Network System Design

CS6100

Tutorial 03

Hardware-based Multicast Filtering

Student Details

Name: R Abinav
Roll No: ME23B1004

Date: January 31, 2026

1 Problem Statement

In high-speed networks, a host interface receives significant multicast traffic. Processing every packet in the Operating System kernel to determine relevance is computationally expensive and causes high CPU interrupt overhead.

Network Interface Cards (NICs) employ a hardware-based ”pre-filter” using hashing to discard unwanted packets before CPU interruption. This assignment simulates the hardware filtering mechanism and the complete life-cycle of a multicast packet from physical wire to final software decision.

1.1 Objectives

- **Understand CRC-32:** Implement CRC-32 hashing logic used in Ethernet standards
- **Simulate Hardware Constraints:** Model limited-size hardware hash table (16-64 entries)
- **Analyze Collisions:** Demonstrate false positives where unwanted traffic leaks through
- **Performance Metrics:** Calculate filtering ratio (hardware vs software processing)

2 Background Theory

2.1 Multicast Traffic and CPU Overhead

Network interfaces receive all packets on the wire, including multicast traffic for other hosts. Without hardware filtering, every packet triggers CPU interrupt for software processing, creating overhead from context switches and header examination. High multicast traffic can saturate CPU with interrupt handling.

2.2 Two-Stage Filtering Architecture

Modern NICs use two-stage filtering:

1. **Hardware Pre-filter:** Fast hash table on NIC drops most unwanted packets before CPU interruption
2. **Software Filter:** OS kernel maintains exact list of subscribed multicast groups for final verification

This trades some false positives for massive reduction in CPU interrupts.

2.3 CRC-32 Hash Function

CRC-32 (Cyclic Redundancy Check) is a polynomial-based hash function standardized in IEEE 802.3:

- Deterministic: same input produces same hash
- Fast: uses table lookup

- Good distribution: spreads values across hash space
- Hardware-friendly: simple bitwise operations
- Polynomial: 0xEDB88320 (reversed representation)

2.4 IP Multicast to MAC Mapping

IPv4 multicast (224.0.0.0 to 239.255.255.255) maps to Ethernet MAC per RFC 1112:

- Multicast MAC prefix: `01:00:5E`
- Lower 23 bits of IP map to lower 23 bits of MAC
- Upper 5 bits lost in mapping
- Creates inherent collisions: 32 IPs map to each MAC

Example: 224.0.0.1 → 01:00:5E:00:00:01

2.5 Hash Collisions and False Positives

Hardware hash table is small (16-64 entries) to fit in fast NIC memory, creating:

1. **IP-to-MAC collisions:** Multiple IPs map to same MAC (RFC 1112 inherent)
2. **Hash collisions:** Multiple MACs hash to same table index

Collisions cause hardware filter to pass unwanted packets (false positives). Software filter performs exact matching to drop these.

3 Implementation

3.1 CRC-32 Hash Engine

- Precomputes 256-entry lookup table using polynomial 0xEDB88320
- Computes 32-bit CRC for 6-byte MAC addresses
- Extracts hash index using upper N bits (N = table size in bits)

3.2 Hardware Hash Table

- Bit vector (each bit = one hash bucket)
- Size: 4 bits = 16 entries
- Operations: set bit on subscription, check bit on packet arrival
- Passes packet if bit set (includes false positives)

3.3 Software Filter

- Maintains exact set of subscribed IP addresses
- Final verification on packets passed by hardware
- Drops false positives (unsubscribed IPs that passed hardware)

3.4 Packet Flow

1. Packet arrives with destination IP
2. Convert IP to multicast MAC
3. Hash MAC to get table index
4. Check hardware hash table bit
5. If bit = 0: drop (no CPU interrupt)
6. If bit = 1: pass to software
7. Software checks exact IP subscription
8. Accept if subscribed, drop if false positive

4 Simulation Results

Configuration:

- Hash table: 4 bits (16 entries)
- Subscribed groups: 5 multicast IPs
- Test packets: 400 (100 subscribed, 300 unsubscribed)

4.1 Output Analysis

4.2 Key Observations

1. **Hardware Filtering Ratio: 60.00%**

   240/400 packets dropped by hardware without CPU interruption, representing 60% reduction in interrupt load.

