prefix matches)
- iterate over routes
- notify netlink on changes

## ⌚ 4. Where Does `fib_table` Fit in Packet Routing?

When routing a packet, kernel calls:

`fib_lookup(net, fl4, result)`

`fib_lookup` checks multiple `fib_tables` in order:

1. local table
2. main table
3. default table
4. custom tables (based on ip rules)

The first table to return a match → routing is done.

So `fib_table` is essential because:

Routing lookup = searching multiple `fib_tables`

## ❖ 5. Relationship Between `fib_table`, `fib_info`, `fib_result`

Here is the hierarchy in simple form:

```
fib_table (routing table)
  |--- route entry: prefix/mask
    |   |--- fib_info (next-hop info)
    |--- route entry: prefix/mask
      |   |--- fib_info (next-hop info)
```

When you write:

`ip route add 192.168.2.0/24 via 10.0.0.1 dev eth0`

Internally:

- `fib_table` ("main") gets a new prefix entry
- `fib_info` contains nexthop, device, metrics, RTT info

- multiple fib\_table entries may share the same fib\_info

## 6. Why store fib\_info in a hash table? (fib\_info\_hash)

Because:

- Many routing entries have identical nexthop data
- Storing duplicate nexthop info wastes memory

Linux optimizes:

Multiple prefixes → One shared fib\_info

Example:

192.168.1.0/24 via 10.0.0.1 dev eth0

192.168.1.10/32 via 10.0.0.1 dev eth0

Both share the same fib\_info.

## 7. Important Functions (Explained simply)

### `fib_create_info()`

- Creates a new fib\_info when adding a route
- Inserts it into fib\_info\_hash
- Increments fib\_info\_cnt

### `fib_find_info()`

- Looks for an existing fib\_info with the same nexthop
- Reuses it if identical

### `free_fib_info()`

- Deletes a fib\_info
- Decrements fib\_info\_cnt

### `fib_info_lock`

Spinlock to protect access across CPUs.

## 8. fib\_info\_cnt (Simple Explanation)

A counter keeping track of:

- How many active fib\_info objects exist**
- Useful for debugging and memory management
- Helps kernel decide when to resize fib\_info\_hash

## 9. What Does fib\_table Look Like in the Kernel? (Conceptually)

Imagine a box like this:

`fib_table (main)`

└─ 0.0.0.0/0 → fib\_info\_A (gateway 10.0.0.1)

- └── 10.0.0.0/8 → fib\_info\_B (dev eth1)
- └── 192.168.1.0/24 → fib\_info\_A (same as first!)
- └── 192.168.1.100/32 → fib\_info\_C
- └── 224.0.0.0/4 → fib\_info\_D (multicast routing)

Note that multiple prefixes can share the same fib\_info.

## 10. Real Example: Routing Table and fib\_table Work Together

Let's say system routing table is:

```
default via 10.0.0.1 dev eth0
192.168.1.0/24 dev eth1
192.168.1.100 via 10.0.0.1 dev eth0
```

Packet destined to: **192.168.1.50**

Step-by-step:

**1** *fib\_lookup() checks local table*

→ Nothing

**2** *fib\_lookup() checks main table*

→ Longest prefix match = /24

The fib\_table(main) returns:

fib\_info (dev eth1)

prefixlen = 24

type = RTN\_UNICAST

**3** *Packet gets an rtable + dst\_entry*

→ so it knows:

- how to forward
- which mtu to use
- which gateway (if any)
- which input function (ip\_forward)

## 11. Important Points Extracted from the Paragraph (Explained Clearly)

You gave this text:

I'll rewrite the important parts in simple terms:

✓ fib\_table = main routing structure

Used to store all route entries.

✓ `fib_info` = holds next-hop information

Many prefixes can share the same `fib_info`.

✓ `fib_create_info` = creates `fib_info`

Also inserts it into `fib_info_hash`.

✓ `fib_info_hash` = hash table to detect duplicates

Avoids creating multiple identical nexthops.

✓ `fib_info_laddrhash` = special hash when route uses `prefsrc`

Used when selecting source address.

✓ `fib_info_cnt` = number of `fib_info` objects

Increases / decreases as routes added/removed.

✓ `spinlock fib_info_lock` = protects shared data

✓ `fib_table` is central to routing lookup

Every lookup uses `fib_table` → returns `fib_result`.

## ⌚ Summary (Simple + Full)

- `fib_table` = routing table instance
- **Used for** storing prefixes and performing LPM lookups
- **Why needed?** Because Linux supports multiple routing tables
- **Each entry** points to a `fib_info` (shared nexthop info)
- `fib_info_hash` avoids duplicate nexthops
- `fib_create_info/fib_find_info` manage `fib_info`
- `fib_info_cnt/lock` maintains consistency
- **Example** shows how `fib_table` selects correct route

If you want next:

I can explain:

⌚ struct `fib_table` each field in detail

⌚ how kernel performs **longest-prefix match (LPM)**

⌚ how multipath ECMP works in `fib_info`

⌚ how policy routing selects which table to search

Just tell me “next”.

Below is a **simple but detailed explanation** of the **FIB table**, *why* it exists, *what* it contains, and *how* it connects to `fib_result` and `dst_entry (rtable)` inside the Linux IPv4 routing subsystem.

I will explain it like a step-by-step story so everything becomes clear.

## 1. What is a FIB Table?

**FIB** = *Forwarding Information Base*

It is simply a **routing table inside the Linux kernel**.

Just like userspace has:

- ip route show
- route -n

the **kernel stores those routing entries** inside an internal structure called **fib\_table**.

Think of a fib\_table as:

“A big bookshelf where all routing entries for a particular routing table are stored.”

Linux has **multiple FIB tables**:

| Table ID   | Name           | Purpose                                    |
|------------|----------------|--------------------------------------------|
| <b>255</b> | local          | Routes for local IPs, broadcast, multicast |
| <b>254</b> | main           | Normal routing table (ip route add)        |
| <b>253</b> | default        | Used sometimes when main has no match      |
| custom     | 100, 200, etc. | Policy routing tables                      |

Each of these is represented in kernel by:

```
struct fib_table { ... }
```

## 2. Why Do We Need a FIB Table?

Because the kernel must answer one simple question for every packet:

**“Where should this packet go?”**

For every RX packet (forward) and for every TX packet (locally generated), the kernel must quickly find:

- Which **route** matches the destination IP?
- Which **device** should send it?
- Whether a **gateway** is needed?
- Whether it is **local, broadcast, or multicast?**
- What **source IP** to choose?
- What **MTU** to use?

The **FIB table** is the storage for this routing information.

## 3. What Does a FIB Table Actually Contain?

A FIB table contains **routing entries**, but not directly.

Each entry is broken into components:

### A. **fib\_alias**

Represents a route prefix (ex: 192.168.1.0/24)

Used when multiple routes have same prefix.

## B. fib\_info

The actual routing data:

- nexthop(s)
- device
- metric
- gateway
- scope
- prefsrc

Think of **fib\_info** as the “meat” of the routing entry.

## C. fib\_nh (nexthop)

Inside fib\_info:

- out interface
- gw
- flags
- scope

## D. Trie Node

Kernel uses a **trie** (prefix tree) to store and search routes.

fib\_table → trie → leaf → alias → fib\_info → fib\_nh

## ✳ 4. How is a FIB Table Used?

Whenever a packet arrives or is transmitted, kernel calls:

`fib_lookup(net, flowi4, fib_result)`

This:

5. Finds appropriate FIB table (local → main → default)
6. Searches its trie
7. Selects best matching prefix (longest prefix match)
8. Fills a **fib\_result** structure

## 📦 5. What is fib\_result and why do we need it?

`fib_result` is a **temporary object** used during lookup.

It stores the *results* of the FIB search:

```
struct fib_result {  
    prefixlen;  
    type;      // unicast, local, multicast...  
    scope;  
    nh_sel;    // nexthop chosen  
    struct fib_info *fi;  // routing data  
    struct fib_table *table;  
    struct list_head *fa_head; // alias list
```

```
};
```

Simply:

`fib_result` = “Here is the route that matches this packet.”

But **`fib_result` is NOT cached**.

It is only used during lookup.



## 6. How `fib_result` Connects to `dst_entry` (`rtable`)

After a successful lookup, kernel creates a **`dst_entry`**, specifically an **`rtable`**, which stores the “final routing decision” for an SKB.

**Summary:**

`fib_table` → `fib_lookup()` → `fib_result` → `rtable(dst_entry)`

Let’s see how:

**Step 1: `fib_result` contains `fib_info`**

`fib_info` contains:

- device
- gateway
- MTU
- metrics
- nexthop

**Step 2: Kernel calls**

`rt_dst_alloc()`

`rt_fill_info()`

They fill an **`rtable`**:

```
struct rtable {  
    struct dst_entry dst; // base routing object  
    u32 rt_gateway;  
    u32 rt_pmtu;  
    u8 rt_type; // UNICAST, LOCAL...  
    struct net_device *dev; // outgoing interface  
};
```

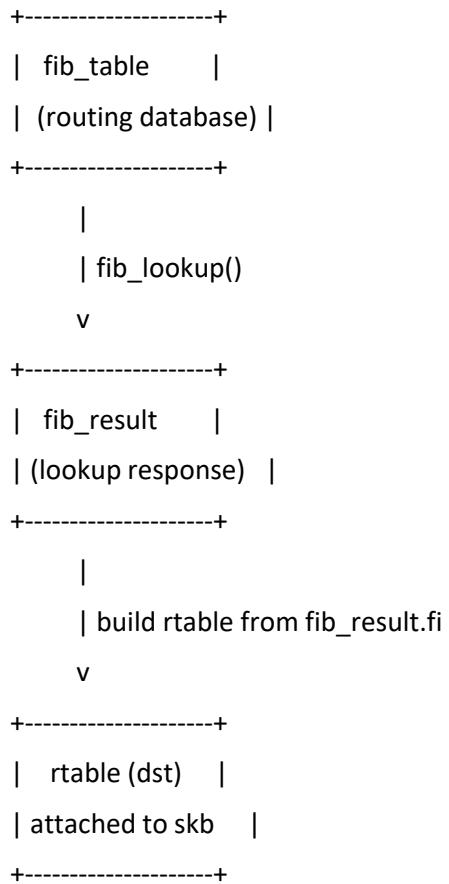
**Step 3: `rtable` is attached to SKB**

`skb->dst = &rtable->dst;`

This allows the packet to continue in the IP output/input pipeline.



## Relationship Between All Three



In simple words:

- **fib\_table** = *where routes live*
- **fib\_result** = *which route matched*
- **rtable (dst\_entry)** = *the decision attached to the packet*

## Putting It In a Simple Story

Imagine the kernel is a post office:

### **1** FIB Table

This is the big book containing all delivery routes ("if address starts with 192.168 → give to delivery guy eth0").

### **2** fib\_result

A postmaster looks at a letter and checks the book. He writes on a sticky note: "Deliver via eth0 using gateway 192.168.1.1".

### **3** rtable/dst\_entry

The sticky note gets attached to the letter.

Now every next handler (firewall, output routine, driver) reads this sticky note and knows what to do.

## If you want, I can also explain:

- fib\_alias in detail
- trie implementation

- fib\_rules and policy routing
- how multipath routing works
- how dst cache works

Tell me what you want next!

Here is a **simple, deep, and structured explanation** of the fib\_table structure, what each field means, why the kernel needs it, and how it fits into the routing subsystem.

I'll make it easy but still technically correct.

## What Is a fib\_table?

A **fib\_table** represents one routing table inside the Linux kernel.

Linux supports multiple routing tables:

- 254 → main (RT\_TABLE\_MAIN)
- 255 → local (RT\_TABLE\_LOCAL)
- 253 → default
- Custom tables (e.g., 100, 200) for **policy routing**

Each of these is implemented internally as a fib\_table.

So:

A fib\_table = “A full routing table stored in kernel memory.”

## The fib\_table Structure

```
struct fib_table {
    struct hlist_node tb_hlist;
    u32      tb_id;
    int      tb_default;
    int      tb_num_default;
    unsigned long  tb_data[0];
};
```

Let's explain each field one-by-one in simple but deep detail.

### 1. struct hlist\_node tb\_hlist;

This links each fib\_table into a **global hash list** of tables.

The kernel keeps a list like:

255 → local table

254 → main table

253 → default table

100 → custom table

200 → custom table

...

All these tables live inside a global list called:

`fib_tables`

This link allows the kernel to find a table by ID.

### 2. u32 tb\_id; — Table Identifier

This field tells the kernel **which routing table this is**:

Examples:

| Table ID             | Meaning               |
|----------------------|-----------------------|
| 255                  | local routing table   |
| 254                  | main routing table    |
| 253                  | default routing table |
| custom (100, 200...) | user-defined tables   |

**Why is tb\_id important?**

During a routing lookup the kernel checks tables in this order:

9. **Local table first → tb\_id = 255**
10. **Main table → tb\_id = 254**
11. Custom tables depending on rules

This matches the RFC: local routes must override normal unicast routes.

### 3. int tb\_default;

This is an **index/marker** used internally by the trie (fib\_trie) implementation.

Its meaning is **not the same as default route**.

It is mainly used for:

- internal fast lookup optimization
- marking the default leaf inside the trie

You can think of it as:

“Where in the trie is the default entry kept?”

#### 4. int tb\_num\_default;

This is **very important**.

It counts **how many default routes exist** in this table.

A default route is:

ip route add default via 192.168.1.1

or

0.0.0.0/0

Why does the kernel store this number?

Because default routes are **very frequently accessed** if no better-prefix matches.

Rather than scanning the route list every time, the kernel:

- increments this count when adding a default route
- decrements it when deleting one

This gives a quick check:

if (table->tb\_num\_default == 0)

    no default route exists

This improves performance dramatically in routing lookup.

#### 5. unsigned long tb\_data[0];

This one confuses everyone at first.

This is simply a placeholder for:

##### The root node of the FIB trie (fib\_trie)

The trie is where the real routing entries live.

fib\_table.tb\_data --> fib\_trie (tree structure)

Inside this trie, routes are stored based on prefix length:

/32 exact routes (local IPs)

/24 LAN routes

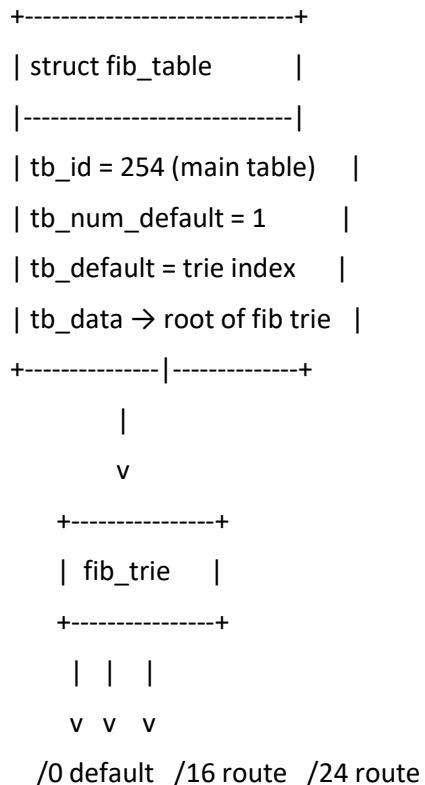
/16 networks

/0 default route

So tb\_data[0] is **not an array**, it is a **flexible array member**

— memory is extended beyond this structure to hold the actual trie root.

## Putting It All Together — Simple Visualization



## How fib\_table connects to fib\_result and dst\_entry

Routing lookup process:

### Step 1 → Looking in FIB tables

fib\_lookup()

- |— checks local table (255)
- |— checks main table (254)
- └— returns fib\_result

### Step 2 → fib\_result contains:

- prefixlen
- route type
- pointer to fib\_info
- nexthop selection

### Step 3 → Kernel converts fib\_result → rtable

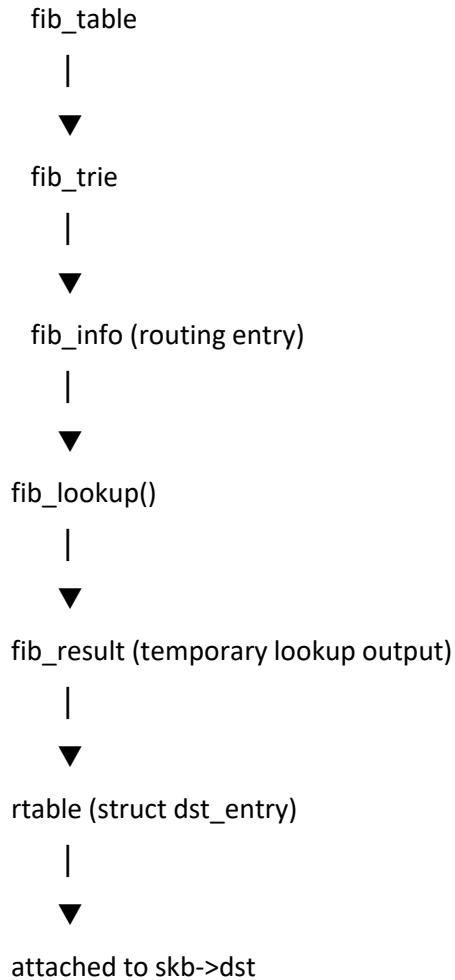
rtable → embedded inside dst\_entry

struct dst\_entry {

  input() and output() handlers

```
    MTU  
    metrics  
    cache lifetime  
}
```

## FULL CONNECTION MAP



## Quick Summary (Easy to Remember)

| Component                 | Meaning                                           |
|---------------------------|---------------------------------------------------|
| <b>fib_table</b>          | A full routing table (main/local/custom)          |
| <b>fib_trie</b>           | The actual tree storing route prefixes            |
| <b>fib_info</b>           | A single routing entry (gateway, device, metrics) |
| <b>fib_result</b>         | The lookup answer returned for a packet           |
| <b>rtable / dst_entry</b> | Routing decision attached to the packet           |

If you want, I can also explain:

- fib\_info (very important)
- fib\_alias
- fib\_trie structure
- how routing rules select tables
- how multipath routing works
- how dst cache works

Just tell me!

Below is a **very detailed, beginner-friendly explanation** of ONLY the part you pasted — **fib\_scope**, **fib\_type**, **fib\_prefsrc**, and **fib\_priority**, along with what they mean in routing, where they are used, and realistic examples.

I'll treat each field like a "story" so you understand both the concept *and how the kernel uses it internally*.

## ★ 1. fib\_scope — “How far can this route or address be reached?”

Every IPv4 route in Linux is assigned a **scope**.

Scope tells the kernel:

- is this route only valid on this host?
- valid only on this link (LAN)?
- valid globally (internet)?
- or nowhere?

Think of it like the *visibility distance* of a route.

### ◊ Scope Categories

| Scope                                        | Meaning                                    | Example                                    |
|----------------------------------------------|--------------------------------------------|--------------------------------------------|
| <b>host</b> (RT_SCOPE_HOST)                  | Route is valid only for the local machine  | Loopback 127.0.0.1                         |
| <b>link</b> (RT_SCOPE_LINK)                  | Valid only for directly-attached neighbors | ARP neighbors, directly connected networks |
| <b>global / universe</b> (RT_SCOPE_UNIVERSE) | Valid everywhere; route is “real”          | Internet routes, gateway routes            |
| <b>site</b>                                  | Useless for IPv4; IPv6-only                | ignore for IPv4                            |
| <b>nowhere</b>                               | Route does not exist / invalid             | internal use                               |

## ★ How kernel decides scope when you add a route

If you do not specify scope, Linux assigns it automatically based on route type.

## 1. Gateawayed route (with via) → scope = GLOBAL

ip route add 8.8.8.8 via 192.168.1.1

This generates:

fib\_scope = RT\_SCOPE\_UNIVERSE

Because the route is meant to reach outside the local link.

## 2. Direct route (no gateway) → scope = LINK

Example:

ip addr add 192.168.1.10/24 dev eth0

Kernel automatically inserts routes like:

192.168.1.0/24 dev eth0 scope link

Meaning: neighbors on same LAN can reach this.

## 3. Local route → scope = HOST

Kernel automatically creates:

local 127.0.0.1 dev lo scope host

local 192.168.1.10 dev eth0 scope host

These routes never leave the machine.

## ★ Why does fib\_scope matter?

Because **routing lookup rejects routes whose scope is too small**.

For example:

- A route with scope link is never used to reach a far-away destination.
- A route with scope host can never be used to forward traffic.

This prevents illegal routing decisions.

## ★ 2. fib\_type — “What kind of route is this?”

This field classifies routes into categories:

| fib_type      | Meaning              |
|---------------|----------------------|
| RTN_UNICAST   | Normal unicast route |
| RTN_LOCAL     | Local address route  |
| RTN_BROADCAST | Broadcast route      |

|                 |                                   |
|-----------------|-----------------------------------|
| RTN_BLACKHOLE   | Drops packets                     |
| RTN_UNREACHABLE | Sends ICMP unreachable            |
| RTN_PROHIBIT    | Sends ICMP “Packet filtered”      |
| RTN_THROW       | Pass lookup to next routing table |
| RTN_NAT         | NAT route (rarely used)           |

### ◊ Why fib\_type exists

Originally, Linux stored the type in fib\_alias (older kernel versions).

But that was **not enough** because:

- Two routes could have same prefix/gateway but different behavior (e.g., prohibit, blackhole).
- Kernel needs this type at the actual fib\_info level.

So kernel 3.7 introduced fib\_type into fib\_info.

### ◊ Real Example: Prohibit Route

ip route add prohibit 192.168.1.17 from 192.168.2.103

Kernel creates a fib\_info with:

fib\_type = RTN\_PROHIBIT

When packets match this route:

- Kernel **does not forward them**
- It replies with:  
**ICMP: Packet filtered (ICMP\_PKT\_FILTERED)**

This is useful for implementing firewall-like behavior at routing level.

## ★ 3. fib\_prefsrc — Preferred Source Address

Sometimes, you want packets leaving through a specific route to use a **specific source IP**.

Example:

ip route add 10.0.0.0/24 via 192.168.1.1 src 192.168.1.55

The kernel stores:

fib\_prefsrc = 192.168.1.55

This means:

When sending a packet along this route, use 192.168.1.55 as the source address.

### Why is this needed?

Because a system may have **multiple IPs on the same interface**.

Example:

eth0 has:

192.168.1.10  
192.168.1.11  
192.168.1.12

And you want a certain traffic flow to always use .12 as source.

`fib_prefsrc` forces that choice.

## ★ 4. `fib_priority` — Route Priority (Metric)

Every route has a *priority* number.

This is used when:

- two routes match the same prefix
- kernel chooses one with BEST (lowest number) priority

### ◊ Important Rule

**Lower number = higher priority**

Example:

| Priority | Meaning  |
|----------|----------|
| 0        | Highest  |
| 5        | Lower    |
| 100      | Very low |

### ◊ Example Usage

All three commands below set the SAME field:

```
ip route add 192.168.1.0/24 via 10.0.0.1 metric 5
ip route add 192.168.1.0/24 via 10.0.0.1 priority 5
ip route add 192.168.1.0/24 via 10.0.0.1 preference 5
```

All result in:

`fib_priority` = 5

## ! A CRITICAL POINT (often misunderstood)

The **metric** you set via `ip route add metric X`

is stored in **fib\_priority**, NOT in `fib_metrics`.

The `fib_metrics` array stores TCP/MTU metrics, like:

- MTU
- RTT
- MSS
- cwnd
- window

But NOT the route's sorting priority.

## ★ Putting it all together — A Real Example

Imagine this command:

```
ip route add 8.8.8.8 via 192.168.1.1 dev eth0 metric 20 src 192.168.1.55 prohibit
```

The inside fib\_info fields become:

| Field                                      | Meaning                |
|--------------------------------------------|------------------------|
| <code>fib_scope = RT_SCOPE_UNIVERSE</code> | because it's gatewayed |
| <code>fib_type = RTN_PROHIBIT</code>       | prohibit route         |
| <code>fib_prefsrc = 192.168.1.55</code>    | preferred source       |
| <code>fib_priority = 20</code>             | metric = 20            |

This route:

- is reachable everywhere,
- but is prohibited from use,
- and forces packets to use source 192.168.1.55,
- AND has low priority (20 > lower than 0).