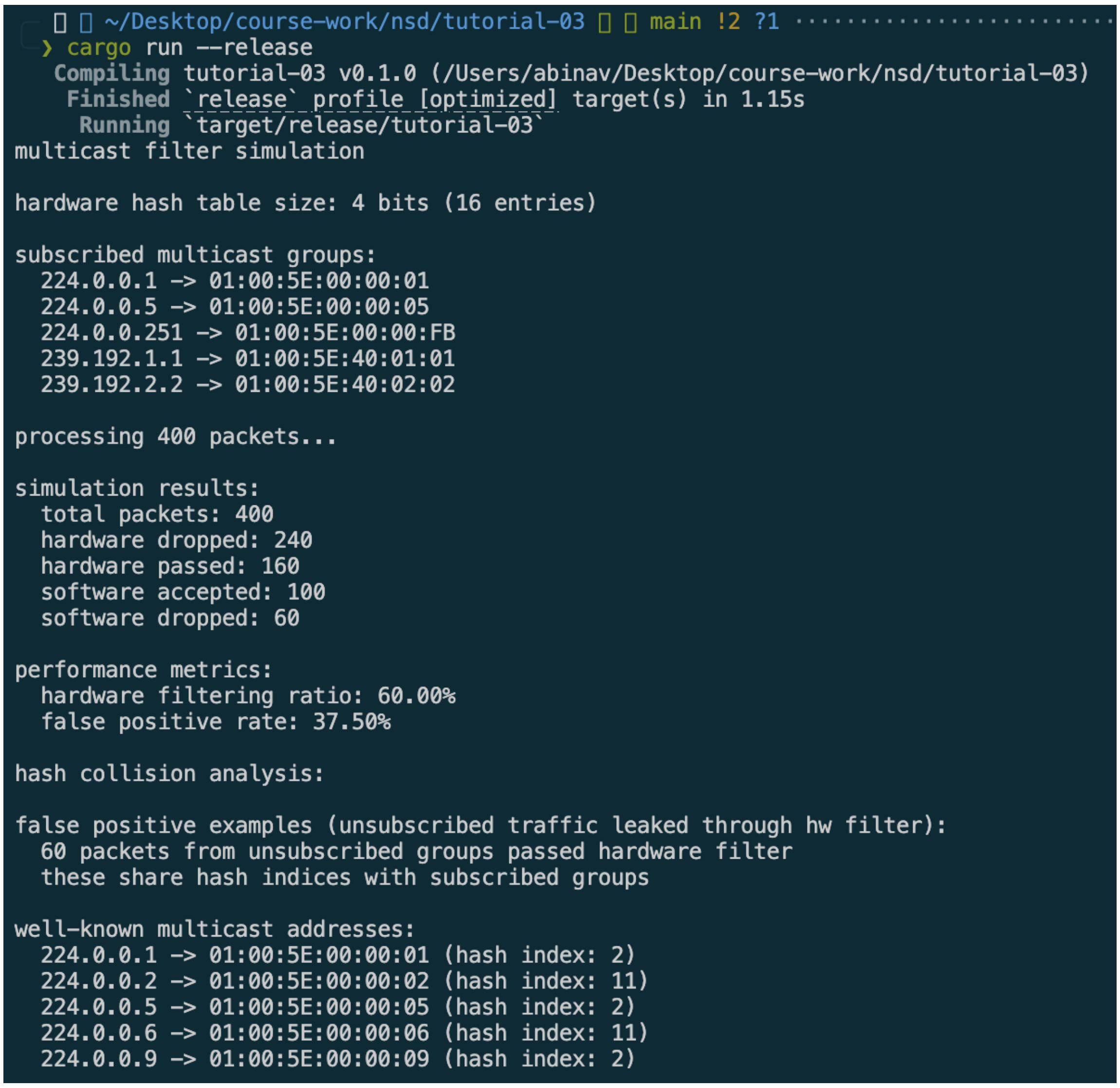
2. **False Positive Rate: 37.50%**

   60/160 packets passed hardware filter as false positives (unsubscribed traffic). These caused unnecessary CPU interrupts due to hash collisions.

3. **Hash Collision Example**

   Well-known multicast addresses show multiple IPs mapping to same hash index:

   - 224.0.0.1, 224.0.0.5, 224.0.0.9 → index 2



```
 ~/Desktop/course-work/nsd/tutorial-03   main !2 ?1 ························
❯ cargo run --release
   Compiling tutorial-03 v0.1.0 (/Users/abinav/Desktop/course-work/nsd/tutorial-03)
    Finished `release` profile [optimized] target(s) in 1.15s
     Running `target/release/tutorial-03`
multicast filter simulation

hardware hash table size: 4 bits (16 entries)

subscribed multicast groups:
  224.0.0.1 -> 01:00:5E:00:00:01
  224.0.0.5 -> 01:00:5E:00:00:05
  224.0.0.251 -> 01:00:5E:00:00:FB
  239.192.1.1 -> 01:00:5E:40:01:01
  239.192.2.2 -> 01:00:5E:40:02:02

processing 400 packets...

simulation results:
  total packets: 400
  hardware dropped: 240
  hardware passed: 160
  software accepted: 100
  software dropped: 60

performance metrics:
  hardware filtering ratio: 60.00%
  false positive rate: 37.50%

hash collision analysis:

false positive examples (unsubscribed traffic leaked through hw filter):
  60 packets from unsubscribed groups passed hardware filter
  these share hash indices with subscribed groups

well-known multicast addresses:
  224.0.0.1 -> 01:00:5E:00:00:01 (hash index: 2)
  224.0.0.2 -> 01:00:5E:00:00:02 (hash index: 11)
  224.0.0.5 -> 01:00:5E:00:00:05 (hash index: 2)
  224.0.0.6 -> 01:00:5E:00:00:06 (hash index: 11)
  224.0.0.9 -> 01:00:5E:00:00:09 (hash index: 2)
```

Figure 1: Simulation output showing filtering statistics and hash collisions

- 224.0.0.2, 224.0.0.6 → index 11

4. **Software Accuracy: 100%**

   All 100 subscribed packets correctly accepted after passing hardware filtering.

5 Trade-offs and Design Considerations

5.1 Hash Table Size

- **Smaller (4 bits):** Less hardware cost, higher collisions, more false positives
- **Larger (6-8 bits):** Lower collisions, requires more NIC memory

5.2 Performance Impact

Despite 37.5% false positive rate, the system achieves:

- 60% reduction in CPU interrupts (240 packets dropped in hardware)
- Only 60 unnecessary interrupts (false positives) vs 300 without filtering
- Net benefit: 80% reduction in unnecessary CPU load (240 vs 300)

5.3 Real-world Implementation

Production NICs typically use:

- 64-entry hash tables (6 bits) for cost-performance balance
- Perfect hash functions optimized for common multicast patterns
- Additional filtering layers (VLAN tags, port filtering)

6 Conclusion

The simulation demonstrates hardware-based multicast filtering reduces CPU overhead despite hash collisions:

1. CRC-32 provides fast, deterministic hashing for NIC hardware
2. Small hash tables create collisions but provide significant benefit
3. Two-stage architecture balances hardware cost with CPU efficiency
4. False positives are acceptable for dramatic interrupt reduction

This filtering mechanism is essential for high-speed networking, enabling hosts to handle multicast-heavy traffic without CPU saturation.