## ★ If you want next:

I can continue with:

- ✓ How fib\_info + fib\_alias + fib\_table work together
- ✓ Diagram of the routing lookup algorithm
- ✓ Example: step-by-step lookup of a packet

Just tell me!

Great — let's build this **slowly and clearly** from the ground up.

Before explaining *how* fib\_info, fib\_alias, and fib\_table work together, you must first understand **what each of them is**, what problem each one solves, and why all three are needed.

I'll explain each structure **like you are reading a story of how Linux stores routes**.

## ★ PART 1 — WHAT EACH STRUCTURE IS (BEGINNER-FRIENDLY + DEEP)

We start with the **big picture**:

`fib_table` → `fib_alias` → `fib_info`

Think of it like:

- `fib_table` = a *phonebook*
- `fib_alias` = entries inside the phonebook, grouped by prefix
- `fib_info` = the *actual details* of a route (gateway, interface, metrics)

Let's go one by one.

## 1 fib\_table — The Routing Table

`fib_table` is the **entire routing table** (main, local, custom).

Each routing table has:

- a **prefix trie** storing routes
- an ID (254 = main, 255 = local)
- counters (like number of default routes)
- pointer to the trie structure

A `fib_table` contains **all routes for that namespace**.

☞ Simple analogy:

`fib_table` = a **dictionary** where keys are prefixes (e.g., “192.168.1.0/24”).

☞ What does `fib_table` actually store?

- It does **not** store route details directly.
- It stores **fib\_alias objects**, grouped by prefix.

## 2 fib\_alias — Routes grouped under a prefix entry

A `fib_alias` represents a **routing rule for a specific prefix**.

Example:

192.168.1.0/24 dev eth0

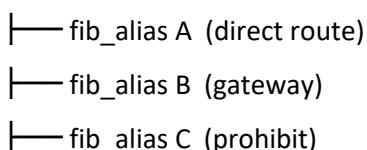
192.168.1.0/24 via 192.168.1.1

192.168.1.0/24 prohibit

All these three belong to the *same prefix* 192.168.1.0/24.

So Linux stores:

prefix (192.168.1.0/24)



Each `fib_alias` contains:

- the type of route (unicast, blackhole, prohibit, etc.)

- priority (metric)
- references to fib\_info
- nexthop count
- scope

### Why fib\_alias exists?

Because **multiple routes can exist for the same network prefix**.

But all of them **share** the same:

- destination network
- prefix length

So Linux groups them using fib\_alias.

## **3** fib\_info — Actual routing data (gateway, device, metrics, nexthops)

fib\_info is the **real route object**, containing:

- gateway (fib\_nh[].nh\_gw)
- output device (fib\_nh[].nh\_dev)
- metrics (MTU, RTT, MSS)
- preferred source address
- route type (unicast, prohibit, blackhole)
- protocol (static, kernel, DHCP)
- number of nexthops
- nexthops themselves

### Important:

Many fib\_alias entries may **share the same fib\_info** if they use the same nexthop/metrics.

This prevents duplication.

## SUMMARY OF WHAT THEY ARE

| Structure        | Purpose                                                  | Level     |
|------------------|----------------------------------------------------------|-----------|
| <b>fib_table</b> | Stores all routes in a routing table                     | Top-level |
| <b>fib_alias</b> | Stores routes under same prefix, sorted by priority/type | Middle    |
| <b>fib_info</b>  | Actual routing entry: gateway, device, metrics, nexthops | Bottom    |

## PART 2 — HOW THEY WORK TOGETHER (FLOW EXPLANATION)

Let's walk through a real example to see how they connect.

You add this route:

```
ip route add 192.168.1.0/24 via 10.0.0.1 dev eth0 metric 10
```

### Step 1 — Kernel finds the correct fib\_table

Since you didn't specify a table, it uses:

```
tbl = fib_get_table(net, RT_TABLE_MAIN)
```

This is your **main routing table** (table ID 254).

## 🔧 Step 2 — Kernel locates the prefix in the trie

Lookup:

```
prefix = 192.168.1.0, prefix_len = 24
```

If no entry exists, kernel creates a new node for that prefix.

## 🔧 Step 3 — Kernel creates fib\_info

Kernel fills out:

```
fib_info {  
    nh_gw = 10.0.0.1  
    nh_dev = eth0  
    fib_scope = link/universe  
    fib_type = RTN_UNICAST  
    fib_priority = 10  
    fib_metrics = default metrics  
}
```

This structure contains **all route details**.

## 🔧 Step 4 — Kernel creates a fib\_alias pointing to that fib\_info

It creates:

```
fib_alias A {  
    fa_type = RTN_UNICAST  
    fa_scope = RT_SCOPE_UNIVERSE  
    fa_info = &fib_info  
    fa_priority = 10  
}
```

And attaches it to the prefix node under:

```
fib_table → trie → prefix node → list of fib_alias entries
```

## Step 5 — Kernel sorts multiple fib\_alias entries by priority/type

If you add this too:

```
ip route add 192.168.1.0/24 dev eth0 metric 0
```

Now the list for 192.168.1.0/24 becomes:

fib\_alias #1 (metric 0)

fib\_alias #2 (metric 10)

Sorted by:

- priority
- type
- scope

## PART 3 — HOW LOOKUP WORKS USING ALL THREE

When a packet for 192.168.1.5 arrives:

### ◊ Step 1: fib\_table

Main routing table is chosen.

### ◊ Step 2: prefix trie lookup → reach prefix node (192.168.1.0/24)

### ◊ Step 3: iterate fib\_alias list

Kernel checks each alias:

12. highest priority match
13. scope valid for packet
14. type is unicast
15. route usable (not dead, not prohibited)

### ◊ Step 4: kernel selects fib\_info

The chosen fib\_alias points to a fib\_info.

That fib\_info tells kernel:

- which interface to use
- what gateway to forward to
- what MTU / metrics to apply

And with that...

**packet is forwarded.**

## PART 4 — VISUAL DIAGRAM

Here is a clear visual of their relationship:

fib\_table (main table 254)

|

  |— prefix trie

```

|   └── prefix: 192.168.1.0/24
|       ├── fib_alias #1 (metric 0)
|       |   └── fib_info A (dev eth0)
|       ├── fib_alias #2 (metric 10)
|       |   └── fib_info B (via 10.0.0.1 dev eth1)
|       └── fib_alias #3 (prohibit)
|           └── fib_info C (type RTN_PROHIBIT)
|
└── other prefixes...

```

Notice:

- multiple **fib\_alias** entries under same prefix
- each fib\_alias may point to a **different fib\_info**
- OR multiple aliases may share the same fib\_info

## PART 5 — WHY ALL THREE STRUCTURES ARE NEEDED

| Need                                        | Structure solving it          |
|---------------------------------------------|-------------------------------|
| Group routes by prefix                      | <code>fib_table</code>        |
| Multiple routes per prefix (policy, metric) | <code>fib_alias</code>        |
| Store real nexthop, device, metrics         | <code>fib_info</code>         |
| Share route data between aliases            | <code>fib_info</code>         |
| Fast longest-prefix lookup                  | <code>fib_table's trie</code> |
| Route preference sorting                    | <code>fib_alias list</code>   |

All three solve different core problems of routing.

## NEXT WHAT YOU CAN ASK:

- Explain **fib\_alias** fields deeply with examples
- Draw packet lookup step-by-step with kernel function names
- Explain **fib\_info** lifetime (refcount, RCU, destruction)
- Show code paths: `ip route add` → `fib_table_insert` → `fib_create_info`

Tell me what you want next, and I'll continue in the same deep detail.

Alright — here is the **complete, deep, step-by-step lookup flow** for IPv4 routing in Linux, including:

- packet arrival point
- kernel functions executed
- how **fib\_table → fib\_alias → fib\_info → rtable/dst** are used
- decisions made at each level
- all the important structures involved

And I will draw it in a clear **layered “map” diagram**, so you can visualize the logic exactly the way it happens in the kernel.

This is designed to be **beginner friendly but extremely detailed**, exactly the style you want.

## COMPLETE PACKET ROUTING LOOKUP FLOW (IPv4)

This is the path for **any packet that needs routing**:

- packets generated locally (TCP, UDP, ICMP)
- packets arriving from NIC and going to forward path

I will show:

- functions at each step
- structures used
- what decisions are made
- how `fib_table`, `fib_alias`, `fib_info`, and `dst_entry` connect

## TOP-LEVEL FLOW (Summary Before Deep Dive)

Packet arrives



`ip_route_input_slow() / ip_route_output_flow()`



`fib_lookup() <-- prefix trie lookup`



`fib_table_lookup() <-- in main routing table`

```
↓  
prefix node  
↓  
fib_alias list sorted by priority/type  
↓  
choose correct fib_alias  
↓  
fib_info (gateway, device, metrics)  
↓  
build rtable (dst_entry)  
↓  
use rtable for actual forwarding
```

Now let's break this down fully.

## 🔗 STEP 1 — Packet Enters Routing Subsystem

For incoming packets:

```
ip_recv()  
↓  
ip_route_input()  
↓  
ip_route_input_slow()
```

For locally-generated packets:

```
ip_queue_xmit()  
↓  
ip_route_output_flow()
```

At this point, the kernel wants to know:

**“What is the route for this destination IP?”**

So it calls the routing lookup code.

## 🔗 STEP 2 — Route Lookup Starts

Both input and output routing call the same core function:

```
fib_lookup()
```

This is the heart of IPv4 routing.

`fib_lookup()` needs:

- destination IP
- source IP
- TOS
- incoming device (for input)
- scope

It returns a structure called:

`struct fib_result`

This contains:

- pointer to `fib_info`
- pointer to `fib_alias`
- prefix length matched
- type, scope
- flags



## STEP 3 — `fib_lookup()` enters a `fib_table`

Linux normally uses:

- **main table → 254**
- **local table → 255**

Function:

`fib_table_lookup()`

This function performs:

- longest-prefix match
- search inside trie
- selection of the correct alias



## STEP 4 — Prefix Trie Search (Level: `fib_table`)

Each `fib_table` has a **trie** of prefixes (LPM).

Search is done in:

`struct trie` (`net/ipv4/fib_trie.c`)

The trie node represents:

prefix (e.g., 192.168.1.0/24)

has list of `fib_alias` objects



## STEP 5 — `fib_alias` Selection (Level: `fib_alias`)

Once kernel finds the prefix node:

`fib_alias` list

```
|  
| sorted by:  
| → priority (metric)  
| → route type  
| → scope
```

Function used:

```
fib_select_default()  
fib_select_path()
```

The kernel iterates over the aliases and selects the correct one whose:

- type is valid (unicast, local, reject, prohibit)
- scope is acceptable
- metrics have the highest priority
- constraints match (source routing, TOS)

Each fib\_alias contains:

- route type
- priority (metric)
- scope
- nexthop count
- pointer to **fib\_info**

**fib\_alias DOES NOT contain gateway or device.**

It only acts as:

→ a selector that chooses **which fib\_info** is used

## ⌚ STEP 6 — fib\_info (Level: actual route details)

The selected alias points to:

```
fib_result.fi ==> struct fib_info
```

fib\_info contains:

- all nexthops (fib\_nh[])
- gateway (nh\_gw)
- device (nh\_dev)
- metrics (fib\_metrics)
- type, scope
- preferred source
- route protocol
- flags
- multipath info

**This is the real routing entry.**

Everything needed to build the forwarding state comes from here.

## STEP 7 — nexthop Selection (if multipath)

If route is:

nexthop via X

nexthop via Y

nexthop via Z

Kernel uses:

`fib_select_multipath()`

It selects:

- the best nexthop
- based on hashing of source+destination

Selected nexthop is:

`fib_nh chosen_nh;`

Contains:

- gateway IP
- output device
- L2 header cache
- flags
- weights

## STEP 8 — Build rtable (dst\_entry)

Once nexthop is selected, kernel constructs:

`struct rtable` (a type of `dst_entry`)

Function:

`rt_dst_alloc()`

Content of rtable includes:

- output interface (`rt->dst.dev`)
- MTU (from `fib_info metrics`)
- gateway (`rt_gateway`)
- path MTU
- metrics
- TTL
- neighbour pointer (ARP entry)
- input/output function pointers

This **rtable** is what the IP stack actually uses to forward packets.

This is stored in the route cache.

## STEP 9 — Return rtable to caller

Finally:

- ip\_route\_input() or ip\_route\_output\_flow()
- receive the **rtable**
- attach it to the socket or SKB
- forwarding begins using:  
dst\_output()

Which eventually calls:

neigh\_output()

dev\_queue\_xmit()

And packet is sent.

## COMPLETE STRUCTURE CHAIN DIAGRAM

Here is the **full chain**, level by level:

PACKET



(1) Routing Decision Layer

---

ip\_route\_input\_slow() / ip\_route\_output\_flow()



fib\_lookup()

---

(2) FIB Table Layer

---

fib\_table (main, local, etc.)



trie lookup (longest prefix match)



prefix node found

---

(3) FIB Alias Layer

---

prefix node contains:

```
fib_alias #1 ← metric 0  
fib_alias #2 ← metric 100  
fib_alias #3 ← prohibit
```

↓

kernel picks correct fib\_alias

↓

points to → fib\_info

---

#### (4) FIB Info Layer (Real Route Data)

---

```
fib_info {  
    nexthops[]  
    gateway  
    device  
    metrics  
    type  
    scope  
    prefsrc  
}
```

↓

select nexthop (if multipath)

---

#### (5) rtable (dst\_entry)

---

```
rtable {  
    dst.dev → output device  
    gateway → next hop  
    metrics → MTU, RTT, MSS  
    neighbour → ARP  
}
```

↓

used for actual forwarding



| Level               | Structure | Purpose                    |
|---------------------|-----------|----------------------------|
| Route database      | fib_table | Stores all prefixes        |
| Routes under prefix | fib_alias | Sort and choose best route |
| Actual route        | fib_info  | Gateway, device, metrics   |
| Forwarding state    | rtable    | Used to forward packet     |

Everything flows in this direction:

fib\_table → fib\_alias → fib\_info → rtable

## ✓ If you want next:

I can also draw:

- ✓ The reverse map: how ip route add builds all these internally
- ✓ A full diagram of memory layout
- ✓ Detailed explanation of fib\_result
- ✓ All major kernel functions in exact sequence

Just tell me which one and I'll make it in the same style.

Below is a **clean, deeply detailed, step-by-step explanation** of the exact paragraph you pasted — focusing **only** on the fields and behavior that appear in that section (fib\_metrics, fib\_nhs, fib\_dev, fib\_nh[], fib\_props, RTN\_PROHIBIT, etc.).

I'll first explain **what these fields are**, then **why they exist**, then **how they act inside a real lookup**, and finally **give a clear worked example in plain English**.

## ✓ 1. fib\_info – What this structure represents

A **fib\_info** represents **one routing entry** after all attributes are processed.

Think of it like:

Destination prefix + attributes → fib\_info

A fib\_info contains:

- nexthop(s)
- metrics
- route type
- preferred source
- priority
- route scope
- pointer to the device

It does **not** store the prefix itself.

The *prefix* (like 192.168.1.0/24) is stored in **fib\_alias / TRIE**.

The attributes (MTU, type, nexthop, metrics etc.) are stored in **fib\_info**.

**fib\_alias → fib\_info → nexthop/device/metrics/etc.**

## 2. fib\_metrics – why this exists

fib\_metrics[] is an array containing 15 routing metrics (RTAX\_MAX entries).

These metrics describe path characteristics, e.g.:

- MTU
- RTT
- Window
- slow-start settings
- hop penalties
- initcwnd
- initrwnd
- etc.

The kernel uses them for TCP, PMTU, decisions about path quality.

Examples:

### MTU

User sets route MTU:

```
ip route add 192.168.1.0/24 mtu 800
```

fib\_info stores:

```
fib_metrics[RTAX_MTU] = 800
```

### initcwnd

```
ip route add 192.168.1.0/24 initcwnd 35
```

fib\_info gets:

```
fib_metrics[RTAX_INITCWND] = 35
```

### Why some metrics use “lock”

```
ip route add 192.168.1.0/24 mtu lock 800
```

Lock here means:

- Path MTU Discovery **cannot** reduce MTU dynamically.
- Kernel checks:  

```
if (dst_metric_locked(dst, RTAX_MTU))  
    return;
```

So metrics affect:

- congestion control
- TCP window sizing
- path MTU discovery
- PMTU update logic

### 3. fib\_nhs / fib\_nh[] – nexthop count

fib\_nhs = number of nexthops for this route.

Without multipath (default):

fib\_nhs = 1

fib\_nh[0] = single nexthop

With multipath routing:

```
ip route add default \
```

```
    nexthop dev eth0 \
```

```
    nexthop dev eth1
```

Then fib\_info looks like:

fib\_nhs = 2

fib\_nh[0] = (dev=eth0)

fib\_nh[1] = (dev=eth1)

Kernel will:

- load balance
- failover
- choose nexthop based on weight or hashing

### 4. fib\_dev – which device will send the packet

fib\_dev is the **output network device** determined by kernel based on:

- nexthop gateway and its L2 resolution
- multipath rules
- interface matching rules

Examples:

Directly connected route

```
ip route add 192.168.1.0/24 dev eth0
```

fib\_dev = eth0

Gateway route

```
ip route add default via 10.1.1.1 dev eth1
```

fib\_dev = eth1

Nexthop = 10.1.1.1

### 5. fib\_type & fib\_props – route behaviour

Each route has a **type**, e.g.:

- RTN\_UNICAST
- RTN\_BLACKHOLE
- RTN\_PROHIBIT
- RTN\_LOCAL
- RTN\_THROW
- RTN\_UNREACHABLE

These route types determine:

- how kernel treats traffic
- what ICMP error to send
- what is the scope

All these properties are stored in the array:

`fib_props[RTN_TYPE]`

Each entry has:

```
struct fib_prop {
    int error; // return error (-EACCES, -ENETUNREACH, etc)
    u8 scope; // RT_SCOPE_UNIVERSE, LINK, HOST, etc.
};
```

Example:

For RTN\_PROHIBIT:

```
fib_props[RTN_PROHIBIT] = {
    .error = -EACCES,
    .scope = RT_SCOPE_UNIVERSE
}
```

Meaning:

- lookup returns -EACCES
- kernel later sends ICMP\_PKT\_FILTERED



## 6. Detailed Walkthrough of the “prohibit” example

User adds a prohibit rule:

`ip route add prohibit 192.168.1.17 from 192.168.2.103`

This creates:

- fib\_alias node in trie → matching 192.168.1.17
- fib\_info with:
  - fib\_type = RTN\_PROHIBIT

Packet arrives...

- Source = 192.168.2.103
- Destination = 192.168.1.17

Kernel performs:

Step 1 → fib\_lookup()

Search TRIE for a matching leaf.

Step 2 → check\_leaf()

During leaf evaluation:

```
err = fib_props[fa->fa_type].error;
```

Since type = RTN\_PROHIBIT:

```
err = -EACCES
```

Step 3 → fib\_lookup() returns -EACCES up to IP input code.

Step 4 → ip\_error()

Sends:

ICMP Destination Unreachable

Code: Packet Filtered (ICMP\_PKT\_FILTERED)

Packet is dropped.

## 7. Final Summary (Clean & Clear)

| Component            | Purpose                                                                      |
|----------------------|------------------------------------------------------------------------------|
| <b>fib_table</b>     | Routing table (TRIE root + IDs)                                              |
| <b>fib_alias</b>     | Prefix node in TRIE (stores prefix, list of fib_info)                        |
| <b>fib_info</b>      | Route attributes (gateway, device, metrics, type, priority, prefs, nexthops) |
| <b>fib_metrics[]</b> | MTU, RTT, TCP metrics, PMTU behaviour                                        |
| <b>fib_nh[]</b>      | One or more nexthops (direct or via gateway)                                 |
| <b>fib_dev</b>       | Output device                                                                |
| <b>fib_type</b>      | Behavior of route (unicast, blackhole, prohibit, unreachable)                |
| <b>fib_props</b>     | Maps type → error + scope                                                    |

Together, they form the core of the IPv4 Routing Subsystem.

## If you want...

I can also draw:

- A complete diagram showing fib\_table → TRIE → fib\_alias → fib\_info → fib\_nh → device
- Full flow of a packet lookup from IP layer to output device
- L3 routing subsystem function call map
- Route insertion path (rtmsg → fib\_table\_insert → fib\_create\_info)

Just tell me “draw the diagram” or “show call flow”.



# CACHING

Thursday, December 11, 2025 10:00 AM

Below is the complete, clear, beginner-friendly explanation of the **caching mechanism** inside the IPv4 routing subsystem exactly for the text you posted.

I'll explain:

1. **Why caching is needed**
2. **What exactly is being cached**
3. **Where it is stored (fib\_nh → nh\_rth\_input / nh\_pcpu\_rth\_output)**
4. **How caching works in RX and TX paths**
5. **When caching is NOT done**
6. **How PMTU + Redirects are cached via FIB Exceptions**
7. **Where rtable fits**

## ✳️ 1. Why Caching?

Routing lookup is expensive:

- It must walk the FIB Trie
- Compare longest prefix
- Fetch route type, nexthops, metrics
- Apply rules and policy routing

Doing this **for every packet** is slow.

→ So kernel caches the **result of the routing lookup**.

## ✳️ 2. What exactly is cached?

**Not the full fib\_info, not the fib\_table** — instead:

The kernel caches a small lightweight structure:

⌚ **struct rtable** (a type of dst\_entry)

It contains:

- nexthop device
- gateway
- MTU
- metrics
- input/output functions
- link-layer info

It's the final “resolved route” that forwarding needs.

This is the structure the kernel uses for:

✈️ packet output

🌐 packet input

📦 TCP PMTU

↑ redirect handling

## 3. Where is this rtable cached?

Inside **fib\_nh**, the nexthop structure inside **fib\_info**.

There are two caches:

### 1. **nh\_rth\_input**

→ Cached route for **RX path**

### 2. **nh\_pcpu\_rth\_output**

→ Cached route for **TX path**

Why two?

RX and TX logic differ, so each has its own rtable.

Also, nh\_pcpu\_rth\_output is:

#### ✓ **per-CPU**

✓ so each CPU has its own cached route

✓ avoids lock contention

## 4. How caching occurs (RX vs TX)

Caching is done by the function:

`rt_cache_route()`

(net/ipv4/route.c)

### A. RX Path Caching

Incoming packet:

8. `fib_lookup()` finds a `fib_result`
9. From `fib_result`, the kernel identifies nexthop: `fib_nh`
10. Kernel creates rtable
11. Stores it into:  
`fib_nh->nh_rth_input`

Next time an identical packet arrives:

- kernel **skips the fib trie lookup**
- reuses cached rtable

### B. TX Path Caching

Outgoing packet:

12. `fib_lookup()` again finds the matching `fib_info`
13. Kernel selects nexthop `fib_nh`
14. Kernel creates rtable for **this CPU**
15. Stores it into:

```
fib_nh->nh_pcpu_rth_output[cpu]
```

This is per CPU → ultra-fast.

## ✿ 5. When caching is NOT done

Caching is **not** performed when:

✗ Packet is NOT unicast

(broadcast, multicast → behave differently)

✗ Realms (tclassid) are used

If policy routing uses traffic class → caching breaks correctness.

✗ Error or ICMP redirect case

Redirect requires special exceptions.

✗ Not enough memory

Otherwise caching is **almost always used**.

## ✿ 6. What about PMTU and Redirect caching?

This is NOT stored in nh\_rth\_input or nh\_pcpu\_rth\_output.

It is stored in **FIB Exceptions**.

Each nexthop (fib\_nh) has:

```
struct fib_nh_exception *nh_exceptions;
```

Used for:

❖ Path MTU Discovery

❖ ICMP Redirects

These store:

- updated MTU for a specific path
- updated gateway from redirect
- special case rtable entries

This avoids polluting the main route table.