7 Source Code

```
use std::collections::{HashMap, HashSet};
use std::fmt;

//crc32 ieee 802.3
struct Crc32 {
    table: [u32; 256],
}

impl Crc32 {
    const POLYNOMIAL: u32 = 0xEDB88320;

    fn new() -> Self {
        let mut table = [0u32; 256];
        for i in 0..256 {
            let mut crc = i as u32;
            for _ in 0..8 {
                if crc & 1 != 0 {
                    crc = (crc >> 1) ^ Self::POLYNOMIAL;
                } else {
                    crc >>= 1;
                }
            }
            table[i] = crc;
        }
        Self { table }
    }

    fn compute(&self, data: &[u8]) -> u32 {
        let mut crc = 0xFFFFFFFF;
        for &byte in data {
            let index = ((crc ^ byte as u32) & 0xFF) as usize;
            crc = (crc >> 8) ^ self.table[index];
        }
        !crc
    }

    fn hash_to_index(&self, mac: &MacAddress, bits: u8) -> usize {
        let crc = self.compute(&mac.0);
        let shift = 32 - bits;
        ((crc >> shift) as usize) & ((1 << bits) - 1)
    }
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
struct MacAddress([u8; 6]);

impl MacAddress {
    fn from_multicast_ip(ip: &IpAddress) -> Self {
        let b = ip.0;
        Self([0x01, 0x00, 0x5E, b[1] & 0x7F, b[2], b[3]])
    }
}

impl fmt::Display for MacAddress {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
```

```rust
        write!(
            f,
            "{:02X}:{:02X}:{:02X}:{:02X}:{:02X}:{:02X}",
            self.0[0], self.0[1], self.0[2], self.0[3], self.0[4],
                self.0[5]
        )
    }
}

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
struct IpAddress([u8; 4]);

impl IpAddress {
    fn new(a: u8, b: u8, c: u8, d: u8) -> Self {
        Self([a, b, c, d])
    }

    fn is_multicast(&self) -> bool {
        self.0[0] >= 224 && self.0[0] <= 239
    }
}

impl fmt::Display for IpAddress {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{}.{}.{}.{}", self.0[0], self.0[1], self.0[2],
            self.0[3])
    }
}

#[derive(Clone)]
struct MulticastPacket {
    dst_ip: IpAddress,
    dst_mac: MacAddress,
}

impl MulticastPacket {
    fn new(dst_ip: IpAddress) -> Self {
        let dst_mac = MacAddress::from_multicast_ip(&dst_ip);
        Self { dst_ip, dst_mac }
    }
}

struct HardwareHashTable {
    bits: Vec<bool>,
    size_bits: u8,
    crc: Crc32,
}

impl HardwareHashTable {
    fn new(size_bits: u8) -> Self {
        Self {
            bits: vec![false; 1 << size_bits],
            size_bits,
            crc: Crc32::new(),
        }
    }

    fn add_mac(&mut self, mac: &MacAddress) {
        let idx = self.crc.hash_to_index(mac, self.size_bits);
        self.bits[idx] = true;
    }

    fn check_mac(&self, mac: &MacAddress) -> bool {
        let idx = self.crc.hash_to_index(mac, self.size_bits);
        self.bits[idx]
    }
}

struct SoftwareFilter {
    subscribed: HashSet<IpAddress>,
}

impl SoftwareFilter {
    fn new() -> Self {
        Self {
            subscribed: HashSet::new(),
        }
    }

    fn subscribe(&mut self, ip: IpAddress) {
        self.subscribed.insert(ip);
    }

    fn is_subscribed(&self, ip: &IpAddress) -> bool {
        self.subscribed.contains(ip)
    }
}

#[derive(Default)]
struct SimulationStats {
    total: usize,
    hw_dropped: usize,
    hw_passed: usize,
    sw_accepted: usize,
    sw_dropped: usize,
}

struct MulticastFilterSimulator {
    hw: HardwareHashTable,
    sw: SoftwareFilter,
    stats: SimulationStats,
    mac_to_ips: HashMap<MacAddress, Vec<IpAddress>>,
}

impl MulticastFilterSimulator {
    fn new(bits: u8) -> Self {
        Self {
            hw: HardwareHashTable::new(bits),
            sw: SoftwareFilter::new(),
            stats: SimulationStats::default(),
            mac_to_ips: HashMap::new(),
        }
    }

    fn subscribe(&mut self, ip: IpAddress) {
        assert!(ip.is_multicast());
        self.sw.subscribe(ip);
        let mac = MacAddress::from_multicast_ip(&ip);