## ✿ 7. How it all fits together (full picture)

```
fib_table
```

```
|
```

```
| lookup (prefix)
```

```
v
```

```
fib_alias (leaf)
```

```

|
| references
v
fib_info (one route entry)
|
| contains
v
fib_nh (nexthop)
|
+-----+
| CACHING DONE HERE          |
|                               |
| nh_rth_input --> cached rtable (RX)
| nh_pcpu_rth_output[cpu] ---> cached rtable (TX)
+-----+

```

Plus:

`fib_nh->nh_exceptions --> PMTU / Redirect Exceptions`

This is the entire caching subsystem.

## ★ Summary in One Line

The kernel stores fast-path routing decisions (rtable objects) inside `fib_nh` to avoid repeated `fib_lookup` calls — using one cache for RX, one per-CPU for TX, and separate exception tables for PMTU and redirects.

If you want, I can next explain:

- `fib_nh` structure in deep detail
- rtable structure and how it drives forwarding
- How `fib_info + fib_nh + rtable` integrate into full routing pipeline

Just tell me “**explain fib\_nh**” or “explain rtable next.”

Below is **the most beginner-friendly explanation of “realms” and “itags”** in Linux routing — what they are, where they come from, and why the kernel disables nexthop caching when they’re used.

## 1. What Are “Realms”?

(*Userspace name: realms / rt\_realms — Kernel name: tclassid / itag*)

**Realms** are **administrative tags** (labels) that you can attach to routes or packets.

They let the administrator **classify packets** as they move through the routing subsystem.

Think of realms as **tags used for policy routing**, traffic engineering, or accounting.

### Where do realms come from?

You set realms from userspace:

```
ip route add 10.1.0.0/16 dev eth0 realm blue
```

```
ip rule add fwmark 0x10 lookup 100 realm green
```

Realms are simply **unsigned 32-bit integers** divided into:

```
realm = (source_realm << 16) | destination_realm
```

### Kernel: This is stored in skb->tclassid

Inside the kernel, this tag is stored in:

```
skb->tclassid
```

This variable is also called **itag** (“interface tag”).

## Why Realms Exist

They allow you to:

### Classify traffic for monitoring

Example: count traffic from different customers

### Mark packets from different security zones

Example: firewall zones

### Influence advanced routing rules

Example: bind realms to route tables

They do **NOT** affect normal routing.

They simply carry tags to help admins apply special rules/policies.

## 2. What Are “itags”?

In the book, “itags” refers to:

```
skb->tclassid
```

This is the **in-kernel representation of realms**, inherited from tc/classifier framework.

Whenever a packet has a realm assigned, the kernel internally stores it in this "tclassid" tag.

## ⚠️ VERY IMPORTANT:

When itag != 0, the kernel **disables nexthop caching**.

From your text:

when the packet itag is not 0, the results are not cached.

Why?

🔴 Because caching per-realm would mix up tagged flows

If route A and route B both use the same nexthop but different realms:

| Packet | Realm | Expected Behavior         |
|--------|-------|---------------------------|
| pkt1   | blue  | match rule for realm blue |
| pkt2   | red   | match rule for realm red  |

If the nexthop cached the result:

✓ pkt1 cached

✗ pkt2 would incorrectly reuse pkt1's cached result

Therefore:

☞ If realms/tags are used → kernel **MUST NOT** cache route lookup result.

## 📝 Summary: Realms & itags in One Shot

| Term                 | Meaning                                                             |
|----------------------|---------------------------------------------------------------------|
| Realms               | User-space names for per-route or per-packet classification labels. |
| itag                 | Kernel's internal "tag" stored in skb->tclassid.                    |
| Purpose              | Traffic classification, QoS, monitoring, policy routing.            |
| Effect               | When present (itag != 0), disables nexthop caching.                 |
| Why disable caching? | To avoid mixing routing decisions of packets with different realms. |

## ✳️ If You Want Next

I can draw a **full routing flow showing**:

**fib\_table → fib\_alias → fib\_info → fib\_nh → dst caching → realms → itag → where caching gets disabled**

using a very simple diagram that fits in your mental model.

Just say "**show me the diagram**".

Below is a **simple + detailed** explanation of the **nexthop (fib\_nh)** structure, what it represents, why it exists, how it is used inside routing, and what happens when interfaces go down — all explained like a teacher.

## ★ WHAT IS A NEXTHOP (fib\_nh)? – SIMPLE DEFINITION

A **nexthop** is the **actual path** a packet takes when leaving the host.

Every route ultimately boils down to:

- ❖ Which interface to use? (eth0? eth1?)
- ❖ Which gateway to use? (192.168.1.1?)
- ❖ Which source IP to use?
- ❖ Is the nexthop alive/dead?
- ❖ What exceptions exist? (PMTU exceptions, redirect exceptions)

Linux stores these nexthop details inside:

### ► struct fib\_nh

And the parent route (fib\_info) may contain **one or more nexthops** (for multipath routing).

## ★ WHY DOES Linux NEED fib\_nh?

Because a **route** may have:

- ✓ One nexthop
- ✓ Multiple nexthops (ECMP, multipath routing: 50% via eth0, 50% via eth1)

Therefore, the actual routing entry (fib\_info) only **describes the route**, while fib\_nh **describes the physical next hop**.

Example:

```
ip route add default \
    nexthop via 10.0.0.1 dev eth0 \
    nexthop via 10.0.0.2 dev eth1
```

Here:

- fib\_info = one route entry
- fib\_info.fib\_nhs = 2
- fib\_info.fib\_nh[0] and fib\_info.fib\_nh[1] = two nexthops

## ★ STRUCTURE BREAKDOWN (field-by-field explained clearly)

Here is the struct again:

```
struct fib_nh {
```

```

struct net_device *nh_dev;
struct hlist_node nh_hash;
struct fib_info *nh_parent;
unsigned int nh_flags;
unsigned char nh_scope;

int nh_oif;
__be32 nh_gw;
__be32 nh_saddr;
int nh_saddr_genid;

struct rtable __rcu * __percpu *nh_percpu_rth_output;
struct rtable __rcu *nh_rth_input;

struct fnhe_hash_bucket *nh_exceptions;
};


```

Now let's explain **every element** in clear, simple language.

## ◊ 1. nh\_dev — Outgoing network interface

This is the **net\_device** pointer for the nexthop.

Examples:

- eth0
- eth1
- wlan0
- bond0

 **Meaning:**

*Which interface should transmit packets that use this nexthop?*

## ◊ 2. nh\_hash — Nexthop hash table entry

Used to store this nexthop in an internal hash table for fast lookup.

 Mostly internal book-keeping.

## ◊ 3. nh\_parent — Back pointer to parent **fib\_info**

Every nexthop belongs to a route (**fib\_info**).

 This field allows the kernel to go back from nexthop → route.

## ◊ 4. nh\_flags — Nexthop state flags

Example flags:

- RTNH\_F\_DEAD → nexthop unusable (interface down)
- RTNH\_F\_ONLINK → gateway is directly reachable

❖ Used to disable / reject nexthop when needed.

## ◊ 5. nh\_scope — Route scope for this nexthop

Same meaning as fib\_scope:

- RT\_SCOPE\_LINK
- RT\_SCOPE\_UNIVERSE
- RT\_SCOPE\_HOST

❖ Defines reachability: local / link / global.

## ◊ 6. nh\_oif — output interface index

Interface index (ifindex), e.g.:

- 2 for eth0
- 3 for eth1

❖ Used by routing engine for fast processing.

## ◊ 7. nh\_gw — Gateway IP (next-hop router)

Example:

nh\_gw = 192.168.1.1

If the route is direct (no gateway), nh\_gw = 0.

❖ This is the IP router we send packets to.

## ◊ 8. nh\_saddr — Source IP to use

If route specifies:

ip route add 10.0.0.0/24 src 10.1.1.5

Then:

nh\_saddr = 10.1.1.5

❖ Helps choose source address automatically.

## ◊ 9. nh\_saddr\_genid — generation ID for source address

Used when source addresses change dynamically (like with DHCP).

❖ Ensures kernel recomputes source address when needed.

## ◊ 10. nh\_pcpu\_rth\_output — Per-CPU output rtable cache

This is:

- ✓ A cache
- ✓ One per CPU
- ✓ For output (Tx path)
- ❖ **Improves performance** by avoiding repeated route lookups.

## ◊ 11. nh\_rth\_input — Cached rtable for input

Rx path (receiving packets) caching.

❖ Same idea: **avoid expensive routing lookups**.

## ◊ 12. nh\_exceptions — FIB nexthop exceptions

Contains:

- PMTU exceptions (Path MTU Discovery)
- ICMP Redirect exceptions

Example: When MTU on path to gateway changes.

❖ **Overrides route behavior for special cases**.

## ★ NOW — How Linux reacts when the outgoing interface goes DOWN

When a device (say eth0) goes down:

Linux triggers a **NETDEV\_DOWN** event.

Routing subsystem handles it using:

① fib\_netdev\_event() (registered in notifier chain)

→ Triggered when interface goes down.

② Calls fib\_disable\_ip():

Inside this:

- ✓ RTNH\_F\_DEAD is set
- ✓ All fib\_info entries using this nexthop are marked

- ✓ Routing cache is flushed
- ✓ ARP entries for that interface removed (arp\_ifdown())

☞ This ensures the kernel does NOT continue sending packets through a dead interface.

## ★ Putting It All Together — A Simple Example

Command:

```
ip route add 192.168.100.0/24 via 10.0.0.1 dev eth0
```

Kernel Internally:

16. **fib\_info** is created for route 192.168.100.0/24
17. **fib\_info.fib\_nhs = 1**
18. Kernel creates **fib\_nh[0]**
19. Populates:
  - nh\_dev = eth0
  - nh\_gw = 10.0.0.1
  - nh\_oif = ifindex(eth0)
  - nh\_saddr = auto-selected
20. Creates route caching entries:
  - nh\_pcpu\_rth\_output
  - nh\_rth\_input
21. Inserts into FIB Trie
22. Uses it during routing lookup

## ★ VERY SIMPLE SUMMARY

**fib\_nh** = the actual nexthop used for routing.

It stores:

| Field      | Meaning                          |
|------------|----------------------------------|
| nh_dev     | which interface sends the packet |
| nh_gw      | gateway address                  |
| nh_oif     | interface index                  |
| nh_saddr   | source IP                        |
| nh_flags   | alive/dead flags                 |
| caches     | fast-path route caches           |
| exceptions | PMTU redirects                   |

And it's used **per nexthop**, not per route.

If you want, I can now draw:

- ✓ Diagram of FIB system: fib\_table → fib\_alias → fib\_info → fib\_nh → dst\_entry
- ✓ Routing lookup flow (step-by-step)
- ✓ Packet forwarding through nexthop
- ✓ Multipath routing example (multiple fib\_nh)

Tell me what to draw next.

Sure — I will **explain this entire section in the simplest possible way**, while keeping it *accurate and detailed*.

We will focus on:

## ★ FIB Nexthop Exceptions — Simple but Detailed Explanation

### ► Why “FIB Nexthop Exceptions” exist?

Normally, the kernel routing table is static unless:

- ✓ Admin changes it (ip route add/delete)
- ✓ Routing daemon changes it (BGP, OSPF, FRR)

But **two things can change routing dynamically**, from unpredictable network events:

### 1 ICMP Redirects

When a router tells the host:

“Hey, you are using the wrong gateway. Use this new gateway instead.”

The kernel must update the routing entry **temporarily**.

### 2 Path MTU Discovery (PMTU)

When a router says:

“Packet too big — use a smaller MTU.”

The kernel must temporarily store the new MTU **only for that destination**.

These modifications **must not change the global routing table**, because:

- They are temporary
- They apply only for one destination
- They are dynamic
- They should be removed after some time

➲ So the kernel stores these changes in a **per-nexthop exception table**.

This is the **FIB nexthop exceptions mechanism**.

### ★ Where are nexthop exceptions stored?

Inside each **fib\_nh** (nexthop object), there is:

```
struct fnhe_hash_bucket *nh_exceptions;
```

This is a **hash table of exceptions**, keyed by:

- ✓ destination IP address

## ★ Structure of one nexthop exception

```
struct fib_nh_exception {  
    struct fib_nh_exception *fnhe_next;  
    __be32    fnhe_daddr; // destination  
    u32      fnhe_pmtu; // new MTU (PMTU discovery)  
    __be32    fnhe_gw;   // new gateway (ICMP redirect)  
    unsigned long fnhe_expires; // expiry time  
    struct rtable *fnhe_rth; // cached route  
    unsigned long fnhe_stamp;  
};
```

Let's simplify each field.

## ★ Simple Meaning of Each Field

- ◊ `fnhe_daddr`

The destination IP that this exception applies to.

Example: "For packets to 8.8.8.8, use new MTU 1400."

- ◊ `fnhe_pmtu`

The new MTU discovered by PMTU.

Stored when router sends **ICMP Frag Needed** messages.

- ◊ `fnhe_gw`

New gateway discovered via **ICMP Redirect**.

Example:

Host → Router A → Router B

Router A says: "Use Router B directly."

Kernel stores new GW in exception.

- ◊ `fnhe_expires`

Exception timeout (default 10 minutes).

- ◊ **fnhe\_rth**

A cached rtable object for this exception.

Improves performance because lookup is skipped next time.

- ◊ **fnhe\_stamp**

Used for aging/refreshing the entry.

## ★ When and How are Exceptions Created?

- ◊ Case 1 — ICMP Redirect

Function:

`__ip_do_redirect()`

Router tells host:

“Use this new gateway!”

Kernel calls:

`update_or_create_fnhe()`

This sets:

`fnhe_gw = <new gateway>`

- ◊ Case 2 — Path MTU Discovery

Function:

`__ip_rt_update_pmtu()`

Router says:

“Packet too big — use MTU = 1400.”

Kernel calls the same helper:

`update_or_create_fnhe()`

This sets:

`fnhe_pmtu = <new MTU>`

`fnhe_expires = jiffies + ip_rt_mtu_expires`

(Default is 600 seconds = 10 minutes)

## ★ Why store exceptions per-nexthop?

Each nexthop (fib\_nh) handles many destinations.

Example:

Route: 0.0.0.0/0 via 10.0.0.1 dev eth0

Destinations:

8.8.8.8

1.1.1.1

172.217.0.0

PMTU for 8.8.8.8 may be 1400.

ICMP redirect for 1.1.1.1 may point to a new gateway.

These do *not* change the global route; they only apply for that specific destination.

Thus each nexthop holds exception entries, keyed by destination.

## ★ Exceptions Hash Table Size and Handling

- Size: **2048 buckets**
- Key: destination IP
- Reclaim rule: delete entries when hash chain > 5 entries
- They also auto-expire after timer lapse.

## ★ Kernel Versions Behavior

◊ Before Linux 3.6

No nexthop exception support.

◊ Linux 3.6

Introduced FIB nexthop exceptions in **Tx path**.

◊ Linux 3.11

Added exception lookup in **Rx path**.

Now exceptions influence both incoming and outgoing routing decisions.

## ★ Why exceptions improve performance?

Without exceptions:

- Every PMTU change requires re-running full lookup
- Every ICMP redirect forces rebuilding rtable
- Slow performance, especially for repeated packets

With exceptions:

- Exception stored once
- Re-used for subsequent packets
- Avoids hitting fib\_lookup repeatedly
- Uses cached rtable

## ★ Real Example

Router tells:

"MTU for 8.8.8.8 is 1400"

Kernel stores:

`fnhe_daddr = 8.8.8.8`

`fnhe_pmtu = 1400`

`fnhe_expires = now + 600 sec`

Next packet to 8.8.8.8:

- No need to perform PMTU discovery again
- Kernel uses exception
- Sets MTU = 1400
- Builds packet accordingly

## ★ Summary in One Line

**FIB nexthop exceptions are per-destination mini-routing overrides created automatically due to network events (ICMP Redirect and PMTU Discovery), stored inside each nexthop to speed up routing and avoid polluting the global routing table.**

## Want a Diagram?

If you want, I can draw:

- How fib\_table → fib\_alias → fib\_info → fib\_nh → fnhe\_exception → rtable connects
- The routing lookup flow including exception handling

Just say "**draw diagram**".

# POLY ROUTING

Thursday, December 11, 2025 12:06 PM

Below is the **cleanest, simplest, teacher-style explanation of Policy Routing**, based **only on the text you provided**, with highlighted important terms, meanings, and how everything fits together.

Let's go step by step.

## ★ POLICY ROUTING — SIMPLE EXPLANATION

Linux normally uses **only destination IP** to choose a route.

**Policy Routing** changes that.

→ With Policy Routing, the route chosen depends on **multiple factors**, not just destination:

- **source address**
- **TOS (Type of Service)**
- **fwmark**
- **interface**
- **UID**
- and more.

This allows advanced behaviors such as:

- routing traffic from different IPs through dd

## ★ IMPORTANT TERM #1 — CONFIG\_IP\_MULTIPLE\_TABLES

This kernel config decides:

- **OFF** → simple routing (only table *local* and *main*)
- **ON** → full Policy Routing support (up to **255 routing tables**)

## ★ WHEN POLICY ROUTING IS DISABLED

(i.e., **CONFIG\_IP\_MULTIPLE\_TABLES = 0**)

Linux creates **2 tables**:

### 1. Local Table (RT\_TABLE\_LOCAL = 255)

- contains routes for **local addresses**
- **only kernel can add/delete entries**
- includes:
  - 127.0.0.1 loopback
  - Your machine's IP addresses
  - Broadcast & multicast local entries

### 2. Main Table (RT\_TABLE\_MAIN = 254)

- contains **normal user-configurable routes**
- Admin adds/removes routes via:
  - ip route add
  - ip route del
  - routing daemons

These tables are created during boot in:

`fib4_rules_init()` (`net/ipv4/fib_frontend.c`)

## ★ WHEN POLICY ROUTING IS *ENABLED*

(i.e., `CONFIG_IP_MULTIPLE_TABLES = 1`)

Linux creates **3 initial tables**:

| Table          | ID  | Purpose                            |
|----------------|-----|------------------------------------|
| <b>local</b>   | 255 | Local addresses, created by kernel |
| <b>main</b>    | 254 | Normal routing entries             |
| <b>default</b> | 253 | Used for fallback/default routing  |

And you can add **up to 255 custom routing tables**.

This enables **ip rule**:

```
ip rule add from 192.168.1.5 table 100  
ip route add default via 10.10.10.1 table 100
```

Meaning:

Traffic **from** 192.168.1.5 uses **routing table 100**, not the main table.

## ★ IMPORTANT TERM #2 — `fib_get_table()`

This is a very important internal kernel function.

Regardless of simple routing or policy routing:

`fib_get_table(net, table_id)`

This returns the **routing table object** (`struct fib_table`) corresponding to:

- table 255
- table 254
- table 253
- or any of the custom tables you added

This gives a **unified way** to access routing tables.

## ★ HOW ROUTES ARE ADDED (USERSpace → Kernel)

### 1 Admin adds route:

```
ip route add 10.0.0.0/24 via 192.168.1.1
```

### 2 iproute2 sends:

```
RTM_NEWRROUTE
```

to kernel via Netlink.

### 3 Kernel receives in:

```
inet_rtm_newroute()
```

- ④ Kernel inserts route into:
- table **main** (default), unless table XYZ is specified

## ★ HOW ROUTES ARE DELETED

```
ip route del 10.0.0.0/24
```

Generates netlink message:

```
RTM_DELROUTE
```

Handled in:

```
inet_rtm_delroute()
```

## ★ HOW ROUTING TABLES ARE DISPLAYED

```
ip route show
```

Sends:

```
RTM_GETROUTE
```

Kernel responds via:

```
inet_dump_fib()
```

Default is to show **main table**.

To show local table:

```
ip route show table local
```

## ★ OLD route add command (legacy)

- Uses **IOCTL SIOCADDRT**
- Handled in kernel by:  
`ip_rt_ioctl()`

This is older than netlink/Iproute2.

## ★ ROUTING DAEMONS AND POLICY ROUTING

Large routers (BGP, OSPF, etc.) use userspace daemons:

- Zebra / Quagga / FRR
- OSPF daemons
- BGP daemons
- XORP

- MRoute
- BIRD

They update routing tables via **netlink messages**.

These daemons often maintain:

- their own dynamic routing tables
- constantly update table 254 (main) for Linux

## ★ ROUTES CHANGED BY KERNEL ITSELF

These are **NOT added by admin**, but kernel-generated:

### 1. ICMP Redirect

- Kernel updates gateway for a path
- Stored in **nexthop exception table**

### 2. Path MTU Discovery

- Kernel updates PMTU for a path
- Also stored in FIB nexthop exceptions

These are cached and not stored as normal routing entries.

## ★ SHORT SUMMARY

- ✓ Policy Routing allows routing decisions based on more than just destination
- ✓ Linux uses routing tables: local, main, default
- ✓ `fib_get_table()` provides unified access
- ✓ Routes are added via `RTM_NEWRROUTE` (netlink)
- ✓ Old route uses IOCTL
- ✓ Routing daemons manage large dynamic routing environments
- ✓ ICMP Redirects & PMTU changes generate special nexthop exceptions

If you want, I can draw:

- ✓ Diagram of **all routing tables**
- ✓ Diagram of **policy routing lookup flow**
- ✓ Relationship: **RULE → TABLE → FIB → fib\_info → fib\_nh → rtable → dst\_entry**

Just ask: “**draw full routing subsystem flow**”

# Fib alias

Thursday, December 11, 2025 2:02 PM

Below is a **clean, simple, beginner-friendly explanation of everything in your last message about FIB Alias (fib\_alias)** — what it is, why it exists, how it connects to fib\_info, fib\_table, and an **easy example**.

I will explain only **what is inside the paragraph**, plus background needed to understand it.

## ★ 1. What is a FIB Alias (fib\_alias)?

**Simple Definition:**

A **FIB alias** = a lightweight routing “entry variant” used when **multiple routes share the same destination/prefix**, but differ only in **TOS (Type of Service)** or similar minor fields.

It avoids wasting memory by **NOT creating separate fib\_info objects** for each rule.

## ★ 2. Why FIB Alias is Needed

Consider you add three routes:

```
ip route add 192.168.1.10 via 192.168.2.1 tos 0x2
ip route add 192.168.1.10 via 192.168.2.1 tos 0x4
ip route add 192.168.1.10 via 192.168.2.1 tos 0x6
```

These routes differ **only by TOS**.

Everything else (gateway, device, metrics, preferences) is the **same**.

☞ If the kernel created **three fib\_info objects**, it would waste memory.

So instead, it creates:

- **one fib\_info** (shared route data)
- **three fib\_alias objects** (TOS-specific variants)

Thus:

☞ **fib\_info holds route core details**

☞ **fib\_alias holds the small variations (tos, type)**

This is memory-efficient and faster.

## ★ 3. The Structure: struct fib\_alias

```
struct fib_alias {
    struct list_head fa_list;
    struct fib_info *fa_info;
    u8      fa_tos;
```

```
    u8      fa_type;
    u8      fa_state;
    struct rcu_head rcu;
};
```

Let's break these fields:

✓ **fa\_list**

A linked list pointer → all aliases for one prefix are linked together.

✓ **fa\_info**

Pointer to a **shared fib\_info object**.

This is the key point:

**different fib\_alias entries may all point to the same fib\_info.**

✓ **fa\_tos**

The TOS (Type of Service) value for this alias.

Used during route lookup when TOS matters.

✓ **fa\_type**

Route type for this alias (unicast, prohibit, local...).

Examples:

- RTN\_UNICAST
- RTN\_PROHIBIT (block traffic rule)
- RTN\_LOCAL

✓ **fa\_state**

Internal state flags (like whether it was recently used).

✓ **rcu**

Used so that removing an alias is safe under RCU synchronization.

## ★ 4. How fib\_alias Works With fib\_info and fib\_table

This is the MOST important point — the relationship:

**fib\_table**

|

└─ contains many **fib\_nodes** (prefixes)

|

└─ each **fib\_node** has a list of **fib\_alias**

|

└─ each **fib\_alias** points to a **fib\_info**

|

└─ **fib\_info** contains real route data

This is the hierarchy:

- ✓ **fib\_table**
  - Entire routing table (main, local, custom)
- ✓ **fib\_node**
  - Represents a prefix (like 192.168.1.0/24)
- ✓ **fib\_alias**
  - Actual routing rule for that prefix
  - Stores TOS + type + pointer to fib\_info
- ✓ **fib\_info**
  - Heavyweight structure holding full route parameters  
(device, gateway, metrics, protocol, scope, flags, nexthops)

## ★ 5. Example — How fib\_alias Is Created

You add the first route:

```
ip route add 192.168.1.10 via 192.168.2.1 tos 0x2
```

Kernel does:

1. No existing route → create **fib\_info A**
2. Create **fib\_alias #1** with:
  - TOS = 0x2
  - alias → fib\_info A

You add the second route:

```
ip route add 192.168.1.10 via 192.168.2.1 tos 0x4
```

Kernel logic:

3. Build new fib\_info (temporarily)
  4. Compare with existing fib\_info A:
    - They are identical (except TOS)
  5. Kernel **discards the new fib\_info**
  6. It **reuses fib\_info A**
  7. Create **fib\_alias #2**, TOS=0x4, fa\_info = fib\_info A
- Third route same story—fib\_alias #3.

Result:

fib\_info A

↑ ↑ ↑

alias1 alias2 alias3

tos2 tos4 tos6

## ★ 6. The Reference Counter: fib\_treeref

Because **one fib\_info can be shared by many aliases**,  
the kernel maintains:  
`fib_info->fib_treeref = number of fib_alias pointing to it`

When:

- a new alias is created → increment
- an alias is removed → decrement
- counter reaches 0 → route is removed → free fib\_info

## ★ 7. Important Line from the Paragraph

Because **TOS is stored in fib\_alias**, not fib\_info:

"You should remember that the TOS field exists in the fib\_alias object but not in the fib\_info object"

This is WHY fib\_info objects can be shared.

## ★ 8. Why fib\_alias matters (Importance)

- ✓ Saves memory
- ✓ Optimizes routing lookups
- ✓ Avoids duplicating identical route data
- ✓ Keeps route table lighter and faster
- ✓ Supports TOS-specific routing without bloat

## ★ 9. Summary in One Sentence

**fib\_alias = lightweight TOS-specific route variant that points to a shared fib\_info containing all heavy route parameters.**

If you want, I can also draw:

- █ A diagram showing fib\_table → fib\_node → fib\_alias → fib\_info
- █ How lookup uses fib\_alias and fib\_info
- █ How multipath routes fit into this
- █ How Policy Routing interacts with fib\_alias

Just say "**make diagram**" or "**explain lookup flow**".

# LONGEST-PREFIX MATCH (LPM)

Wednesday, December 10, 2025 2:24 PM

# Checksum offloading

Thursday, December 11, 2025 10:29 AM

Here is a **clean, deep, kernel-level explanation of checksum offloading in the Linux networking stack**, written exactly the way a kernel beginner needs: step-by-step, with structures, flags, and datapath details.

## ✓ What Is Checksum Offloading?

Checksum offloading means:

**The NIC (hardware) calculates or verifies checksums instead of the CPU.**

Linux supports offloading for:

### TX Offload

- TCP checksum
- UDP checksum
- IPv4 header checksum  
(IPv6 has no header checksum)

### RX Offload

- Checks whether checksum is valid and marks it in SKB

**Goal:** reduce CPU cycles and speed up networking.

## ✓ How Checksum Works Normally (without offload)

For TCP/UDP packets, the OS must compute:

Checksum = pseudo-header + L4 header + payload

This takes CPU time (especially for big packets).

Without offload:

`skb->ip_summed = CHECKSUM_NONE`

Stack must compute checksum manually before sending.

## ✓ How Linux TX Checksum Offloading Works

When a packet is created by TCP/UDP:

1. TCP sets:  
`skb->ip_summed = CHECKSUM_PARTIAL`  
meaning: *payload checksum left for hardware.*
2. NIC driver sees this flag in `.ndo_start_xmit()`
3. NIC hardware fills the correct checksum before putting frame on wire.

## ✓ What CHECKSUM\_PARTIAL means

This is the key flag.

```
enum {  
    CHECKSUM_NONE,  
    CHECKSUM_UNNECESSARY,  
    CHECKSUM_COMPLETE,  
    CHECKSUM_PARTIAL,  
};
```

### **CHECKSUM\_PARTIAL**

- Header already has pseudo header sum
- Hardware calculates rest

Linux sets up two fields in SKB:

#### **1. skb->csum\_start**

Offset (from skb->data) where L4 header starts

Example: offset of TCP header.

#### **2. skb->csum\_offset**

Offset inside L4 header where checksum field resides

Example: offsetof(struct tcphdr, check)

Hardware uses this:

```
final_checksum_location = csum_start + csum_offset
```

So driver must program NIC TX descriptor with these SKB parameters.

## TX Path Code Locations (Important)

### **Network stack sets checksum:**

tcp\_sendmsg()

```
-> tcp_transmit_skb()  
-> skb_checksum_help() (if needed)  
-> skb->ip_summed = CHECKSUM_PARTIAL
```

### **Driver reads checksum info:**

Example: Intel e1000e driver

File:

drivers/net/ethernet/intel/e1000e/netdev.c

Function:

e1000\_tx\_csum()

It checks:

```
if (skb->ip_summed == CHECKSUM_PARTIAL) {
```

```
// set Tx descriptor flags for hardware checksum  
}
```

## ✓ RX Checksum Offloading (Receive Side)

When NIC receives a packet:

NIC verifies checksum → gives status to driver.

Driver sets:

**CHECKSUM\_UNNECESSARY**

```
skb->ip_summed = CHECKSUM_UNNECESSARY
```

Meaning:

Checksum validated by hardware → do not recompute in stack

Then networking stack (IP/TCP) will **trust** this and skip verification.

## ✓ RX Path Code Locations

Example again e1000e:

```
drivers/net/ethernet/intel/e1000e/netdev.c
```

In RX function:

```
if (rx_desc->status & CHECKSUM_GOOD)  
    skb->ip_summed = CHECKSUM_UNNECESSARY;  
else  
    skb->ip_summed = CHECKSUM_NONE;
```

## ✓ Effects on Tools (tcpdump, wireshark)

When TX checksum offload is used, tcpdump sees "bad checksum".

Why?

Because tcpdump sees the packet before hardware fills checksum.

## ✓ How to Disable Offload (debugging)

```
sudo ethtool --offload eth0 rx off tx off
```

Check status:

```
ethtool -k eth0
```

## ✓ Important SKB Fields for Offload

#### **TX:**

skb->ip\_summed = CHECKSUM\_PARTIAL  
skb->csum\_start = offset to TCP/UDP header  
skb->csum\_offset = position of checksum field inside that header

#### **RX:**

skb->ip\_summed = CHECKSUM\_UNNECESSARY (hardware verified)  
skb->csum = hardware-provided checksum value

## ✓ Driver Responsibilities

Every NIC driver must:

#### **If CHECKSUM\_PARTIAL:**

- setup descriptor with csum\_start + csum\_offset

#### **If CHECKSUM\_NONE:**

- do nothing

#### **RX Path:**

- interpret NIC checksum status bits
- set ip\_summed properly
- if driver does not support offload → stack computes manually

## ✓ "cook\_skb" and GSO Interaction

If GSO (segmentation offload) is active:

Large SKB → hardware calculates checksum for every segment produced.

The stack sets CHECKSUM\_PARTIAL for the entire GSO packet.

## ✓ Files to Read in Kernel

Here is the exact path & file list to **learn checksum offload deeply**:

include/linux/skbuff.h  
include/linux/netdevice.h  
include/uapi/linux/net\_tstamp.h  
net/core/dev.c  
net/ipv4/tcp\_output.c  
net/ipv4/udp.c  
drivers/net/ethernet/\*/\* (specific NIC drivers)

Main logic is in:

net/core/dev.c → skb\_checksum\_help()

## ✓ Summary (Very Quick)

| Offload Type      | skb->ip_summed       | Who computes?          |
|-------------------|----------------------|------------------------|
| None              | CHECKSUM_NONE        | CPU                    |
| TX offload        | CHECKSUM_PARTIAL     | NIC HW                 |
| RX offload        | CHECKSUM_UNNECESSARY | NIC HW                 |
| Software-complete | CHECKSUM_COMPLETE    | Stack computed earlier |

If you want, I can also explain:

- GSO/TSO vs Checksum offload
- Flow diagrams of TX/RX checksum paths
- Walk-through of a real driver (e1000e / cpsw on TI / virtio-net)
- Show you exact code in csw.c (TI driver)

Understood — you want “**HOW EXACTLY checksum offloading happens inside the Linux kernel + NIC**”

Meaning: **each step, what touches what, how SKB fields get set, how kernel writes to NIC registers/descriptors, how NIC completes checksum.**

Here is the *internal step-by-step datapath*, no skipping, no magic.

## ★ HOW TX CHECKSUM OFFLOADING ACTUALLY HAPPENS

We start from your application writing data → TCP stack → SKB created → driver → NIC hardware → wire.

I'll describe each part in detail.

### 1 Application writes data → TCP builds SKB

User calls:

```
send()
```

TCP does:

```
tcp_sendmsg()  
→ tcp_transmit_skb()
```

Inside tcp\_transmit\_skb():

TCP sets:  
skb->ip\_summed = CHECKSUM\_PARTIAL  
skb->csum\_start = (offset of TCP header)  
skb->csum\_offset = offsetof(struct tcphdr, check)

## Why?

Linux wants the NIC to compute the final TCP checksum.

TCP builds *pseudo-header* checksum only:

$$\text{sum} = \text{src\_ip} + \text{dst\_ip} + \text{proto} + \text{length}$$

This is put into the TCP header's checksum field.

BUT not finalized — hardware will finish.

## 2 SKB goes to qdisc → to driver → ndo\_start\_xmit()

Example driver: **cpsw.c** on TI

Kernel calls the driver's transmit function:

```
dev->netdev_ops->ndo_start_xmit(skb, dev);
```

Inside driver TX function (pseudo code):

```
static netdev_tx_t cpsw_ndo_start_xmit(struct sk_buff *skb, struct net_device *ndev)
{
    if (skb->ip_summed == CHECKSUM_PARTIAL) {
        // hardware checksum offload required
        desc->flags |= CPDMA_DESC_TD_CSUM; // checksum enable
        desc->csum_start = skb->csum_start;
        desc->csum_offset = skb->csum_offset;
    }

    desc->buffer = skb->data;
    desc->len   = skb->len;

    submit_desc_to_dma(desc);
}
```

This is where the REAL magic starts:

The driver writes into the **DMA descriptor** fields:

- where checksum calculation should **begin** (csum\_start)
- where checksum field **is located** (csum\_offset)
- tells NIC “compute checksum” (some flag)

So the NIC now has enough information to do the checksum for us.

## 3 NIC DMA engine fetches packet data

NIC reads the descriptor:

csum\_start = value provided by driver

csum\_offset = value inside header

And starts reading packet bytes from memory via DMA.

## 4 NIC calculates checksum internally (hardware engine)

NIC performs:

sum = 0

```
for (i = csum_start; i < end_of_packet; i += 2)
    sum += 16-bit-word(packet[i:i+2]);
```

This is EXACTLY what the CPU would do, but hardware has a dedicated checksum unit.

At the end:

final\_checksum = ~sum

Then NIC writes this value **into the TCP header's checksum field** at:

location = csum\_start + csum\_offset

## 5 NIC transmits the packet with correct checksum

Now packet is **fully valid** and goes on the wire.

tcpdump running on Linux (before NIC) will see wrong checksum

tcpdump on a remote machine will see correct checksum

This is why.

## ★ HOW RX CHECKSUM OFFLOADING HAPPENS

Receive side is simpler but equally important.

## 1 NIC receives frame → checks checksum internally

NIC parses the Ethernet frame → IP → TCP/UDP header.

NIC computes checksum itself.

Depending on NIC, it sets bits in RX descriptor:

CHECKSUM\_GOOD

CHECKSUM\_BAD

CHECKSUM\_UNKNOWN

## 2 Driver processes RX descriptor → sets skb->ip\_summed

Example:

```
if (rx_desc->flags & CHECKSUM_GOOD)
    skb->ip_summed = CHECKSUM_UNNECESSARY;
else
    skb->ip_summed = CHECKSUM_NONE;
```

Meaning:

✓ CHECKSUM\_UNNECESSARY

Stack should NOT compute checksum — NIC has validated it.

✗ CHECKSUM\_NONE

Stack must compute checksum in software.

## 3 Packet continues through stack without re-checksum

IP/TCP code sees:

```
skb->ip_summed == CHECKSUM_UNNECESSARY
```

So it **skips expensive software checksum**.

## ★ SUMMARY DIAGRAM

### TX Path (offload)

User → TCP → sets CHECKSUM\_PARTIAL + offsets

→ SKB

→ Driver (ndo\_start\_xmit)

→ program NIC DMA descriptor:

    csum\_start

    csum\_offset

    checksum-enable flag

→ NIC hardware computes checksum

→ packet to wire (correct checksum)

### RX Path (offload)

Wire → NIC → validates checksum



```
RX descriptor checksum status  
↓  
Driver sets skb->ip_summed = CHECKSUM_UNNECESSARY  
↓  
Stack skips checksum computation
```

## ★ WANT EVEN DEEPER?

I can show you any of the following:

- Exact csw.c lines where checksum offload is implemented (TI AM335x/AM572x)
- Exact SKB lifecycle with offsets and memory layout
- Exact NIC descriptor format (CPDMA descriptors)
- Add debug prints to see hardware flags
- Full flow diagram from socket → NIC → wire

Just tell me what you want next.

Great question — now you are asking the **core mechanism**:

- How does the Linux kernel know if checksum offload is supported & enabled on the NIC hardware?

The answer involves:

- driver → netdevice features flags
- ethtool operations
- netdev->features and netdev->hw\_features
- negotiation between kernel and driver

Let's break it down completely.

## ★ 1. Driver tells Linux “my hardware supports checksum offload”

Each network driver (e.g., csw for TI, e1000e for Intel, etc.) sets hardware supported features in:

netdev->hw\_features

Example:

```
dev->hw_features = NETIF_F_HW_CSUM | NETIF_F_RXCSUM;
```

These flags describe **hardware capability**, not what is currently enabled.

## ★ 2. Linux copies hw\_features → features during device probe

During network device registration:

```
dev->features |= dev->hw_features;
```

So enabled features (dev->features) initially match supported hardware features (dev->hw\_features).

### ★ 3. These flags (features) are what TCP/IP stack checks

Common checksum offload flags:

| Flag                     | Meaning                  |
|--------------------------|--------------------------|
| <b>NETIF_F_HW_CSUM</b>   | TX checksum offload      |
| <b>NETIF_F_RXCSUM</b>    | RX checksum offload      |
| <b>NETIF_F_IP_CSUM</b>   | IPv4 checksum TX offload |
| <b>NETIF_F_IPV6_CSUM</b> | IPv6 TCP/UDP offload     |
| <b>NETIF_F_SCTP_CSUM</b> | SCTP checksum offload    |

When TX code wants to assign CHECKSUM\_PARTIAL, it checks:

```
if (netdev->features & NETIF_F_HW_CSUM)
    skb->ip_summed = CHECKSUM_PARTIAL;
else
    calculate csum in software
```

This is the exact decision point.

### ★ 4. How ethtool fits in (enabling/disabling offload)

User can enable/disable:

```
ethtool -K eth0 tx on/off
ethtool -K eth0 rx on/off
```

When you run this, kernel calls:

```
netdev->ethtool_ops->set_rx_csum()
netdev->ethtool_ops->set_tx_csum()
```

Driver must implement these and update:

```
dev->features
```

Example in cswp:

```
if (tx_enable)
    dev->features |= NETIF_F_HW_CSUM;
else
```

```
dev->features &= ~NETIF_F_HW_CSUM;
```

So kernel knows OFFLOAD ENABLED/DISABLED based on dev->features.

## ★ 5. The TCP/IP stack checks the feature flags in dev->features

When building a packet, TCP checks:

```
if (skb->dev->features & NETIF_F_CSUM_MASK)
    skb->ip_summed = CHECKSUM_PARTIAL;
else
    do full software checksum
```

This decision happens in:

```
net/core/dev.c → validate_xmit_skb()
net/ipv4/tcp_output.c
```

## ★ 6. Driver also checks these flags when preparing descriptors

In ndo\_start\_xmit():

```
if (skb->ip_summed == CHECKSUM_PARTIAL &&
    dev->features & NETIF_F_HW_CSUM)
{
    // program hardware csum in desc
}
```

So both **TCP stack** and **NIC driver** use the same flag.

## ★ 7. Final Answer (How kernel knows?)

✓ Hardware support

→ Driver sets it during probe in

dev->hw\_features

✓ Enabled or not

→ ethtool commands modify

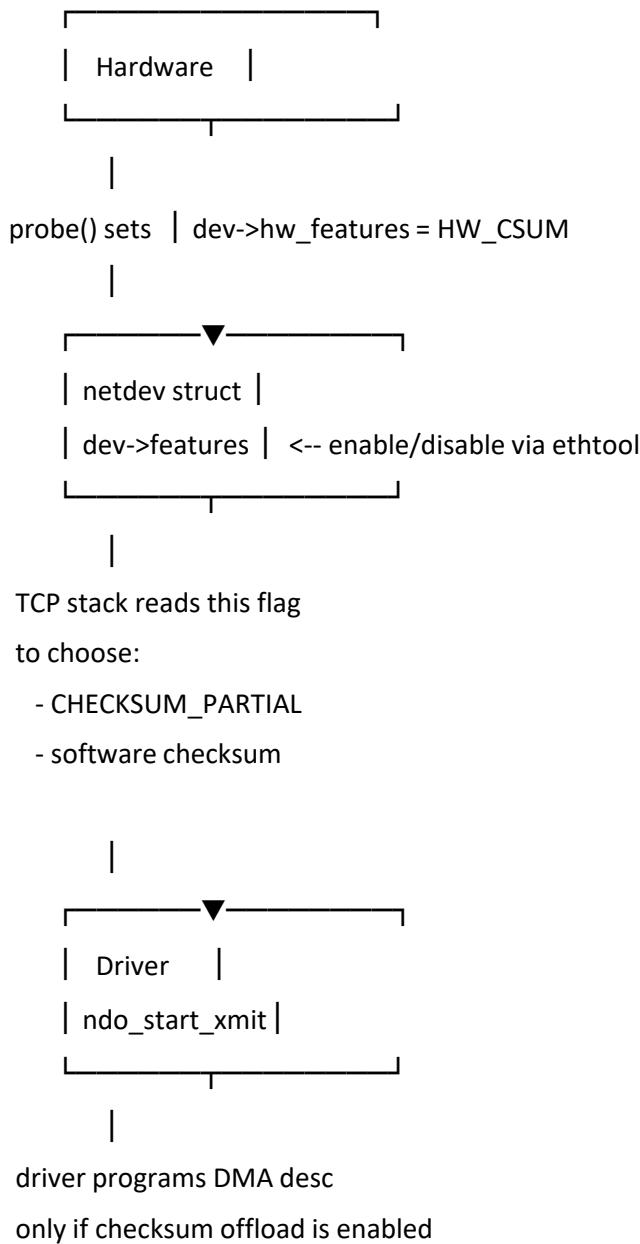
dev->features

✓ Kernel checks dev->features to decide:

- should it compute checksum?
- or ask NIC to compute checksum?

- ✓ Driver checks the same flag before programming NIC descriptor.

## ★ Visual Summary



## ★ If you want, I can now show you:

- ✓ exact cpsw.c code where hw\_features/features are set
- ✓ ethtool ops for CPSW (TI)
- ✓ where dev->features is checked inside TCP stack
- ✓ where validate\_xmit\_skb() forces software checksum

Just tell me which one.

# Ethtool tool

Thursday, December 11, 2025 10:50 AM

## ★ 1. What is ethtool?

ethtool is a **userspace tool** used to query and configure **network interface parameters** through a **special ioctl interface** exposed by the kernel.

ethtool lets you control:

- checksum offload
- GSO/TSO/LRO
- link speed / duplex modes
- pause / flow control
- link statistics
- EEPROM access
- ring sizes
- channels / RSS config
- coalescing (interrupt moderation)
- driver info
- wake-on-LAN
- many more...

## ★ 2. Architecture: How ethtool works end-to-end

**Userspace (ethtool) ↔ Kernel (ethtool ioctl) ↔ Driver (ethtool\_ops)**

Full pipeline:

[User runs command]



[ethtool program forms ETHTOOL\_xxx struct]



[issues ioctl(SIO CETHTOOL)]



[kernel receives via dev\_ioctl()]



[calls \_\_ethtool\_ioctl() in net/core/ethtool.c]



[kernel parses command]



[calls netdev->ethtool\_ops->XXXX()]

↓

[driver executes hardware setting]

Perfect — here is **exactly what you want**:

- ✓ **ALL important ethtool commands**
- ✓ **What they do (functionality)**
- ✓ **Which ETHTOOL ioctl code they use**
- ✓ **Which driver callback (ethtool\_ops->xxx) gets called**

Presented cleanly and deeply.