        self.hw.add_mac(&mac);
        self.mac_to_ips.entry(mac).or_insert_with(Vec::new).push(ip)
            ;
    }

    fn process(&mut self, pkt: MulticastPacket) {
        self.stats.total += 1;

        if !self.hw.check_mac(&pkt.dst_mac) {
            self.stats.hw_dropped += 1;
            return;
        }

        self.stats.hw_passed += 1;

        if self.sw.is_subscribed(&pkt.dst_ip) {
            self.stats.sw_accepted += 1;
        } else {
            self.stats.sw_dropped += 1;
        }
    }
}

fn generate_well_known_addresses() -> Vec<IpAddress> {
    vec![
        IpAddress::new(224, 0, 0, 1),
        IpAddress::new(224, 0, 0, 2),
        IpAddress::new(224, 0, 0, 5),
        IpAddress::new(224, 0, 0, 6),
        IpAddress::new(224, 0, 0, 9),
    ]
}

fn main() {
    let mut sim = MulticastFilterSimulator::new(4);

    let subs = vec![
        IpAddress::new(224, 0, 0, 1),
        IpAddress::new(224, 0, 0, 5),
        IpAddress::new(224, 0, 0, 251),
        IpAddress::new(239, 192, 1, 1),
        IpAddress::new(239, 192, 2, 2),
    ];

    println!("multicast filter simulation");
    println!();
    println!("hardware hash table size: {} bits ({} entries)", 4, 1
        << 4);
    println!();

    println!("subscribed multicast groups:");
    for ip in &subs {
        let mac = MacAddress::from_multicast_ip(ip);
        println!("  {} -> {}", ip, mac);
        sim.subscribe(*ip);
    }
    println!();

    let mut packets = Vec::new();

    for ip in &subs {
        for _ in 0..20 {
            packets.push(MulticastPacket::new(*ip));
        }
    }

    let others = vec![
        IpAddress::new(224, 0, 0, 2),
        IpAddress::new(224, 0, 0, 9),
        IpAddress::new(224, 0, 1, 1),
        IpAddress::new(239, 100, 1, 1),
        IpAddress::new(239, 100, 2, 2),
        IpAddress::new(238, 50, 50, 50),
        IpAddress::new(224, 1, 1, 1),
        IpAddress::new(224, 2, 2, 2),
        IpAddress::new(225, 1, 1, 1),
        IpAddress::new(230, 5, 5, 5),
    ];

    for ip in &others {
        for _ in 0..30 {
            packets.push(MulticastPacket::new(*ip));
        }
    }

    println!("processing {} packets...", packets.len());
    println!();

    for pkt in packets {
        sim.process(pkt);
    }

    println!("simulation results:");
    println!("  total packets: {}", sim.stats.total);
    println!("  hardware dropped: {}", sim.stats.hw_dropped);
    println!("  hardware passed: {}", sim.stats.hw_passed);
    println!("  software accepted: {}", sim.stats.sw_accepted);
    println!("  software dropped: {}", sim.stats.sw_dropped);
    println!();

    let hw_filter_rate = (sim.stats.hw_dropped as f64 / sim.stats.
       total as f64) * 100.0;
    let false_positive_rate = (sim.stats.sw_dropped as f64 / sim.
       stats.hw_passed as f64) * 100.0;

    println!("performance metrics:");
    println!("  hardware filtering ratio: {:.2}%", hw_filter_rate);
    println!("  false positive rate: {:.2}%", false_positive_rate);
    println!();

    println!("hash collision analysis:");
    for (mac, ips) in &sim.mac_to_ips {
        if ips.len() > 1 {
            println!("  collision at mac {}", mac);
            for ip in ips {
                println!("    {}", ip);
            }
        }
    }

    if sim.stats.sw_dropped > 0 {
        println!();
        println!("false positive examples (unsubscribed traffic leaked through hw filter):");
        println!("  {} packets from unsubscribed groups passed hardware filter", sim.stats.sw_dropped);
        println!("  these share hash indices with subscribed groups");
    }

    let crc = Crc32::new();
    println!();
    println!("well-known multicast addresses:");
    for ip in generate_well_known_addresses().iter() {
        let mac = MacAddress::from_multicast_ip(ip);
        let hash_index = crc.hash_to_index(&mac, 4);
        println!("  {} -> {} (hash index: {})", ip, mac, hash_index);
    }
}
```