## ★ MASTER TABLE — ethtool COMMAND → ACTION → IOCTL → DRIVER FUNCTION

### 1. Driver & Basic Device Info

#### 1.1 Driver Information

Command:

ethtool -i eth0

What it does:

- Shows driver name
- Version
- Firmware version
- Bus info (PCI address)

IOCTL:

ETHTOOL\_GDRVINFO

Driver Function:

ops->get\_drvinfo()

#### 1.2 Link Status

Command:

ethtool eth0

What it does:

- Shows speed, duplex, autoneg status, port type
- Shows “Link detected: yes/no”

IOCTL:

## ETHTOOL\_GLINKSETTINGS

Driver Function:

`ops->get_link_ksettings()`

### 2. Link Configuration (Speed / Duplex / Autoneg)

#### **2.1 Set link parameters**

Command:

`ethtool -s eth0 speed 100 duplex full autoneg off`

What it does:

- Forces PHY/NIC speed & duplex
- Can enable/disable auto-negotiation

IOCTL:

ETHTOOL\_SLINKSETTINGS

Driver Function:

`ops->set_link_ksettings()`

### 3. Checksum Offloading

#### **3.1 View all offload features**

Command:

`ethtool -k eth0`

What it does:

- Shows status of:
- TX checksum
- RX checksum
- TSO/GSO/LRO
- GRO/SG

IOCTL:

ETHTOOL\_GFEATURES

Kernel function:

`ethtool_get_features()`

#### **3.2 Enable/Disable TX Checksum**

Command:

`ethtool -K eth0 tx on/off`

**What it does:**

- Enables NIC hardware to compute TCP/UDP checksums
- If OFF → kernel computes checksum in software

**IOCTL:**

ETHTOOL\_STXCSUM

**Driver Function:**

ops->set\_tx\_csum()

### 3.3 Enable/Disable RX Checksum

**Command:**

ethtool -K eth0 rx on/off

**What it does:**

- Enables NIC hardware to verify checksum
- If OFF → kernel verifies checksum in software

**IOCTL:**

ETHTOOL\_SRXCSUM

**Driver Function:**

ops->set\_rx\_csum()

## 4. TSO / LRO / GSO / GRO Features

### 4.1 TCP Segmentation Offload

**Command:**

ethtool -K eth0 tso on/off

**What it does:**

- Lets NIC split large TCP packets (e.g. 64KB) into 1500B frames
- Improves performance for large transfers

**IOCTL:**

ETHTOOL\_STSO

**Driver:**

ops->set\_tso()

### 4.2 Generic Segmentation Offload

**Command:**

ethtool -K eth0 gso on/off

**What it does:**

- Kernel segments big packets before sending to NIC (software TSO)

IOCTL:

ETHTOOL\_SGSO

Driver:

ops->set\_sg()

### 4.3 Large Receive Offload (LRO)

Command:

ethtool -K eth0 lro on/off

What it does:

- NIC merges multiple incoming TCP frames into one large buffer

IOCTL:

ETHTOOL\_SLRO

Driver may handle inside:

ops->set\_rx\_csum() or PHY handlers



## 5. Ring Buffers (Descriptor Rings)

### 5.1 Show ring sizes

Command:

ethtool -g eth0

What it does:

- Shows RX/TX descriptor ring sizes

IOCTL:

ETHTOOL\_GRINGPARAM

Driver:

ops->get\_ringparam()

### 5.2 Set ring sizes

Command:

ethtool -G eth0 rx 512 tx 256

What it does:

- Adjusts descriptor queues
- Larger rings → fewer drops
- Smaller rings → less memory usage

IOCTL:

ETHTOOL\_SRINGPARAM

Driver:

ops->set\_ringparam()

## 6. Coalescing (Interrupt Moderation)

### 6.1 View coalescing

Command:

ethtool -c eth0

What it does:

- Shows RX/TX interrupt delay, packet thresholds

IOCTL:

ETHTOOL\_GCOALESCE

Driver:

ops->get\_coalesce()

### 6.2 Set coalescing

Command:

ethtool -C eth0 rx-usecs 40 tx-usecs 20

What it does:

- Reduces interrupt load
- NIC interrupts only after N microseconds of traffic

IOCTL:

ETHTOOL\_SCOALESCE

Driver:

ops->set\_coalesce()

## 7. Channel / RSS Settings

### 7.1 Show channels

Command:

ethtool -l eth0

What it does:

- Shows # of TX/RX queues, RSS queue count

IOCTL:

## ETHTOOL\_GCHANNELS

Driver:

ops->get\_channels()

### 7.2 Set channels

Command:

ethtool -L eth0 combined 4

What it does:

- Sets number of RX/TX queues
- Enables multiqueue / RSS tuning

IOCTL:

ETHTOOL\_SCHANNELS

Driver:

ops->set\_channels()



## 8. Statistics

### 8.1 NIC statistics

Command:

ethtool -S eth0

What it does:

- Shows hardware counters
- Rx drops
- Tx drops
- CRC errors
- alignment errors
- DMA errors

IOCTL:

ETHTOOL\_GSTATS

Driver:

ops->get\_ethtool\_stats()



## 9. EEPROM Operations

Command:

ethtool -e eth0

ethtool -E eth0 magic 0x1234 data 0x56

### What it does:

- Read/write NIC EEPROM or PHY EEPROM

#### IOCTLs:

ETHTOOL\_GEEPROM

ETHTOOL\_SEEPROM

#### Driver:

ops->get\_eeprom()

ops->set\_eeprom()

## ▀ 10. Pause / Flow Control

#### View pause settings:

ethtool -a eth0

#### IOCTL:

ETHTOOL\_GPAUSEPARAM

#### Driver:

ops->get\_pauseparam()

#### Set pause:

ethtool -A eth0 rx on tx off

#### IOCTL:

ETHTOOL\_SPAUSEPARAM

#### Driver:

ops->set\_pauseparam()

## ★ FINAL SUMMARY TABLE (Clean & Quick)

| ethtool command | What it does  | IOCTL            | Driver Function    |
|-----------------|---------------|------------------|--------------------|
| -i              | Driver info   | ETHTOOL_GDRVINFO | get_drvinfo        |
| (plain)         | Link settings | GLINKSETTINGS    | get_link_ksettings |
| -s              | Set link      | SLINKSETTINGS    | set_link_ksettings |
| -k              | Show offloads | GFEATURES        | internal           |
| -K tx           | TX checksum   | STXCSUM          | set_tx_csum        |
| -K rx           | RX checksum   | SRXCSUM          | set_rx_csum        |

|        |               |             |                   |
|--------|---------------|-------------|-------------------|
| -K tso | TSO           | STSO        | set_tso           |
| -K gso | GSO           | SGSO        | set_sg            |
| -G     | Set rings     | SRINGPARAM  | set_ringparam     |
| -g     | Show rings    | GRINGPARAM  | get_ringparam     |
| -c     | Show coalesce | GCOALESCE   | get_coalesce      |
| -C     | Set coalesce  | SCOALESCE   | set_coalesce      |
| -I     | Show channels | GCHANNELS   | get_channels      |
| -L     | Set channels  | SCHANNELS   | set_channels      |
| -S     | Stats         | GSTATS      | get_ethtool_stats |
| -a     | Show pause    | GPAUSEPARAM | get_pauseparam    |
| -A     | Set pause     | SPAUSEPARAM | set_pauseparam    |
| -e     | EEPROM read   | GEEPROM     | get_eeprom        |
| -E     | EEPROM write  | SEEPROM     | set_eeprom        |

If you want next:

- FULL path: ethtool command → userspace struct → kernel ioctl parser → driver → hardware registers**
- Flow diagrams**
- Walkthrough inside csw.c specifically**

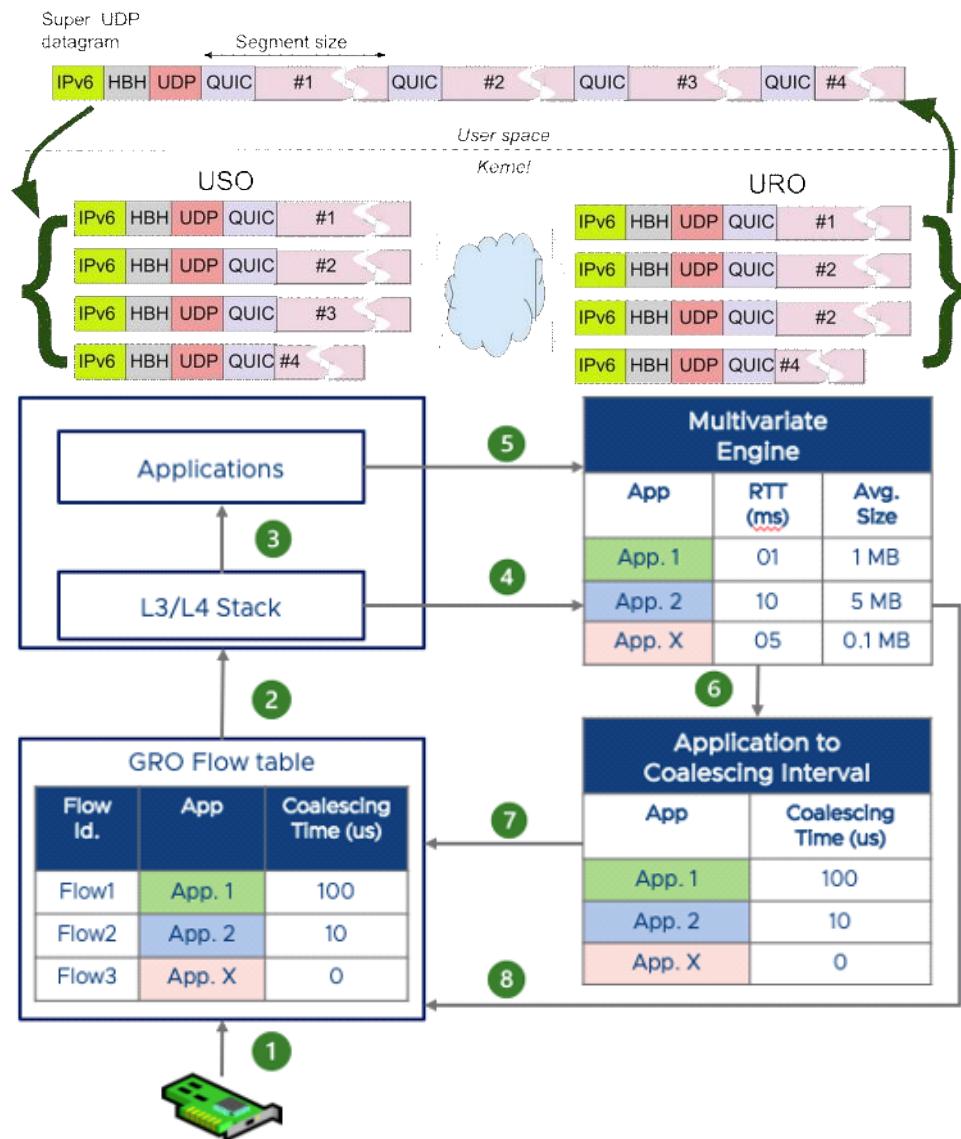
Tell me and I'll continue.

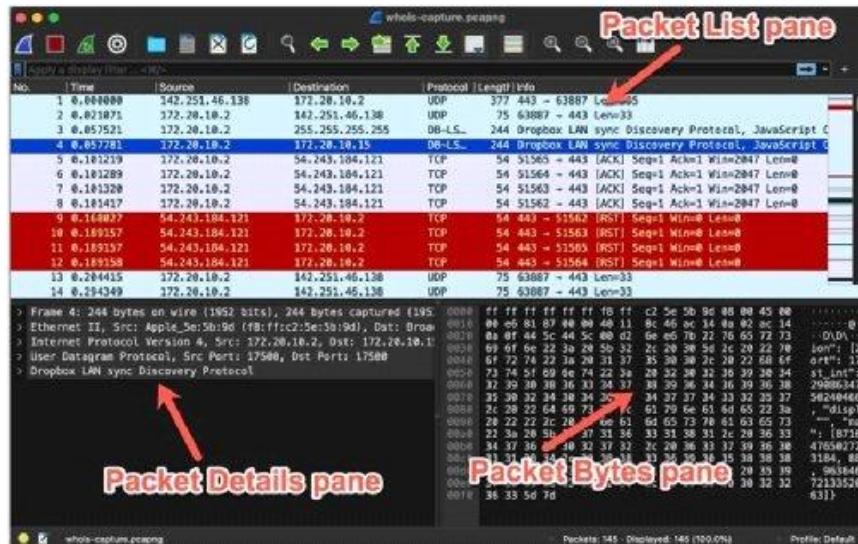
# GRO AND LRO

Thursday, December 11, 2025 11:33 AM

Below is a **very clear, kernel-internals-level explanation of LRO vs GRO**, exactly how they work, how they differ, how they appear in the RX path, and why GRO replaced LRO.

To make the concepts visually intuitive, here are helpful diagrams:





## ★ LRO (Large Receive Offload) vs GRO (Generic Receive Offload)

Both merge multiple incoming packets into one larger packet → reduce CPU overhead.

But HOW, WHERE, and WHEN they do this is completely different.

### ★ LRO — Large Receive Offload

✓ Happens in NIC hardware

LRO is performed **inside the network card**, before the data ever reaches the kernel stack.

### ★ What LRO does

When several TCP segments arrive **in order**, with:

- Same src/dst IP
- Same src/dst port
- Increasing sequence numbers
- No gaps, no flags, no retransmissions

**NIC merges multiple TCP packets into one big packet** (like  $4 \times 1500 \rightarrow 6000$  bytes).

So instead of generating 4 interrupts and 4 SKBs, NIC provides **1 big skb**.

### ★ How LRO works (hardware behavior)

1. NIC receives multiple packets: P1, P2, P3, P4
2. NIC validates they are part of the same TCP flow
3. NIC merges them into one frame
4. NIC gives kernel a single giant packet

Kernel never sees the individual packets.

### ★ Pros of LRO

- Very high performance improvement on RX path
- Fewer interrupts
- Fewer SKBs to process
- Great for high-throughput servers

## ★ Cons of LRO — Why Linux doesn't like it

### ✗ Breaks packet semantics

Kernel and upper layers (bridge, forwarding, netfilter, iptables, conntrack, TCP congestion control) expect packets individually.

### ✗ Breaks forwarding/router roles

If Linux is a router/bridge → LRO corrupts packet boundaries → cannot forward.

### ✗ Incompatible with virtualization / overlay networks

For example:

- VXLAN
- GRE
- Open vSwitch
- SR-IOV
- KVM guest passthrough

Because the NIC merges packets before hypervisor or tunnel decapsulation.

## ➊ GRO — Generic Receive Offload

### ✓ Happens in the Linux kernel, in software

This is Linux's replacement for LRO.

**GRO is done in the kernel networking stack (softirq), not in hardware.**

### ★ What GRO does

Kernel receives multiple packets (SKBs) and **merges them at the SKB layer**, not in NIC.

Flow is:

5. Rx interrupt
6. NAPI poll
7. GRO engine checks if packets belong to the same flow
8. Merges SKBs **logically** (not physically modifying hardware DMA buffers)
9. Passes merged packet up the stack

### ★ How GRO works step by step

10. NIC driver allocates SKB for each packet normally
11. NAPI poll loops collect packets and put them in a GRO list
12. Linux merges packets into a single SKB **only in the receive path**
13. Upper-layer protocol (TCP) receives a coalesced SKB
14. If forwarding is needed → GRO automatically **flushes** packets and doesn't break semantics

## ★ Benefits of GRO

- ✓ Safe for routers, bridges, and forwarding
- ✓ Works with tunnels (VXLAN, GRE, GENEVE — GRO has special handlers)
- ✓ Works with virtualization (virtio-net GRO, software tunnels)
- ✓ Linux controls aggregation logic → always correct semantics
- ✓ Works naturally with RFS, RPS, XDP, DDP

GRO is smarter, safer, and more compatible than LRO.

## ★ Why GRO replaced LRO

- LRO merges at *hardware layer* → breaks packet boundaries
- GRO merges at *software layer* → kernel can choose when to merge or not
- GRO supports tunneling & virtualized networking
- LRO cannot handle encapsulated packets
- GRO has per-protocol merge handlers (TCP, UDP, VXLAN, UDP-GSO, geneve-gro, etc.)

Modern kernels **disable LRO by default** and rely on GRO.

## 💧 Key Differences Table (Short & Clear)

| Feature                     | LRO                                    | GRO                                       |
|-----------------------------|----------------------------------------|-------------------------------------------|
| Where it runs               | NIC hardware                           | Linux kernel (software)                   |
| Visibility                  | Merges packets before kernel sees them | Kernel sees packets individually first    |
| Safe for forwarding         | ✗ No                                   | ✓ Yes                                     |
| Tunnel support (VXLAN etc.) | ✗ No                                   | ✓ Yes                                     |
| Virtualization friendly     | ✗ Bad                                  | ✓ Excellent                               |
| Protocol support            | TCP only                               | TCP, UDP, tunnels, GRO for many protocols |
| Replaceable / extensible    | ✗ Fixed in hardware                    | ✓ Yes, kernel code can extend             |
| Recommended                 | ✗ No                                   | ✓ Yes                                     |

## ★ When to disable or enable them

### ⌚ Disable LRO when:

- Linux is a router / firewall
- Using bridge
- Using tunneling
- Running VMs (KVM, Xen)
- Using DPDK or XDP
- Using NIC passthrough
- Doing traffic analysis or capture

### ⌚ Enable GRO always (default)

GRO gives best performance and compatibility.

## ★ Commands

Disable LRO:

```
ethtool -K eth0 lro off
```

Disable/Enable GRO:

```
ethtool -K eth0 gro off
```

```
ethtool -K eth0 gro on
```

## ★ Summary (One-Liner)

- **LRO = hardware merges packets → fast but breaks networking.**
- **GRO = kernel merges packets intelligently → fast AND correct.**

If you want:

- A **diagram** of LRO vs GRO packet flow inside the kernel
- A **step-by-step walkthrough** in **net/core/dev.c** showing where GRO happens
- How GRO interacts with **TSO, GSO, RX checksum offload**

Just say the word!

# ICMP redirect

Friday, December 12, 2025 10:04 AM

Below is a **clear, simple, and detailed explanation** of everything in the paragraph you just pasted about **ICMPv4 Redirect Messages** — including:

- What a redirect is
- Why it exists
- When the kernel sends it
- What the four redirect codes mean
- The example in the paragraph (AMD → Windows → Laptop)
- The role of routing tables and how the kernel detects a suboptimal path
- How the Linux kernel triggers the redirect message internally

This will give you perfect clarity before you dive into kernel functions.

## ★ 1. What is an ICMP Redirect Message? (Simple Meaning)

An **ICMPv4 Redirect** message is sent by a router to a host when:

→ The router notices the host is sending packets to the wrong gateway

There exists a **better, shorter, or more direct** route.

Examples:

- Host sends traffic to Router A
- Router A knows the correct next hop is Router B
- Router A also sees that the host could have sent directly to Router B
- So Router A forwards the packet AND sends a **Redirect** telling the host:

“Next time, send the packet directly to that better gateway, not me.”

The goal is **optimization**, not correctness.

Packets still get forwarded, just more slowly.

## ★ 2. Important Point: Redirects happen only when a route is *suboptimal*.

A **suboptimal route** happens when:

`incoming_interface == outgoing_interface`

Example:

Host → Router → Same LAN

If input and output device are equal, the router says:

“Why did you send this to me? You could reach the destination yourself!”

## ★ 3. Types of ICMP Redirect Codes

| ICMP Code                 | Meaning                      | Example                                 |
|---------------------------|------------------------------|-----------------------------------------|
| <b>ICMP_REDIR_NET</b>     | Redirect entire network      | "Use another router for 192.168.2.0/24" |
| <b>ICMP_REDIR_HOST</b>    | Redirect for a specific host | "Use this gateway for 192.168.2.7 only" |
| <b>ICMP_REDIR_NETTOS</b>  | Redirect network + TOS value | Used when routing depends on TOS        |
| <b>ICMP_REDIR_HOSTTOS</b> | Redirect host + TOS value    | Rare, QoS-specific routing              |

Most common in real life: **ICMP\_REDIR\_HOST**

## ★ 4. Example from the paragraph (explained clearly)

Network Setup:

192.168.2.200 → AMD server

192.168.2.10 → Windows server (gateway)

192.168.2.7 → Laptop

AMD server route:

ip route add 192.168.2.7 via 192.168.2.10

So AMD sends **all packets for laptop to Windows**.

But all three machines are on the **same LAN**:

192.168.2.0/24

Thus AMD *could* send packets directly to laptop (192.168.2.7)

— no gateway needed.

**What happens:**

1. AMD sends packet for 192.168.2.7 → to Windows
2. Windows forwards it
3. Windows detects:  
incoming\_interface == outgoing\_interface
4. Windows sends an **ICMP Redirect: Redirect Host (ICMP\_REDIR\_HOST)** telling AMD:  
"Stop sending traffic for 192.168.2.7 through me.  
Send it directly to the laptop."
5. AMD updates its routing table with a better gateway for that host.

## ★ 5. Why Redirects Are Useful

Redirect messages:

- ✓ Reduce unnecessary traffic through routers
- ✓ Shorten paths
- ✓ Improve latency
- ✓ Reduce router load
- ✓ Automatically optimize routes in LANs

## ★ 6. Conditions required for Linux to send a Redirect

Linux sends a redirect ONLY if **all** of these are true:

- 1 Source host must be on the same LAN

The packet must come from a host directly reachable on link layer.

- 2 Input interface == Output interface

This is the core suboptimal route condition.

- 3 Destination is also reachable via same interface

Meaning the host can route there directly.

- 4 Redirects must be enabled

/proc/sys/net/ipv4/conf/\*/send\_redirects must be = 1

- 5 The route is not prohibited by rules (e.g., netfilter)

## ★ 7. How Linux Actually Generates the Redirect Internally

When forwarding inside `ip_forward()`:

- 6 Routing lookup picks the outgoing route
- 7 Checks if **RTCF\_DOREDIRECT** flag is set
- 8 The input and output interfaces match
- 9 `ip_rt_send_redirect()` is called
- 10 ICMP header built with correct code
- 11 Packet is sent back to original sender

Important kernel structures involved:

- `rtable` → routing decision result
- `dst_entry` → destination cache entry
- `fib_result` → routing lookup result
- `fib_info` → routing entry info
- `fib_nh` → nexthop information

## ★ 8. What Happens in the Host After Receiving Redirect

The receiving host:

- 12 Updates its own routing table with a **host-specific route**
- 13 Or updates a nexthop exception entry (depends on kernel version)
- 14 Redirect persists until timeout (10 minutes by default)

This avoids repeatedly sending traffic through the old gateway.

## ★ FINAL SUMMARY (Teacher Level)

ICMP Redirect messages are sent when:

- A router forwards a packet

- ⦿ But notices the host could reach the destination directly
- ⦿ Input and output interface match
- ⦿ Redirect signaling is allowed

The router forwards the packet **and sends a redirect** to optimize future traffic.

There are 4 types (NET, HOST, NETTOS, HOSTTOS) depending on destination and TOS.

In your example, Windows sends **Host Redirect** to AMD telling it to send packets directly to the laptop.

This is part of the **IPv4 routing subsystem**, tied to:

- fib\_info
- fib\_alias
- fib\_table
- rtable
- dst\_entry

Linux implements redirect message generation inside:

→ `ip_forward() → ip_rt_send_redirect()`

Below is a **clear + simple explanation of ICMPv4 Redirect generation**, covering:

- why redirects exist
- exact kernel functions involved
- meaning of flags
- the two-phase mechanism
- the conditions required
- the example you gave
- what each structure and flag is used for

Written in the same style you asked for so you can turn this directly into your notes.

## ★ Why ICMPv4 Redirect Messages Exist

Routers send ICMP Redirect messages when they notice a host is **sending packets through a suboptimal path**.

Example:

- Host A → sends packets to Gateway B → intended for Host C
- But Host A could send **directly** to Host C
- Gateway B sends an **ICMP Redirect** telling A:  
“**Next time, send your traffic directly to Host C.**”

## ★ ICMPv4 Redirect Message Types

| Code                      | Meaning                         |
|---------------------------|---------------------------------|
| <b>ICMP_REDIR_NET</b>     | Redirect for a network (subnet) |
| <b>ICMP_REDIR_HOST</b>    | Redirect for a single host      |
| <b>ICMP_REDIR_NETTOS</b>  | Redirect for network + TOS      |
| <b>ICMP_REDIR_HOSTTOS</b> | Redirect for host + TOS         |

## ★ Where It Happens in Linux

Redirect generation happens in **two stages**:

### ◇ PHASE 1 — Decide whether redirect should be sent

Function:

`__mkroute_input()`

This function **sets the flag** that indicates a redirect should be sent:

`RTCF_DOREDIRECT`

This flag is a member of:

`struct rtable: rt_flags`

The flag is set ONLY if all conditions are met.

## ★ Conditions for Redirect (must ALL be true)

Linux only sends a redirect if:

**1** Input device == Output device

Means traffic came in and is going out **through the same NIC**.

Example:

Packet enters eth0 → kernel decides output is also eth0.

This is the strongest indicator of a suboptimal route.

**2** `/proc/sys/net/ipv4/conf/<if>/send_redirects` is enabled

If admin disabled this, **no redirect is sent**.

**3** Either:

- The device is **shared media** (like Ethernet), **OR**
- The sender and the next-hop gateway are **on the same subnet**

That ensures the sender **can** reach the real destination directly.

This check is:

```
IN_DEV_SHARED_MEDIA(out_dev) ||  
inet_addr_onlink(out_dev, saddr, FIB_RES_GW(*res))
```

Meaning:

✓ shared network

**OR**

✓ sender and real gateway are “on-link” (same subnet)

✓ If all conditions are true:

```
flags |= RTCF_DOREDIRECT;
```

and

**kernel sets redirect flag in rtable entry**

## ◇ PHASE 2 — Actually Send the Redirect

Function:

[ip\\_forward\(\)](#)

During forwarding, the kernel checks:

```
if (rt->rt_flags & RTCF_DOREDIRECT && !opt->srr && !skb_sec_path(skb))
    ip_rt_send_redirect(skb);
```

Redirect is sent ONLY if:

- The RTCF\_DOREDIRECT flag is set
- No strict source route option (opt->srr)
- Packet is not IPsec (skb\_sec\_path())

## ★ Final Step: Send the message

Function:

[ip\\_rt\\_send\\_redirect\(\)](#)

This constructs and sends the actual ICMP packet:

```
icmp_send(skb, ICMP_REDIRECT, ICMP_REDIR_HOST,
          rt_nexthop(rt, ip_hdr(skb)->daddr));
```

The **third argument** is the new gateway that should be used in future.

## ✿ Example from the Paragraph Explained Simply

Setup:

- AMD Server: **192.168.2.200**
- Windows Server (gateway): **192.168.2.10**
- Laptop: **192.168.2.7**

AMD incorrectly sends traffic to Laptop via Windows:

192.168.2.200 → 192.168.2.10 → 192.168.2.7

But AMD and the laptop are **on the same subnet**:

- ✓ shared media
- ✓ same input/output device
- ✓ same subnet
- ✓ send\_redirects enabled

Thus conditions for redirect = met.

Windows sends:

ICMP REDIRECT (ICMP\_REDIR\_HOST)  
New gateway: 192.168.2.7

## ★ What Happens Inside the Kernel for This Example?

Step 1: Packet arrives at Windows on eth0

Step 2: \_\_mkroute\_input() sees:

- in\_dev == out\_dev → ✓
- send\_redirects enabled → ✓
- on-link check passes → ✓

So kernel sets:

RTCF\_DOREDIRECT

Step 3: ip\_forward() sees this flag and calls:

ip\_rt\_send\_redirect()

**Step 4: ICMP Redirect sent back to AMD**

AMD updates its routing table for 192.168.2.7 (temporary exception).

## ★ Summary Table for Notes

| Stage                   | Function                            | Purpose                                                         |
|-------------------------|-------------------------------------|-----------------------------------------------------------------|
| <b>Decision</b>         | __mkroute_input()                   | Determine whether redirect should be sent & set RTCF_DOREDIRECT |
| <b>Check conditions</b> | inside __mkroute_input()            | Input==Output, shared media/onlink, send_redirects=1            |
| <b>Forwarding</b>       | ip_forward()                        | Detect flag, ensure no strict source route or IPsec             |
| <b>Send Redirect</b>    | ip_rt_send_redirect() → icmp_send() | Send ICMP redirect message                                      |

Absolutely — let's take this entire last section “**Receiving an ICMPv4 Redirect Message**” and break it down into:

- Very simple theoretical explanation
- Why each step exists
- What each function does

- What happens internally
- What special cases are handled

I'll ignore the unrelated parts and explain ONLY what is in the paragraph **clearly and in detail**.

## ★ Receiving an ICMPv4 Redirect Message — Simple Theory

When a host or router receives an ICMP Redirect, it means:

*"Hey! The route you are using is not optimal — use this gateway instead."*

Linux must validate the redirect and then update its routing cache accordingly.

This processing happens inside:

`__ip_do_redirect()`

This is the **core handler** for incoming ICMP Redirect messages.

## ❖ Step-by-Step Explanation of the Paragraph

### 1 Extract information from incoming ICMP Redirect

Code:

```
__be32 new_gw = icmp_hdr(skb)->un.gateway;
__be32 old_gw = ip_hdr(skb)->saddr;
```

Meaning:

- **new\_gw** = gateway being suggested
- **old\_gw** = gateway that originally forwarded the packet (the ICMP sender)

#### ❖ Why:

Redirect packet contains:

- old gateway
- suggested new gateway

Linux must check whether this redirect is *trustworthy*.

### 2 Verify device settings

Redirects are **disabled** on many servers for security reasons.

Relevant checks:

```
if (!IN_DEV_RX_REDIRECTS(in_dev))
    goto reject_redirect;
```

#### ⌚ Security Reason:

Redirects can be used for **man-in-the-middle attacks**, so Linux often disables them unless explicitly enabled.

### 3 Reject bad or malicious redirects

Linux verifies:

- ✓ new gateway is not multicast
- ✓ not broadcast
- ✓ not zero-net
- ✓ on the same subnet
- ✓ interface allows redirects
- ✓ packet comes from the current gateway
- ✓ routing table is consistent

Examples:

```
if (ipv4_is_multicast(new_gw) || ipv4_is_lbcast(new_gw))  
    reject;
```

#### ❖ Why:

Malicious actors can send incorrect redirect messages to poison routing.

### 4 Check that input device and redirect make sense

Linux rejects if:

- new\_gw is not reachable from this interface
- using shared media (WiFi, hubs, etc.) changes redirect rules
- secure redirect rules apply

Example rule:

```
if (!inet_addr_onlink(in_dev, new_gw, old_gw))  
    reject_redirect;
```

#### ❖ Why:

Redirect only makes sense if **new\_gw is directly reachable**.

### 5 Lookup the new gateway in ARP/neighbour table

```
n = ipv4_neigh_lookup(&rt->dst, NULL, &new_gw);
```

Meaning:

“Do we know the MAC address for this new gateway?”

If not:

```
neigh_event_send()
```

Triggers ARP resolution.

## 6 Lookup the routing entry for the destination that triggered the redirect

```
fib_lookup(net, fl4, &res)
```

Meaning:

“Find the exact route that carried the original packet.”

This returns a **fib\_result**, which contains nexthop(s).

## 7 Update FIB Nexthop Exception Table

```
update_or_create_fnhe(nh, fl4->daddr, new_gw, 0, 0);
```

This is the **actual fix** of the routing optimization.

### What is a FIB nexthop exception?

A small entry that:

- Overrides the nexthop gateway
- Overrides PMTU (if needed)
- Is used only for this specific destination

#### ❖ Purpose:

Apply redirect **without modifying the user's routing tables**.

Linux only updates the RT cache, **not the static routing table**.

## 8 Possibly mark route for deletion

```
if (kill_route)
    rt->dst.obsolete = DST_OBSOLETE_KILL;
```

Meaning:

“This dst entry is now outdated — flush it.”

## 9 Notify neighbour subsystem

```
call_netevent_notifiers(NETEVENT_NEIGH_UPDATE, n);
```

Meaning:

“Routing changed → ARP/neighbour subsystem must update state.”

## **10** Final return or rejection

If checks fail:

goto reject\_redirect;

Redirect is ignored.

## **FULL THEORETICAL SUMMARY (Simple Words)**

When Linux receives an ICMP Redirect:

15. Extract new gateway
16. Check device settings prevent malicious redirects
17. Reject invalid or dangerous redirects
18. Verify new gateway is on the same link
19. Ensure ARP entry exists or create it
20. Find the current route
21. Add a nexthop exception (small temporary override)
22. Mark stale routes for deletion
23. Notify ARP/neighbour layer
24. Done

## **★ WHY DOES LINUX HANDLE REDIRECTS LIKE THIS?**

Because:

- Users should not see routing table changes caused by redirects
- Performance should improve via direct routing
- Security must be preserved
- Redirects should not override admin-configured routes

So kernel updates only:

- nexthop exceptions
- dst cache entries

Instead of actual routing table.

## **★ If you want next:**

I can explain:

- How FIB exceptions work (fnhe tables)
- How PMTU and redirect exceptions differ
- How redirect interacts with ARP
- How forwarding + redirect + nexthop ties into \_\_mkroute\_input()

Just tell me “**explain nexthop exceptions deeper**” or anything else!



# IPV4 Caching

Friday, December 12, 2025 10:42 AM

Below is a complete, **clear + simplified explanation** of the final section you posted: **IPv4 Routing Cache**, why it existed, why it was removed, how routing lookup changed before vs after Linux 3.6, and the key terms highlighted.

## ★ IPv4 Routing Cache — What It Was, Why It Existed, Why It Was Removed

Let's break it down **theoretically and cleanly**, explaining every important phrase in the paragraph.

### ❖ What Was the IPv4 Routing Cache?

Before Linux **3.6**, the kernel kept a **routing cache**:

- A hash table that stored routing lookup results for every **flow**

(flow = src IP, dst IP, TOS)

If a packet arrived with the same source, destination, TOS as a previous one:

- ✓ The kernel could skip full routing-table lookup
- ✓ It would directly return the cached routing result
- ✓ This made routing *very fast* for repeated flows

### ★ Why a Routing Cache Was Considered Useful

Routing table lookups for millions of packets can be expensive.

So a cache allowed:

- ⌚ Fast lookups
- ⌚ The route carried extra data (PMTU, redirects, MSS info)
- ⌚ Higher performance on servers handling repeated traffic

! BUT there was a huge problem...

### ⚠ Major Security Issue — EASY DoS Attack

Because the cache created **one entry for every unique flow**, a malicious attacker could:

- send packets to many random IP addresses
- kernel creates thousands of new cache entries
- memory fills up
- CPU burns maintaining garbage collector

**Result: easy Denial of Service (DoS).**

This was the *main reason* the routing cache was removed.

## ★ What Replaced the Routing Cache? — The FIB TRIE

Linux switched to **FIB TRIE (LC-Trie)** permanently in kernel **3.6**.

FIB TRIE features:

| Feature                     | Meaning                                                    |
|-----------------------------|------------------------------------------------------------|
| <b>Longest prefix match</b> | Fast lookups for even large routing tables                 |
| <b>Resistant to DoS</b>     | No per-flow caching → attacker cannot explode memory usage |
| <b>Better scalability</b>   | Good for huge routing tables (ISP routers)                 |

This made the old routing cache unnecessary.

## ★ Important Terms From the Paragraph Explained

### ◊ Routing Cache

A per-flow cache storing:

- next-hop
- neighbor entry
- PMTU
- redirect information
- MSS values

**Existed before Linux 3.6**

**Removed due to DoS vulnerability**

### ◊ Garbage Collector

A kernel component that:

- regularly purged old routing cache entries
- tried to free memory when cache was too large

This GC was both:

- ✓ **asynchronous** (timer-based)
- ✓ **synchronous** (triggered when tables grew too big)

### ◊ FIB TRIE / LC-Trie

A data structure for routing table lookup.

It performs:

- ✓ Longest Prefix Match
- ✓ Fast lookup

- ✓ Good performance under large routing tables
- ✓ No per-flow caching

This allowed removal of the routing cache.

### ◊ TRASH Data Structure

(mentioned as reference)

A hybrid of **Trie + Hash** proposed by Olsson & Nilsson.

Kernel routing subsystem uses LC-Trie based on similar principles.

### ◊ Policy Routing

If enabled, kernel can have **up to 255 routing tables**.

Important point:

- Even when many routing tables existed, **there was still only ONE routing cache**, not 255.

### ◊ Multipath Routing Cache

Earlier Linux had caching support for multipath routing too.

Removed in **2.6.23** because:

- complex
- rarely worked right
- experimental
- caused inconsistencies

## ★ Before vs After Linux 3.6 — Routing Lookup Flow

### ◊ Before ( $\leq 3.5$ )

packet arrives

↓

check routing cache

↓ hit? → use cached route

↓ miss? → lookup main table, then cache result

### ◊ After ( $\geq 3.6$ )

packet arrives

↓

directly lookup FIB TRIE

↓

use result (no caching)

No per-flow routing cache exists anymore.

## ★ Why Removal Was Good for Modern Systems

- ✓ Security improved
- ✓ Routing more predictable
- ✓ Better scaling for big route tables
- ✓ Less kernel memory pressure
- ✓ Removed entire class of DoS attacks

Linux routing became **cleaner and more robust**.

## ⌚ Simple One-Line Summary

The IPv4 routing cache was removed because it allowed easy DoS attacks and modern FIB TRIE lookup made caching unnecessary.

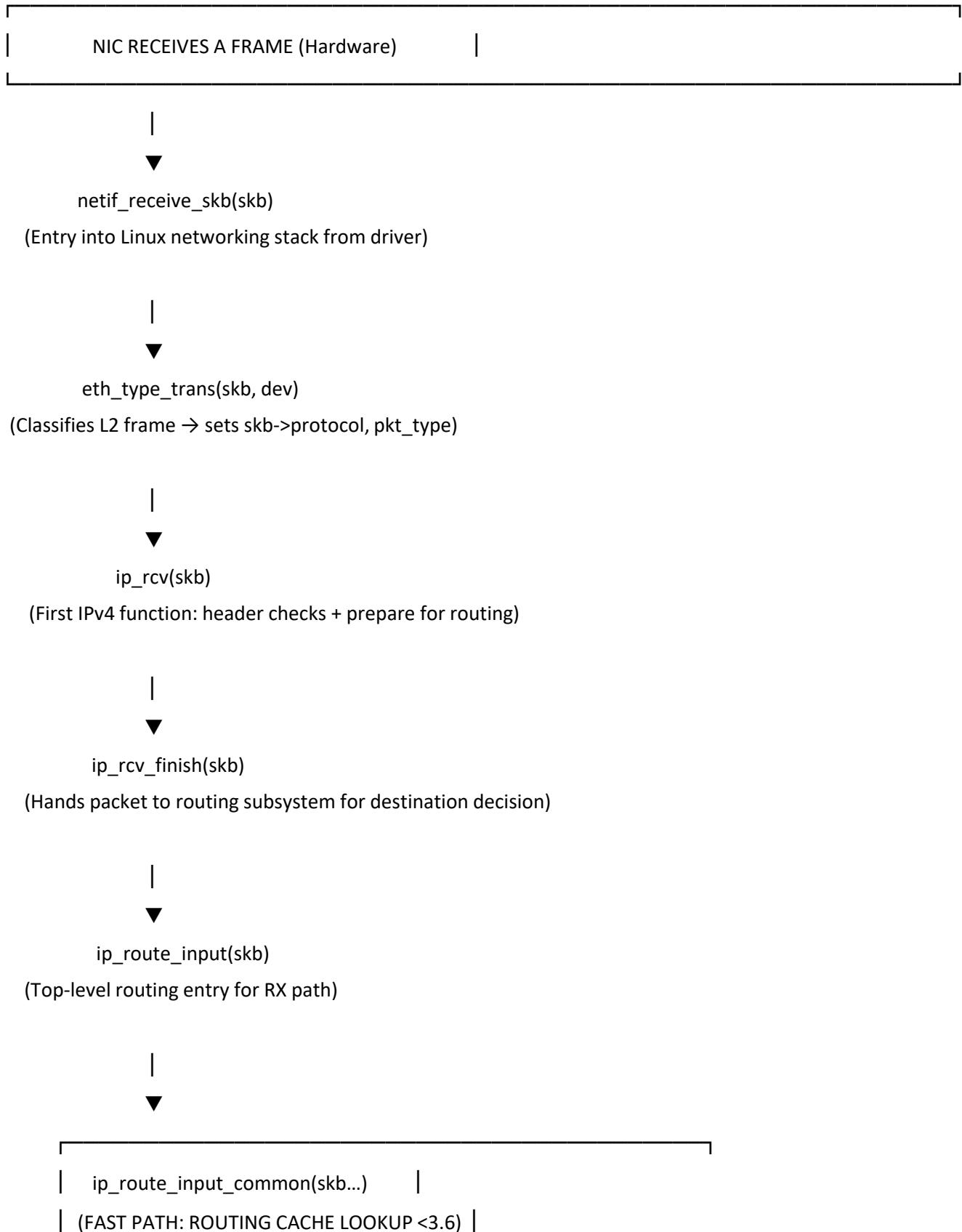
If you want, I can now create:

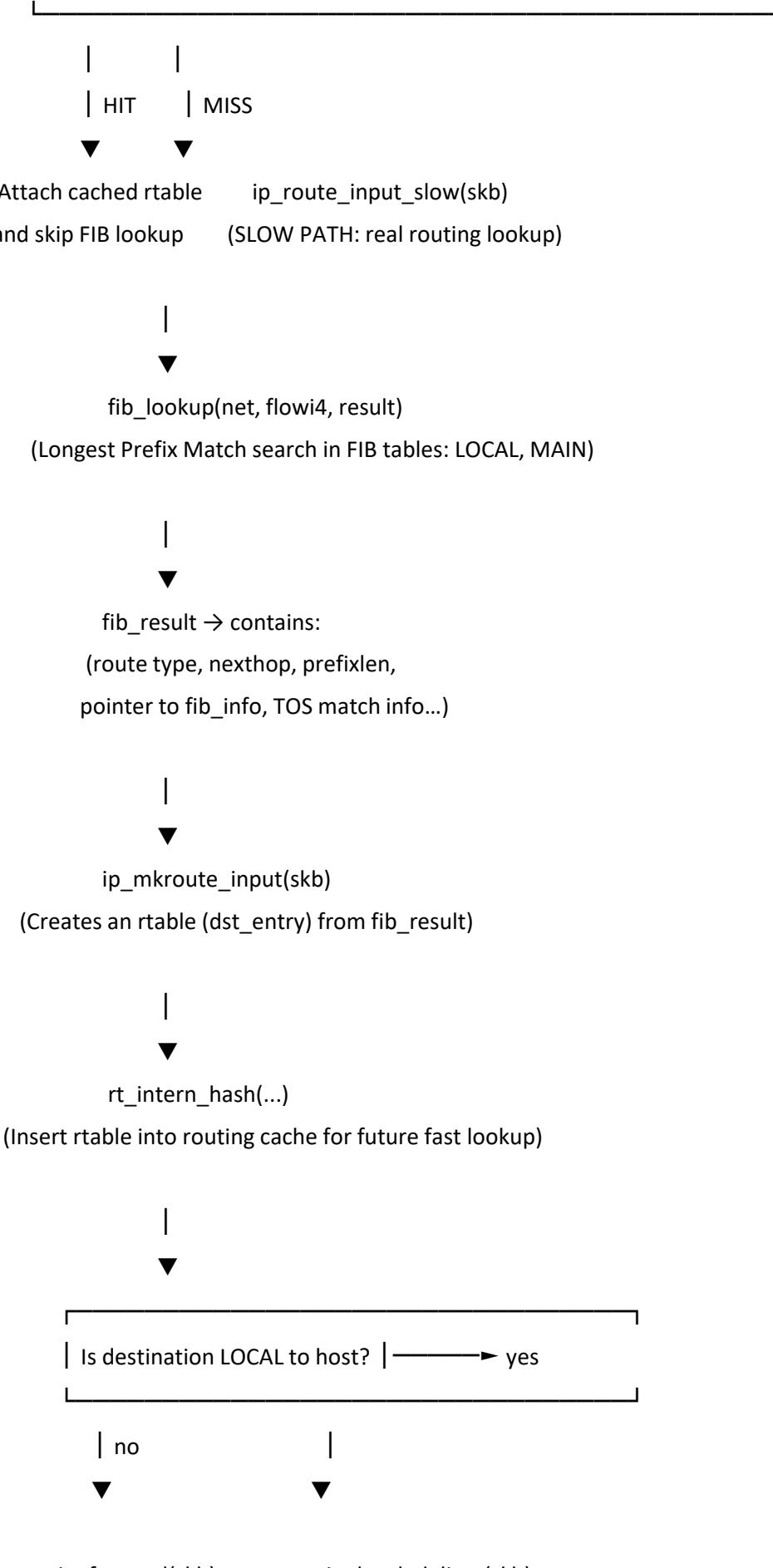
- ⌚ A full diagram of the routing lookup path
- ⌚ A map connecting fib\_table → fib\_alias → fib\_info → fib\_nh → rtable
- ⌚ A side-by-side comparison of IPv4 routing before and after Linux 3.6
- ⌚ A full routing subsystem flowchart

Just tell me “**give diagram**” or what you want next.

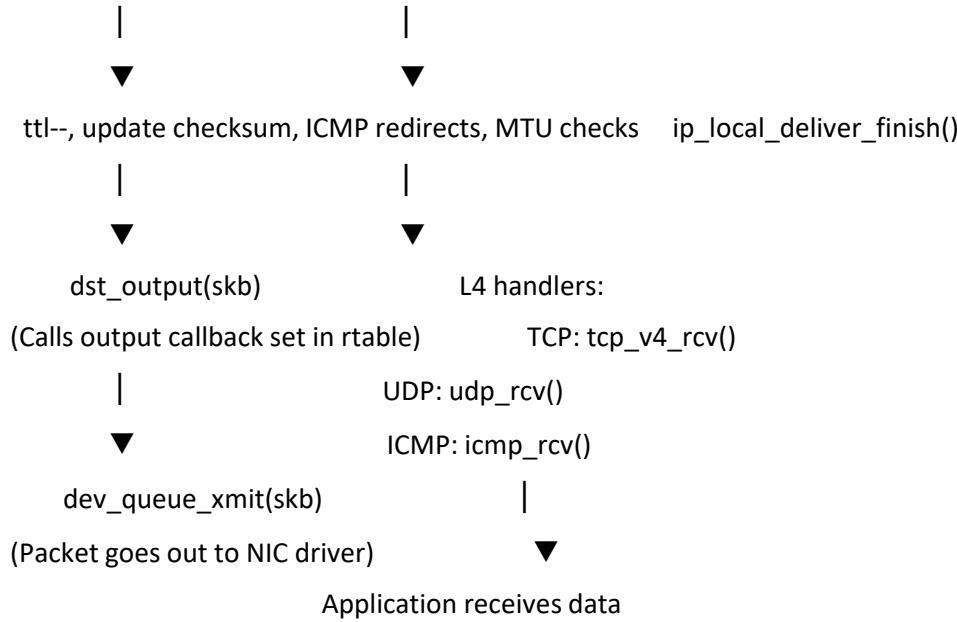
# RX PATH

Friday, December 12, 2025 11:55 AM





(Packet must be forwarded to another host) (Packet belongs to this host)



| Function                         | Layer   | What it does (one line)                                              |
|----------------------------------|---------|----------------------------------------------------------------------|
| <code>netif_receive_skb()</code> | Ingress | Entry into networking stack from NIC driver.                         |
| <code>eth_type_trans()</code>    | L2      | Parses Ethernet header, sets skb protocol + pkt type.                |
| <code>ip_rcv()</code>            | L3      | Validates IPv4 header, prepares packet for routing.                  |
| <code>ip_rcv_finish()</code>     | L3      | Hands packet to routing subsystem ( <code>ip_route_input()</code> ). |

| Function                             | What it does                                                                               |
|--------------------------------------|--------------------------------------------------------------------------------------------|
| <code>ip_route_input()</code>        | Top-level RX routing function.                                                             |
| <code>ip_route_input_common()</code> | <b>FAST PATH:</b> routing cache lookup (<3.6 kernels).                                     |
| <code>ip_route_input_slow()</code>   | <b>SLOW PATH:</b> full FIB table lookup. Calls <code>fib_lookup()</code> .                 |
| <code>fib_lookup()</code>            | Performs <b>Longest Prefix Match</b> on routing tables. Produces <code>fib_result</code> . |
| <code>ip_mkroute_input()</code>      | Builds rtable ( <code>dst_entry</code> ) from lookup result.                               |
| <code>rt_intern_hash()</code>        | Inserts route in routing cache (old kernels).                                              |

| Destination                      | Function                        | Description                       |
|----------------------------------|---------------------------------|-----------------------------------|
| Local IP (machine's own address) | <code>ip_local_deliver()</code> | Packet is for us → send up to L4. |
| Remote host (not for us)         | <code>ip_forward()</code>       | Forward packet as router.         |

| Function | What it does |
|----------|--------------|
|          |              |

|                                        |                                    |
|----------------------------------------|------------------------------------|
| <code>ip_local_deliver_finish()</code> | Calls correct L4 protocol handler. |
| <code>tcp_v4_rcv()</code>              | Handles TCP packet.                |
| <code>udp_rcv()</code>                 | Handles UDP packet.                |
| <code>icmp_rcv()</code>                | Handles ICMP.                      |

| Function                      | What it does                                                     |
|-------------------------------|------------------------------------------------------------------|
| <code>ip_forward()</code>     | Updates TTL, sends redirects, handles fragmentation logic.       |
| <code>dst_output()</code>     | Calls rtable->output function (usually <code>ip_output</code> ). |
| <code>dev_queue_xmit()</code> | Sends packet out via NIC driver.                                 |

# Netfilter

Friday, December 12, 2025 12:24 PM

Alright — let's strip this down to a **pure theoretical explanation**, no commands, no code, no implementation details.

Just **concepts, roles, and packet logic**.

## 1 What Netfilter *is* (theoretical view)

**Netfilter is a policy-enforcement framework inside the Linux kernel's networking subsystem.**

Its theoretical purpose is to **control the life of a packet** as it moves through the system.

At every important transition point, Netfilter asks:

*“Should this packet continue, be modified, be delayed, or be destroyed?”*

## 2 Why Netfilter exists (problem it solves)

Without Netfilter, the kernel would:

- Accept all packets addressed to it
- Forward all routable packets
- Send all locally generated packets unconditionally

This is **not acceptable** for:

- Security (firewalls)
- Address translation (NAT)
- Traffic shaping and inspection
- Stateful decision making

Netfilter introduces **decision points** where policies can be applied.

## 3 Core theoretical idea: *Hooks*

Netfilter is built on the idea of **hooks**.

## What is a hook?

A **hook** is a predefined checkpoint in the packet's journey.

At a hook:

- The packet pauses
- Rules are consulted
- A decision (verdict) is returned
- The packet either continues or stops

Hooks are **logical points**, not physical devices.

## 4 The packet life cycle (abstract)

Every packet in Linux follows one of these **three abstract paths**:

### 1. Incoming packet for this machine

Arrive → Inspect → Deliver to process

### 2. Incoming packet to be forwarded

Arrive → Inspect → Route → Send out

### 3. Packet created locally

Create → Inspect → Send out

Netfilter places hooks along **all three paths**.

## 5 The five Netfilter hook stages (conceptual)

| Hook Stage  | Conceptual meaning                           |
|-------------|----------------------------------------------|
| PREROUTING  | Packet just arrived; destination not decided |
| INPUT       | Packet is meant for this system              |
| FORWARD     | Packet is passing through this system        |
| OUTPUT      | Packet created by this system                |
| POSTROUTING | Packet is about to leave the system          |

These stages represent **logical states**, not code order.

## **6** What decisions can be made (verdict theory)

At any hook, a rule returns a **verdict**.

Conceptual verdicts:

- **Accept** → allow packet to continue
- **Drop** → silently destroy packet
- **Reject** → destroy packet and notify sender
- **Queue** → hand packet to another decision-maker
- **Steal** → remove packet from normal flow

These decisions define **packet destiny**.

## **7** Stateless vs Stateful filtering (theory)

Stateless filtering

Decisions are based on:

- Source
- Destination
- Protocol
- Port

Each packet is treated **independently**.

Stateful filtering

Decisions consider:

- Past packets
- Connection history
- Protocol context

Netfilter supports **stateful logic**, which allows:

- “Allow replies to connections I initiated”
- “Drop packets that do not belong to a valid flow”

## **8** Connection tracking (abstract concept)

Connection tracking introduces **memory** into packet handling.

Instead of asking:

“Is this packet allowed?”

Netfilter asks:

“Does this packet belong to a known conversation?”

This enables:

- Stateful firewalls
- NAT consistency
- Protocol-aware behavior

## 9 NAT as a theoretical concept

NAT (Network Address Translation) is **identity rewriting**.

Netfilter supports two logical forms:

Destination translation

- Changes *where* a packet is going

Source translation

- Changes *who* a packet appears to come from

Key theoretical point:

NAT must be **consistent across packets**, which requires state tracking.

## 10 Tables and chains (mental model)

Think of Netfilter as a **decision engine** with:

- **Tables** → categories of decisions
- **Chains** → ordered rule paths
- **Rules** → conditional logic
- **Targets** → actions

This is a **rule-processing pipeline**, not a single filter.

## 11 Separation of concerns (important theory)

Netfilter separates:

- **Mechanism** (kernel hooks)
- **Policy** (rules)

The kernel:

- Provides hooks
- Executes verdicts

Userspace:

- Defines policy
- Chooses rules

This separation is why Linux firewalls are flexible.

## 1 2 Netfilter in the OS security model

Netfilter is part of:

- Network access control
- Boundary enforcement
- Trust segmentation

It operates:

- Before packets reach applications
- Before packets leave the machine

So it is a **kernel-level security gate**.

## 1 3 Netfilter vs other packet control models (theory)

| Model                  | Characteristics               |
|------------------------|-------------------------------|
| Application firewall   | Sees only application traffic |
| Driver-level filtering | Very fast, little context     |
| <b>Netfilter</b>       | Balanced: context + control   |
| Proxy-based firewall   | Deep inspection, slower       |

Netfilter is a **mid-stack policy engine**.

## 1 4 Conceptual relationship with other layers

- Below Netfilter:  
**Link layer & driver logic**
- Above Netfilter:

## Sockets & applications

Netfilter is **protocol-aware**, but **application-agnostic**.

### 1 5 One-sentence theoretical definition

**Netfilter is a kernel-level decision framework that enforces network policy by intercepting packets at well-defined stages of their lifecycle and applying rule-based verdicts, optionally using state.**

Below is a **purely theoretical, framework-level explanation** of what you wrote — expanding each point conceptually and showing **how these frameworks relate to the Netfilter subsystem**, without commands or code.

## Netfilter Frameworks – Theoretical Explanation

### 1 Netfilter as a *Subsystem*, not a single feature

Netfilter itself is **not one tool**.

It is a **kernel subsystem** that provides:

- Packet interception points
- Rule evaluation logic
- State tracking
- Packet transformation capability

On top of this subsystem, **multiple frameworks** are built, each solving a different networking problem but **reusing the same Netfilter core mechanisms**.

Think of Netfilter as an **operating system for packet decisions**, and these frameworks as **applications running on that OS**.

### 2 Core functionalities provided by Netfilter (conceptual)

#### 2.1 Packet selection (iptables)

**Packet selection** means:

Choosing *which packets* are subject to a given policy.

The selection is based on:

- Packet headers
- Packet direction
- Protocol metadata
- Connection state

Theoretically:

- Netfilter provides the *hook*
- A selection engine matches packets against conditions
- Matching packets trigger actions

This selection logic is what enables **policy-based networking**.

## 2.2 Packet filtering

**Packet filtering** is the act of deciding:

- Should the packet be allowed?
- Should it be denied?
- Should it be rejected explicitly?

Theoretical role:

- Enforces **security boundaries**
- Protects the system from unwanted traffic
- Controls trust relationships between networks

Filtering is **binary decision logic**, but can be:

- Stateless (packet-by-packet)
- Stateful (flow-aware)

## 2.3 Network Address Translation (NAT)

NAT is **identity transformation**.

Conceptually:

- Packets have *identity* (source/destination)
- Netfilter can rewrite that identity
- The rewriting must be **consistent across packets**

NAT relies on:

- Hook placement (before or after routing)
- Connection tracking (to remember translations)

Without Netfilter:

- NAT would require rewriting the entire IP stack

So NAT is **not a separate system**, but a **policy applied at specific Netfilter stages**.

## 2.4 Packet mangling

Packet mangling means:

Modifying packet metadata without changing routing logic

Examples of conceptual changes:

- Marking packets for QoS
- Adjusting TTL
- Setting DSCP bits
- Altering TCP flags

Theoretical purpose:

- Influence **later decisions**
- Communicate intent between subsystems
- Prepare packets for shaping or routing policies

Packet mangling is **context enrichment**, not filtering.

## 2.5 Connection tracking

Connection tracking introduces **temporal awareness**.

Instead of viewing packets as isolated events, Netfilter can view them as:

- Part of a conversation
- Part of a protocol exchange
- Part of a logical flow

This enables:

- Stateful firewalls
- Reliable NAT

- Protocol helpers (FTP, SIP, etc.)

Theoretically:

Connection tracking turns Netfilter into a **state machine**.

## 2.6 Gathering network statistics

Netfilter can observe:

- Packet counts
- Byte counts
- Flow lifetimes
- Decision outcomes

This data is used for:

- Monitoring
- Auditing
- Capacity planning
- Intrusion detection

Netfilter therefore acts as an **observation layer** in addition to enforcement.

## ③ Frameworks built on top of Netfilter

Now let's look at the **frameworks you mentioned**, conceptually.

### ④ IPVS (IP Virtual Server)

What problem does IPVS solve?

**Load balancing at the transport layer (L4).**

IPVS distributes incoming connections across multiple backend servers while presenting:

- A single virtual service
- A single service IP

### Why IPVS uses Netfilter

IPVS needs to:

- Intercept packets early
- Make routing decisions per connection
- Rewrite packet headers
- Maintain connection state

Netfilter provides:

- Hook points for interception
- Connection tracking integration
- Packet rewriting support

Thus, IPVS is:

**A specialized Netfilter consumer focused on load balancing.**

### Conceptual model of IPVS

Client → Virtual IP



Decision Engine



Backend Server A / B / C

Key idea:

- The decision is made **once per connection**
- All future packets follow that decision

This is why IPVS heavily depends on **connection tracking**.

### IPv4 vs IPv6 support (theory)

- IPv4 support existed early because:
- NAT and load balancing were already common
- IPv6 support arrived later because:
- IPv6 adoption was slower
- Address scarcity (a key NAT motivation) is reduced in IPv6

Still, the **decision logic remains identical**.

## 5 IP sets

What problem do IP sets solve?

**Efficiency and scalability of packet matching.**

Instead of writing rules like:

- “Match this IP”
- “Match that IP”
- “Match thousands of IPs...”

IP sets allow:

Treating *groups of addresses* as a single logical object.

Conceptual definition of an IP set

An IP set is:

- A **collection**
- Of IP addresses, networks, ports, or tuples
- With fast membership testing

In theory:

IP sets convert linear rule matching into **set membership checks**.

Why IP sets exist

Without IP sets:

- Rule evaluation time grows with number of rules
- Performance degrades quickly

With IP sets:

- Decision becomes:  
“Is this packet’s address a member of this set?”

This is **algorithmic optimization**, not new policy.

Relationship to Netfilter

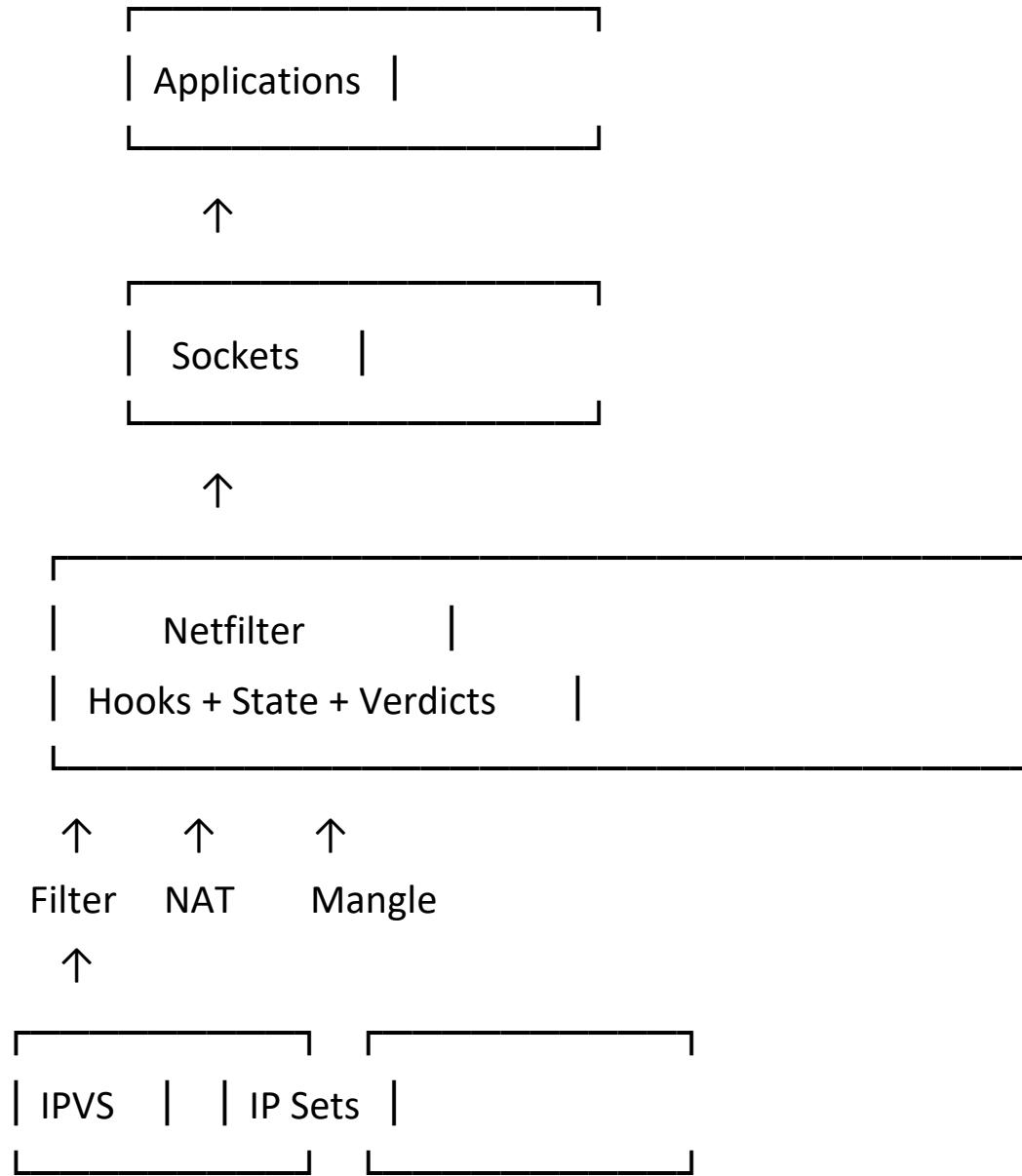
- Netfilter provides the hooks
- IP sets provide **data structures**

- Filtering logic queries these structures

So IP sets are:

A **performance extension** to Netfilter's rule engine.

## 6 How all these frameworks fit together (big picture)



Netfilter is the **foundation**, frameworks are **specialized consumers**.

## 7 One-paragraph summary (exam-ready)

The Netfilter subsystem is a kernel-level framework that enables packet

interception, classification, state tracking, modification, and decision-making at well-defined stages of packet processing. On top of this subsystem, several frameworks are built, such as packet filtering, NAT, and connection tracking, as well as higher-level systems like IPVS for transport-layer load balancing and IP sets for efficient large-scale packet matching. These frameworks reuse Netfilter's hooks and state mechanisms while addressing distinct networking requirements such as security, scalability, and traffic distribution.

Great — we'll now do this **systematically and very deeply, one hook at a time**, with **layers, exact purpose, what the kernel knows at that moment, what can / cannot be done**, and **all common “why” questions answered**.

Read this like a reference.

We'll start with the first hook and move in order.

## 1 NF\_INET\_PRE\_ROUTING

(First hook – Incoming packets, before routing)

Where it is in the stack

OSI layer

- **Layer 3 (Network layer)**  
Kernel functions
  - IPv4: ip\_rcv()
  - IPv6: ipv6\_rcv()

This is the **first point where Netfilter sees an IP packet**.

What has already happened before this hook

- ✓ Ethernet frame received by NIC
- ✓ Driver allocated sk\_buff
- ✓ L2 header processed
- ✓ IP header parsed and validated

So at this point:

- Packet is confirmed IPv4 or IPv6
- Packet is now an IP packet inside the kernel

### What has NOT happened yet

- ✗ Routing decision
- ✗ Local vs forward decision
- ✗ Outgoing interface selection

Kernel **does not yet know where the packet will go.**

### Why this hook exists (core idea)

Some decisions **must be made before routing.**

Routing depends on:

- Destination IP
- Packet marks

If you want to change those, you must do it **before routing happens.**

### What is typically done here

#### 1. Destination NAT (DNAT)

#### Why here?

Because routing uses the destination IP.

If you change the destination IP **after routing**, the routing decision would be wrong.

#### 2. Connection tracking (start)

The very first packet of a flow:

- Must be recorded
- Must be tracked from the beginning

Conntrack attaches here so it sees:

- Original source

- Original destination

### 3. Early packet dropping

Examples:

- Invalid packets
- Bogus headers
- Blacklisted sources

Dropping early:

- Saves CPU
- Avoids useless routing work

### 4. Packet marking

Marks applied here can:

- Influence routing tables
- Influence policy routing

#### What CANNOT be done here (important)

- ✗ Cannot filter based on outgoing interface
- ✗ Cannot know if packet is LOCAL or FORWARD
- ✗ Cannot do SNAT

Why?

Because routing has not happened yet.

#### Typical questions answered

**Q: Why is PREROUTING before routing?**

Because it literally means “before routing”.

**Q: Do local packets go through PREROUTING?**

- ✗ No. Only packets arriving from NIC.

#### One-line definition

**NF\_INET\_PRE\_ROUTING** is the first IP-layer hook for incoming packets, executed before any routing decision is made.

## ② NF\_INET\_LOCAL\_IN

(Incoming packets destined to this host)

Where it is in the stack

OSI layer

- Between Layer 3 and Layer 4
- Kernel functions
- IPv4: ip\_local\_deliver()
- IPv6: ip6\_input()

When this hook runs

- ✓ Packet came from PREROUTING
- ✓ Routing decision completed
- ✓ Destination IP belongs to local host

Only packets **meant for this system** reach here.

What the kernel knows at this point

- Incoming interface
- Packet is LOCAL
- Routing information complete
- Connection state known

Why this hook exists

This is the **last line of defense** before:

- TCP
- UDP
- ICMP
- Applications

Once packet passes this hook, it reaches sockets.

### What is typically done here

#### 1. INPUT firewall rules

Block or allow:

- SSH
- HTTP
- ICMP

This protects services.

#### 2. Stateful filtering

Allow:

- ESTABLISHED
- RELATED

Block:

- Unexpected NEW packets

#### 3. Rate limiting

Protect system from:

- Floods
- DoS attacks

#### 4. Logging

Audit:

- Who tried to access what

### What CANNOT be done here

- ✗ Cannot reroute packet
- ✗ Cannot forward packet
- ✗ Cannot change destination for routing

Routing already happened.

### Typical questions answered

**Q: Why is INPUT filtering here and not earlier?**

Because services live above this point.

**Q: Does forwarded traffic reach LOCAL\_IN?**

✗ No.

### One-line definition

**NF\_INET\_LOCAL\_IN is the hook for packets that are delivered to the local host after routing.**

### ③ NF\_INET\_FORWARD

(Packets passing through the system)

#### Where it is in the stack

OSI layer

- **Layer 3 (Routing path)**  
Kernel functions
- IPv4: ip\_forward()
- IPv6: ip6\_forward()

#### When this hook runs

- ✓ Packet arrived from PREROUTING
- ✓ Routing decision completed
- ✓ Packet is NOT for local host

System is acting as:

- Router
- Gateway
- Firewall

## What the kernel knows here

- Incoming interface
- Outgoing interface
- Routing complete
- Packet is transit traffic

## Why this hook exists

Forwarded packets:

- Never reach applications
- Still must be controlled

This hook enables:

- Firewalling routers
- NAT gateways
- Network segmentation

## What is typically done here

### 1. FORWARD firewall rules

Allow or block:

- Traffic between networks
- VLANs
- Subnets

### 2. Stateful forwarding

Allow return traffic for:

- Established connections

### 3. Packet marking

Used for:

- QoS
- Policy routing

## What CANNOT be done here

- ✗ Cannot deliver packet locally
- ✗ Cannot change routing decision

## Typical questions answered

**Q: Why is FORWARD separate from INPUT?**

Because forwarded traffic is not local traffic.

**Q: Do all systems use FORWARD?**

- ✗ No. Only routers.

## One-line definition

**NF\_INET\_FORWARD is the hook for packets being routed through the system.**

④ NF\_INET\_LOCAL\_OUT

**(Packets created locally)**

## Where it is in the stack

OSI layer

- **Layer 3**  
Kernel functions
- IPv4: \_\_ip\_local\_out()
- IPv6: \_\_ip6\_local\_out()

## When this hook runs

- ✓ Packet created by a local process
- ✓ Before routing decision

Examples:

- curl

- ping
- ssh client

### What the kernel knows here

- Packet is local
- Socket information available
- Outgoing interface may not be finalized

### Why this hook exists

Locally generated packets:

- Do not go through PREROUTING
- Need their own control point

### What is typically done here

1. OUTPUT firewall rules

Control:

- What programs can send traffic
- Prevent data exfiltration

2. Local DNAT

Redirect local traffic:

- Example: localhost → proxy

3. Packet marking

Influence routing decisions.

### What CANNOT be done here

- SNAT (needs outgoing interface)
- Filtering based on egress interface

## Typical questions answered

**Q: Why is LOCAL\_OUT needed?**

Because local packets skip RX path.

## One-line definition

**NF\_INET\_LOCAL\_OUT** is the hook for packets generated by the local host before routing.

### 5 NF\_INET\_POST\_ROUTING

(Last hook before packet leaves)

Where it is in the stack

OSI layer

- Layer 3 → Layer 2 boundary  
Kernel functions
- IPv4: ip\_output()
- IPv6: ip6\_finish\_output2()

When this hook runs

- ✓ Routing completed
- ✓ Outgoing interface known
- ✓ Applies to forwarded AND local packets

Why this hook exists

Some actions require knowing:

- Which interface packet leaves on
- Final source address

What is typically done here

1. Source NAT (SNAT)

Change source IP:

- Based on outgoing interface

## 2. MASQUERADE

Dynamic SNAT for:

- DHCP interfaces

## 3. Final packet marking

Used for:

- Traffic shaping

**What CANNOT be done here**

✗ Destination NAT

✗ Routing changes

**Typical questions answered**

**Q: Why SNAT is here?**

Because interface is now known.

**One-line definition**

**NF\_INET\_POST\_ROUTING** is the final IP-layer hook before a packet is sent to the network.



## FINAL LOCK-IN SUMMARY (MEMORIZE)

| Hook         | Meaning                |
|--------------|------------------------|
| PRE_ROUTING  | Before routing         |
| LOCAL_IN     | Packet for this host   |
| FORWARD      | Packet passing through |
| LOCAL_OUT    | Packet from this host  |
| POST_ROUTING | Before leaving         |

If you want next, I can:

- Draw **exact function call graphs**
- Show **iptables table ↔ hook mapping**
- Explain **conntrack timing**
- Compare **Netfilter vs XDP vs TC**

Just tell me 

# NF HOOK

Monday, December 15, 2025 1:51 PM

Good, this is an **important internal concept**.

I'll explain **NF\_HOOK** slowly, line-by-line, concept-by-concept, and I'll answer all the natural "why / when / what happens next" questions.

No shortcuts here.

## 1 What NF\_HOOK REALLY is (high-level meaning)

**NF\_HOOK** is the exact point where the Linux network stack hands a packet to Netfilter.

Think of it as:

"Kernel says: I've reached a Netfilter checkpoint — now ask Netfilter what to do with this packet."

Every Netfilter hook you studied earlier (PREROUTING, LOCAL\_IN, etc.) is reached **only because NF\_HOOK()** is called at that location in the code.

## 2 Where NF\_HOOK sits in packet flow

Example (IPv4 RX path):

Driver

↓

netif\_receive\_skb()

↓

ip\_rcv()

↓

NF\_HOOK(NFPROTO\_IPV4, NF\_INET\_PRE\_ROUTING, ...)

↓

ip\_rcv\_finish()

So:

⌚ Hooks do not magically exist

⌚ They exist because kernel code explicitly calls **NF\_HOOK()**

## 3 The macro itself (what it expands to)

You showed:

```
static inline int NF_HOOK(uint8_t pf, unsigned int hook,  
                         struct sk_buff *skb,
```

```

    struct net_device *in,
    struct net_device *out,
    int (*okfn)(struct sk_buff *))

{
    return NF_HOOK_THRESH(pf, hook, skb, in, out, okfn, INT_MIN);
}

```

Important observations:

1. NF\_HOOK is **inline**
2. It forwards everything to NF\_HOOK\_THRESH
3. INT\_MIN means **lowest possible priority**

So conceptually:

**NF\_HOOK = “run all registered Netfilter hooks for this hook point”**

## 4 Why this function exists at all

Without NF\_HOOK:

- No firewall
- No NAT
- No conntrack
- No iptables / nftables
- No packet filtering

The kernel networking stack would:

blindly route and deliver packets

So NF\_HOOK is the **bridge between**:

- core networking code
- Netfilter subsystem

## 5 Parameter-by-parameter explanation (VERY IMPORTANT)

Let's take them one by one.

### ◊ 1. pf — Protocol Family

uint8\_t pf

What it means

This tells Netfilter:

**“What kind of packet is this?”**

Common values:

- NFPROTO\_IPV4
- NFPROTO\_IPV6

- NFPROTO\_ARP
- NFPROTO\_BRIDGE

### Why this is needed

Because:

- IPv4 hooks ≠ IPv6 hooks
- Different rule engines
- Different header parsing

Netfilter internally maintains **separate hook lists per protocol family**.

So this parameter selects **which hook list to run**.

### Typical question answered

**Q: Why not detect protocol from skb?**

A: Because Netfilter is generic; protocol dispatch is explicit and faster.

### ◊ 2. hook — Which hook point

unsigned int hook

This is one of:

- NF\_INET\_PRE\_ROUTING
- NF\_INET\_LOCAL\_IN
- NF\_INET\_FORWARD
- NF\_INET\_LOCAL\_OUT
- NF\_INET\_POST\_ROUTING

### What it means

This tells Netfilter:

**“At which logical checkpoint are we?”**

Netfilter maintains **separate hook chains per hook number**.

So:

- PREROUTING rules run only at PREROUTING
- OUTPUT rules run only at LOCAL\_OUT

### Typical question answered

**Q: Why not infer hook from function name?**

A: Because Netfilter is protocol-agnostic and modular; hooks are explicit.

### ◊ 3. skb — The packet itself

struct sk\_buff \*skb

This is the **actual packet** being processed.

Inside skb:

- Ethernet header (maybe)
- IP header
- TCP/UDP/ICMP header
- Payload
- Metadata (marks, conntrack, etc.)

**Why skb is passed**

Because Netfilter may:

- Inspect headers
- Modify headers
- Attach metadata
- Drop packet
- Clone or queue packet

**Important concept**

**Netfilter never works on raw memory — it always works on sk\_buff.**

#### ◊ 4. in — Input network device

struct net\_device \*in

**What it means**

The interface **on which the packet arrived**.

Examples:

- eth0
- wlan0
- lo

**When in is valid**

- Incoming packets → in is set
- Locally generated packets → in = NULL

**Why this matters**

Firewall rules often depend on:

- Incoming interface
- Trust boundary

Example logic:

“Drop packets coming from eth1”

## ◊ 5. out — Output network device

struct net\_device \*out

What it means

The interface **on which the packet will leave.**

When out is NULL

This is **very important.**

out = NULL when:

- Routing has NOT happened yet

Example:

- ip\_rcv() → PREROUTING

At this point:

Kernel does not know where the packet will go

When out is valid

- FORWARD
- POST\_ROUTING
- Some LOCAL\_OUT cases

Typical question answered

**Q: Why can't SNAT be done when out is NULL?**

A: Because SNAT depends on outgoing interface.

## ◊ 6. okfn — Continuation function

int (\*okfn)(struct sk\_buff \*)

This is **one of the most misunderstood parts.**

What okfn really is

okfn means:

**“What function should be called if Netfilter ACCEPTS the packet?”**

Think of it as:

“Resume normal packet processing from here.”

Example

In ip\_rcv():

- okfn points to ip\_rcv\_finish()

So flow is:

ip\_rcv()

↓

NF\_HOOK(...)

↓

if ACCEPT:

call ip\_rcv\_finish(skb)

**Why this design is used**

Because:

- Netfilter must be able to interrupt packet flow
- But kernel must know **where to continue**

So instead of hardcoding:

- Netfilter returns control via okfn

## 6 What happens INSIDE NF\_HOOK

Conceptually:

4. Kernel reaches hook point
5. Calls NF\_HOOK
6. Netfilter:
  - Finds all registered hooks for (pf, hook)
  - Sorts them by priority
  - Calls them one by one
7. Each hook returns a verdict
8. Verdict decides packet fate

## 7 Netfilter verdicts — deep explanation

These are **NOT random numbers**.

Each one has a very specific meaning.

### ◊ NF\_ACCEPT (1)

Meaning

“I’m done. Let the packet continue normally.”

**What happens next**

- Netfilter continues to next hook
- Eventually calls okfn(skb)

## Most common verdict

Yes — most packets are ACCEPTed.

### ◊ NF\_DROP (0)

#### Meaning

“Destroy the packet silently.”

#### What happens next

- skb is freed
- Packet disappears
- No error sent (unless protocol does later)

#### Typical usage

- Firewall DROP rules
- Invalid packets

### ◊ NF\_STOLEN (2)

#### Meaning

“I have taken ownership of this packet.”

#### What happens next

- Kernel MUST NOT touch skb again
- Hook is responsible for skb lifetime

#### Used when

- Packet is re-injected later
- Packet redirected
- Packet consumed internally

#### Very important rule

If you return NF\_STOLEN, **you must eventually free or reinject skb.**

### ◊ NF\_QUEUE (3)

#### Meaning

“Send this packet to userspace.”

#### What happens next

- Packet queued via NFQUEUE
- Userspace program inspects it
- Userspace sends verdict back

#### Used by

- IDS / IPS
- DPI engines
- Advanced firewalls

#### ◊ NF\_REPEAT (4)

##### Meaning

“Run this hook again.”

##### Why it exists

Rarely used, but useful when:

- Packet was modified
- Hook wants re-evaluation

## 8 Typical exam / interview questions (answered)

Q: Who calls NF\_HOOK?

☞ Core network stack (ip\_rcv, ip\_forward, etc.)

Q: Does Netfilter pull packets?

☞ ✗ No. Kernel pushes packets into Netfilter.

Q: Can Netfilter change packet path?

☞ ☑ Yes, via verdicts and packet modification.

Q: Is NF\_HOOK used for both IPv4 and IPv6?

☞ ☑ Yes, with different pf.

## 9 One-paragraph summary (lock this in)

NF\_HOOK is the central interface between the Linux kernel networking stack and the Netfilter subsystem. At specific points in packet traversal, the kernel invokes NF\_HOOK to allow registered Netfilter hooks to inspect, modify, accept, drop, queue, or take ownership of packets represented by sk\_buff. The parameters passed to NF\_HOOK describe the protocol family, hook location, packet, input and output devices, and the continuation function that resumes normal processing if the packet is

accepted. The hook functions return standardized verdicts that precisely control the packet's fate within the kernel.

Below are **clean, exam-ready NOTES** on **Netfilter Hook Registration**, written in **point form**, concise but **complete**, exactly the kind you revise before an exam or interview.

## Netfilter Hook Registration — NOTES

### 1. Purpose of Netfilter Hook Registration

- Netfilter is a **kernel framework**, not hard-coded logic
- Packet processing logic (firewall, NAT, conntrack) is added by **registering hooks**
- Hooks allow kernel modules to:
- Inspect packets
- Modify packets
- Decide packet fate (accept/drop/etc.)
- Registration enables **modularity and extensibility**

### 2. Where Hooks Are Invoked

- Core network stack explicitly calls NF\_HOOK()
- NF\_HOOK() hands control to Netfilter at fixed packet traversal points
- Hook points are common for IPv4 and IPv6

### 3. The nf\_hook\_ops Structure

Defined in include/linux/netfilter.h

```
struct nf_hook_ops {  
    struct list_head list; // internal use  
    nf_hookfn *hook; // callback  
    struct module *owner; // module reference  
    u_int8_t pf; // protocol family  
    unsigned int hooknum; // hook point  
    int priority; // execution order  
};
```

### 4. Fields of nf\_hook\_ops (Important)

#### 4.1 hook

- Pointer to the callback function
- Called by Netfilter when packet reaches the hook point
- Prototype:  
`unsigned int hookfn(  
 unsigned int hooknum,`

```

    struct sk_buff *skb,
    const struct net_device *in,
    const struct net_device *out,
    int (*okfn)(struct sk_buff *)
);

```

- Must return a **Netfilter verdict**

#### 4.2 owner

- Usually set to THIS\_MODULE
- Prevents module from unloading while hook is active
- Ensures safety of callback execution

#### 4.3 pf (Protocol Family)

Specifies packet type:

- NFPROTO\_IPV4
- NFPROTO\_IPV6
- NFPROTO\_ARP
- NFPROTO\_BRIDGE

Netfilter maintains **separate hook tables per protocol family**

#### 4.4 hooknum

Specifies **where** the hook is called:

- NF\_INET\_PRE\_ROUTING
- NF\_INET\_LOCAL\_IN
- NF\_INET\_FORWARD
- NF\_INET\_LOCAL\_OUT
- NF\_INET\_POST\_ROUTING

#### 4.5 priority

- Determines **execution order**
- Lower value = executed earlier
- Multiple hooks may exist at the same hook point

Defined for IPv4 in:

include/uapi/linux/netfilter\_ipv4.h

Used to enforce logical order:

- Conntrack before NAT
- NAT before firewall
- Firewall before SNAT

## 5. Netfilter Hook Registration APIs

### 5.1 Register a Single Hook

```
int nf_register_hook(struct nf_hook_ops *reg);
```

- Registers one hook
- Inserts into hook list based on priority

### 5.2 Register Multiple Hooks

```
int nf_register_hooks(struct nf_hook_ops *reg, unsigned int n);
```

- Registers an array of hooks
- Ensures atomic registration (all or none)

## 6. How Hooks Are Stored Internally

- Netfilter maintains:
- One hook list per (pf, hooknum)
- Each list:
  - Sorted by priority
  - Implemented using linked lists

## 7. Runtime Hook Execution Flow

9. Packet reaches hook point
10. Kernel calls NF\_HOOK()
11. Netfilter:
  - Selects hook list using pf and hooknum
  - Calls hooks in ascending priority
12. Each hook returns a verdict
13. Packet traversal continues only if all hooks return NF\_ACCEPT

## 8. Netfilter Verdicts

Defined in include/uapi/linux/netfilter.h

| Verdict   | Meaning                        |
|-----------|--------------------------------|
| NF_ACCEPT | Continue normal processing     |
| NF_DROP   | Drop packet silently           |
| NF_STOLEN | Hook takes ownership of packet |
| NF_QUEUE  | Send packet to userspace       |
| NF_REPEAT | Re-run hook                    |

## 9. Key Design Advantages

- Separation of **mechanism** (kernel) and **policy** (modules/userspace)
- Supports:
  - Firewalls
  - NAT

- Load balancing
- IDS/IPS
- No modification needed in core networking code

## 10. Frequently Asked Questions (Exam Focus)

- Q: Who calls Netfilter hooks?  
A: Core networking code via NF\_HOOK()
- Q: Can multiple hooks exist at one hook point?  
A: Yes, executed in priority order
- Q: What stops packet traversal?  
A: Any hook returning NF\_DROP, NF\_STOLEN, or NF\_QUEUE
- Q: Why priorities are needed?  
A: To enforce correct packet processing order

## 11. One-Line Summary

Netfilter hook registration allows kernel modules to attach packet processing callbacks at well-defined points in the network stack, enabling flexible, ordered, and modular packet filtering and manipulation.

If you want, next I can give:

- **One-page revision sheet**
- **Hook priority table**
- **Packet flow + hook interaction notes**
- **Conntrack + NAT hook order notes**

Just tell me 

# Connection tracking

Monday, December 15, 2025 3:44 PM

Below are **clean, exam-ready, theory-only notes on Connection Tracking (conntrack)**.

No emojis, no commands, no examples tied to tools — only **what, why, how, where, and design logic**.

## Connection Tracking (Conntrack) — Detailed Theoretical Notes

### 1. What is Connection Tracking?

**Connection Tracking is a kernel mechanism that tracks the state and identity of network flows as they traverse the Linux networking stack.**

Instead of treating each packet independently, the kernel groups packets into **logical connections (flows)** and remembers information about them.

A *connection* here does **not** necessarily mean a TCP connection only; it is a **generalized concept** that applies to:

- TCP
- UDP
- ICMP
- Other IP-based protocols

### 2. Why Connection Tracking is needed

#### 2.1 Limitation of stateless packet processing

Without connection tracking:

- Each packet is processed in isolation
- The kernel cannot know if a packet is a reply or a new request
- Firewalls must allow or block packets blindly

This causes problems:

- Return packets may be blocked
- NAT cannot function reliably
- Protocol semantics are ignored

#### 2.2 Problems conntrack solves

Connection tracking enables:

1. **Stateful firewalls**
  - Allow reply packets automatically
  - Block unsolicited traffic
2. **Network Address Translation (NAT)**
  - Maintain consistent address/port rewriting
  - Map replies back to original endpoints
3. **Protocol awareness**
  - Understand TCP handshakes

- Track UDP “pseudo-connections”
  - Handle ICMP errors correctly
4. **Security**
- Drop invalid packets
  - Detect abnormal traffic behavior

### 3. Conceptual definition of a connection

A connection is identified by a **tuple of packet attributes**.

#### Core tuple elements

- Source IP address
- Destination IP address
- Source port (if applicable)
- Destination port (if applicable)
- Layer-4 protocol (TCP, UDP, ICMP, etc.)

This tuple uniquely identifies a flow **in one direction**.

Conntrack internally stores **both directions** of a connection:

- Original direction
- Reply direction

### 4. Where Connection Tracking lives in the kernel

Connection tracking is part of the **Netfilter subsystem**.

It is:

- Implemented entirely in the kernel
- Independent of iptables or nftables (those only use it)

Conntrack logic is invoked at **Netfilter hook points**, primarily:

- PRE\_ROUTING
- LOCAL\_OUT

These are the earliest points where packets can be reliably identified.

### 5. How Connection Tracking works (high-level flow)

5. Packet enters the network stack
6. Netfilter hook is reached
7. Conntrack subsystem:
  - Extracts the tuple from the packet
  - Looks up an existing connection entry
8. One of two cases occurs:
  - Entry exists → packet is associated with that connection
  - Entry does not exist → a new connection entry may be created
9. Packet is tagged with connection metadata
10. Subsequent hooks and subsystems use this metadata

### 6. Connection states (conceptual)

Conntrack assigns a **state** to packets based on their relationship to tracked connections.

## Common connection states

### *NEW*

- First packet of a connection
- No existing entry was found
- May result in creation of a new connection entry

### *ESTABLISHED*

- Packet belongs to an existing connection
- Seen traffic in both directions

### *RELATED*

- Packet is not part of the main flow
- Logically associated with an existing connection
- Common with control protocols and error messages

### *INVALID*

- Packet cannot be associated with any known connection
- Malformed or unexpected traffic

These states allow **policy decisions based on flow history**.

## 7. Protocol-specific behavior

### 7.1 TCP connection tracking

TCP is connection-oriented, so conntrack:

- Tracks TCP handshake states
- Observes SYN, ACK, FIN, RST flags
- Maintains a detailed state machine

This allows detection of:

- Invalid TCP sequences
- Half-open connections
- Unexpected packets

### 7.2 UDP connection tracking

UDP is stateless by design, but conntrack:

- Creates a pseudo-connection
- Uses timeouts instead of handshakes
- Tracks traffic in both directions

This enables:

- Stateful firewalling for UDP
- NAT consistency

### 7.3 ICMP connection tracking

ICMP does not use ports.

Conntrack:

- Tracks ICMP identifiers and types
- Associates error messages with existing connections
- Allows ICMP errors to pass correctly

## 8. Connection Tracking and NAT (tight coupling)

NAT depends on conntrack.

Why:

- NAT rewrites packet headers
- Replies must be mapped back correctly
- Mapping must persist across packets

Conntrack stores:

- Original tuple
- Translated tuple

Every NAT operation:

- Consults the conntrack entry
- Applies consistent translation

Without conntrack:

- NAT would break after the first packet

## 9. Conntrack data structures (conceptual)

Each connection entry stores:

- Original direction tuple
- Reply direction tuple
- Protocol-specific state
- Timeouts
- NAT translation information (if any)
- Status flags

Entries are stored in a **hash table** for fast lookup.

## 10. Lifetime and timeouts

Connections are **not permanent**.

Each protocol has:

- Specific timeout values
- Different lifetimes based on state

Examples (conceptual):

- TCP established connections live longer
- UDP connections expire quickly
- ICMP entries are short-lived

When a connection expires:

- Entry is removed
- Resources are freed

## 11. Performance considerations

Connection tracking:

- Adds memory overhead
- Adds lookup cost per packet

However:

- Hash-based lookups are fast
- Cost is justified by correctness and security

For high-performance paths:

- Conntrack can be bypassed or limited
- Early packet processing mechanisms may be used instead

## 12. Relationship with Netfilter hooks

Connection tracking is invoked:

- Before firewall decisions
- Before NAT decisions

This ordering is crucial because:

- Firewall rules rely on connection state
- NAT relies on connection identity

Thus, conntrack is a **foundational layer** upon which other Netfilter features are built.

## 13. Why connection tracking is optional but critical

Connection tracking can be disabled, but then:

- Firewalls become stateless
- NAT becomes unreliable
- Protocol handling degrades

Therefore:

- Simple routers may avoid it
- Firewalls and NAT gateways depend on it

## 14. Conceptual comparison

| Feature                | Stateless processing | Connection tracking |
|------------------------|----------------------|---------------------|
| Packet awareness       | Individual packets   | Flow-based          |
| Firewall quality       | Weak                 | Strong              |
| NAT support            | Poor                 | Required            |
| Protocol understanding | None                 | High                |

## 15. One-paragraph summary (exam-ready)

Connection tracking is a kernel-level mechanism within the Netfilter subsystem that maintains state information about network flows as packets traverse the Linux networking stack. By grouping packets into logical connections based on protocol and addressing tuples, connection tracking enables stateful

firewalls, reliable network address translation, protocol-aware filtering, and secure packet handling. It operates early in packet traversal, associates packets with existing or newly created connection entries, and provides metadata that higher-level Netfilter components use to make informed policy decisions.

If you want next, I can prepare **similar notes** on:

- Conntrack state machines
- Conntrack vs stateless filtering
- NAT internals using conntrack
- Performance and scaling limits of conntrack

Just tell me.

# IP Tables

Monday, December 15, 2025 5:22 PM

Below are **clean, exam-ready, theoretical notes about iptables**, written from scratch, explaining **what, why, and how**, with **no emojis, no commands, no examples, only concepts and internal logic**.

You can treat this as **kernel-networking notes**.

## [iptables — Theoretical Notes \(Kernel Perspective\)](#)

### [1. What iptables is \(precise definition\)](#)

**iptables is a userspace control interface for configuring the Linux kernel Netfilter subsystem.**

It does **not** process packets itself.

It does **not** sit in the packet path.

Instead, iptables:

- Constructs rule descriptions in userspace
- Transfers those rules to the kernel
- Netfilter enforces those rules during packet traversal

So architecturally:

iptables = policy manager

Netfilter = packet enforcement engine

### [2. Why iptables exists](#)

The Linux kernel networking stack is designed to be:

- Generic
- Modular
- Independent of security policy

Embedding firewall logic directly into the IP stack would:

- Make the kernel rigid
- Prevent dynamic policy changes

- Require recompilation for rule changes  
iptables solves this by:
- Separating **policy (userspace)** from **mechanism (kernel)**
- Allowing rules to be added, removed, or modified at runtime
- Avoiding any change to core networking code

This is a classic operating system design principle:  
**policy–mechanism separation.**

### 3. iptables position in the networking architecture

Layered view:

- Layer 7–4: Applications, sockets, TCP/UDP
- Layer 3: IP stack
- Layer 2: Ethernet, driver
- Hardware

iptables does **not** operate at any OSI layer directly.

Instead:

- It configures Netfilter hooks
- Those hooks operate mainly at Layer 3 (IP)
- With awareness of Layer 4 (ports, protocol)

Thus iptables is **control-plane only**.

### 4. iptables and Netfilter relationship

Netfilter provides:

- Fixed hook points in the packet path
- Hook invocation via NF\_HOOK
- Hook registration infrastructure
- Packet verdict handling

iptables provides:

- A rule syntax
- Rule grouping (tables, chains)
- Rule translation into kernel data structures

- Rule lifecycle management  
Netfilter does not know what “iptables” is.  
iptables knows exactly how to talk to Netfilter.

## 5. Why there are multiple iptables variants

Linux historically supports multiple protocol families:

- IPv4
- IPv6
- ARP
- Ethernet bridging

Each protocol has:

- Different headers
- Different semantics
- Different processing locations

Therefore, iptables was split into:

- iptables (IPv4)
- ip6tables (IPv6)
- arptables (ARP)
- ebtables (Ethernet frames)

Each variant:

- Talks to Netfilter
- Registers hooks in different protocol families
- Uses different kernel rule engines

This separation existed because:

- Protocol logic was hard-coded
- A unified engine did not exist initially

## 6. Core abstraction used by iptables

iptables organizes policy using four abstractions:

1. Tables
2. Chains
3. Rules

## 4. Targets

These abstractions exist **only for humans and userspace**.

The kernel stores optimized representations.

## 7. Tables (conceptual meaning)

A **table** represents a category of decision-making.

Tables do not represent execution order globally.

They represent **purpose**.

Key tables:

- filter: access control decisions
- nat: address translation decisions
- mangle: packet modification decisions
- raw: early processing and conntrack control

Each table is consulted at **specific hook points only**.

This restriction exists because:

- Not all operations make sense at all times
- Some actions require routing information
- Some must happen before routing

## 8. Chains (conceptual meaning)

A **chain** represents a logical path of rule evaluation at a hook point.

Chains correspond to:

- Netfilter hook locations
- Or user-defined branching points

Built-in chains map directly to hooks:

- PREROUTING
- INPUT
- FORWARD
- OUTPUT
- POSTROUTING

User-defined chains exist to:

- Reduce rule duplication
- Improve readability
- Create logical grouping

Chains are not functions.

They are **ordered rule lists**.

## 9. Rules (conceptual meaning)

A **rule** is a conditional statement:

- If packet matches conditions
- Then perform an action

Matching is based on:

- Packet headers
- Connection state
- Interface metadata
- Marks and flags

Rules are evaluated **sequentially within a chain**.

The first rule that produces a terminal verdict stops traversal of that chain.

## 10. Targets (conceptual meaning)

A **target** specifies what happens when a rule matches.

Targets translate to **Netfilter verdicts or actions**.

Conceptually, targets fall into categories:

- Terminating targets: stop evaluation
- Non-terminating targets: modify and continue

Internally, targets map to:

- Netfilter verdicts (ACCEPT, DROP, QUEUE)
- Or packet transformations (NAT, marking)

## 11. iptables and hook execution model

At runtime:

5. Kernel reaches a Netfilter hook
6. NF\_HOOK is called
7. Netfilter identifies relevant tables for that hook
8. Chains are traversed in defined order
9. Rules are evaluated
10. Targets produce verdicts
11. Packet either continues or stops

iptables itself is **not present** in this flow.

## 12. iptables and connection tracking

iptables integrates tightly with connection tracking.

Connection tracking:

- Associates packets with flows
- Maintains state across packets

iptables rules can:

- Match based on connection state
- Enforce stateful firewall policies
- Enable NAT consistency

iptables does not implement connection tracking.

It only **uses the results** provided by Netfilter conntrack.

## 13. Why iptables rule evaluation is linear

iptables uses a sequential rule model.

This was chosen because:

- Simple mental model
- Deterministic behavior
- Easy debugging

However, this causes:

- Performance degradation with large rule sets
- $O(n)$  matching cost

This limitation directly led to nftables.

## 14. How iptables communicates with the kernel

iptables communicates using:

- setsockopt()
- getsockopt()

This mechanism:

- Transfers entire rule sets
- Requires kernel to replace structures atomically
- Is not event-driven

This design:

- Is simple
- Is robust
- Does not scale well for complex updates

## 15. Limitations of iptables (architectural)

Key limitations:

- Separate engines per protocol
- Code duplication
- Linear rule evaluation
- Rigid syntax
- Complex rule management
- Hard to extend cleanly

These are architectural, not implementation bugs.

## 16. Why nftables was introduced

nftables was created to solve iptables' architectural limitations:

- Unified rule engine
- Single protocol-agnostic VM
- Efficient rule evaluation
- Better extensibility
- Cleaner userspace–kernel interface

iptables remains widely used due to:

- Legacy systems
- Familiarity
- Stability

## 17. Important “why” questions answered

Why iptables is userspace:

Because kernel should not embed policy.

Why tables exist:

To separate decision categories.

Why chains exist:

To map rules to packet traversal paths.

Why NAT rules are restricted to certain hooks:

Because routing information availability matters.

Why iptables does not see packets directly:

Because Netfilter owns packet traversal.

Why iptables is being replaced:

Because its architecture does not scale.

## 18. One-paragraph summary (exam quality)

iptables is a userspace management framework that configures the Linux kernel Netfilter subsystem by defining packet filtering, address translation, and packet modification policies. It provides a structured rule model using tables, chains, and targets, which are translated into kernel data structures and enforced at fixed hook points in the network stack. iptables itself never processes packets; instead, it acts as a control-plane interface that allows dynamic manipulation of kernel-level packet handling behavior while maintaining a strict separation between policy and mechanism.

If you want, next I can give:

- Connection tracking notes in the same style

- NAT explained purely theoretically
- iptables vs nftables architecture comparison
- A single packet walkthrough from skb to verdict

Just say what you want next.