POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

NETWORK COMPUTING
Computer Science and Engineering
Ingegneria Informatica

Final Project 2024-25, Comprehensive Report

Author:
Shreesh Kumar Jha

Student ID:
11022306

Advisor:
Prof. Gianni Antichi, Sebastiano Milano

Academic Year:
2024-25

Contents

1 — TCP Connection Tracker Analysis and Performance Report

Executive Summary

This project transformed a basic eBPF/XDP TCP connection tracker into a fully operational, high-performance system. Key outcomes include:

- Achieved **3.57 Gbps** sustained TCP throughput (up from 0.28 Gbps baseline) with **0% packet loss** and **0 retransmits**.
- Enabled **817 Mbps** UDP flows (4% receiver-side loss at 1 Gbps offered).
- Added automatic per-flow TTL for garbage collection, preventing map exhaustion.
- Implemented an RFC 793–compliant TCP state machine with correct RST/FIN handling.

1.1. Problem Statement & Initial Assessment

1.1.1. Baseline Failures

Before any fixes, the tracker exhibited:

- **Broken TCP Handshake:** SYN seen, but SYN-ACK/ACK never progressed → no established connections.
- **No RST/FIN Cleanup:** RSTs were ignored; FIN transitions violated RFC 793 → "zombie" entries persisted.
- **Limited TCP Throughput:**

- `iperf3 baseline`: ∼339 MB in first second (∼2.84 Gbps), then 0 B for remaining 9 s → ∼283 Mbps overall.
- 5 retransmits; handshake dropped after 1 s.

- **No UDP Support:** All UDP packets were treated as invalid → dropped.

1.1.2. Root Causes

1. **Faulty TCP-Flag Logic:** Incorrect Boolean expressions prevented detection of SYN+ACK and ACK.
2. **Absent RST/FIN Handling:** RST never cleaned up; FIN state transitions were incorrect.
3. **No TTL/Garbage Collection:** Connections never expired → map exhaustion under churn.
4. **UDP Flows Not Tracked:** UDP packets were classed as invalid → no UDP traffic passed.

1.2. Key Fixes & Engineering Solutions

1.2.1. TCP Flag Validation & Handshake Logic

Issue Original code used expressions like:

```
if ((pkt.flags & TCPHDR_SYN) != 0 &&
    (pkt.flags | TCPHDR_SYN) == TCPHDR_SYN)
```

which never matched `SYN+ACK` or `ACK`-only, thus breaking the handshake.

Solution Simplify to exact or masked comparisons:

```
// New connection (SYN only)
if (pkt.flags == TCPHDR_SYN) {
    // Insert new 'SYN_SENT' entry
}

// SYN+ACK detection
if ((pkt.flags & (TCPHDR_SYN|TCPHDR_ACK)) == (TCPHDR_SYN|
   TCPHDR_ACK)) {
    // Transition 'SYN_SENT' -> 'SYN_RECV'
```

```
}

// ACK-only detection
if (pkt.flags == TCPHDR_ACK) {
    // Transition 'SYN_RECV' -> 'ESTABLISHED'
}
```

Listing 1.1: Corrected TCP flag detection

Impact Full three-way handshake: `SYN_SENT` → `SYN_RECV` → `ESTABLISHED`.

1.2.2. RST & FIN Handling

RST Handling

Issue RST packets were forwarded without deleting the connection entry, leaving stale states in the map.

Solution

```
if (pkt.flags & TCPHDR_RST) {
    struct ct_v *v = bpf_map_lookup_elem(&connections, &key);
    if (v) {
        bpf_map_delete_elem(&connections, &key);
    }
    // Allow RST to pass
    pkt.connStatus = ESTABLISHED;
    goto PASS;
}
```

Listing 1.2: RST packet handling

FIN Handling

Issue Original code lacked any FIN sequence handling, violating correct TCP teardown.

Solution Implement proper FIN state transitions:

```
// After finding existing entry 'v':
if (saved_state == ESTABLISHED && (pkt.flags & TCPHDR_FIN)) {
    v->state = FIN_WAIT_1;
    v->sequence = pkt.seqN + 1;
```

```
    v->ttl = now + TCP_FIN_WAIT_TIMEOUT;
    bpf_spin_unlock(&v->lock);
    pkt.connStatus = ESTABLISHED;
    goto PASS;
}

if (saved_state == FIN_WAIT_1) {
    if (pkt.flags == TCPHDR_ACK) {
        v->state = FIN_WAIT_2;
        v->ttl = now + TCP_FIN_WAIT_TIMEOUT;
    } else if (pkt.flags & TCPHDR_FIN) {
        v->state = LAST_ACK;
        v->sequence = pkt.seqN + 1;
        v->ttl = now + TCP_LAST_ACK_TIMEOUT;
    }
    bpf_spin_unlock(&v->lock);
    pkt.connStatus = ESTABLISHED;
    goto PASS;
}

if (saved_state == LAST_ACK && pkt.flags == TCPHDR_ACK &&
    pkt.seqN == saved_seq) {
    bpf_spin_unlock(&v->lock);
    bpf_map_delete_elem(&connections, &key);
    pkt.connStatus = ESTABLISHED;
    goto PASS;
}
```

Listing 1.3: FIN packet state transitions

Impact

- RST immediately removes the map entry.
- FIN sequences now progress through `FIN_WAIT_1` → `FIN_WAIT_2`/`LAST_ACK` → `TIME_WAIT` → deletion.
- No "zombie" entries remain.

1.2.3. Garbage Collection via TTL

Issue Connections never expired, causing unbounded map growth under churn.

Solution Assign a TTL (`uint64_t`) to each entry based on its state and the current timestamp (from `bpf_ktime_get_ns()`). On lookup, delete any entry whose TTL has passed:

```
// On lookup:
if (v && v->ttl < now) {
    bpf_map_delete_elem(&connections, &key);
    v = NULL;  // Treat as new connection
}

// Creating a new TCP entry on SYN:
struct ct_v newEntry = {};
newEntry.state = SYN_SENT;
newEntry.ttl   = now + TCP_SYN_SENT_TIMEOUT;  // e.g. 2 minutes
newEntry.sequence = pkt.seqN + 1;
newEntry.ipRev    = ipRev;
newEntry.portRev  = portRev;
bpf_map_update_elem(&connections, &key, &newEntry, BPF_ANY);
pkt.connStatus = NEW;
goto PASS;
```

Listing 1.4: TTL-based garbage collection

Timeout Values

- `TCP_SYN_SENT_TIMEOUT`: 2 minutes
- `TCP_SYN_RECV_TIMEOUT`: 1 minute
- `TCP_ESTABLISHED_TIMEOUT`: 5 days
- `TCP_FIN_WAIT_TIMEOUT`: 2 minutes
- `TCP_LAST_ACK_TIMEOUT`: 2 minutes
- `TCP_TIME_WAIT_TIMEOUT`: 2 minutes

Impact Expired entries are removed lazily when accessed, keeping the BPF map bounded under churn.

1.2.4. UDP Flow Tracking

Issue UDP packets were treated as invalid and therefore dropped.

Solution Use the same 5-tuple key for UDP, but with a simpler state model. On the first packet, insert a `NEW` entry with a TTL; on subsequent packets, refresh or promote to `ESTABLISHED`:

```
if (pkt.l4proto == IPPROTO_UDP) {
    struct ct_v *v = bpf_map_lookup_elem(&connections, &key);
    if (v && v->ttl < now) {
        bpf_map_delete_elem(&connections, &key);
        v = NULL;
    }

    if (v) {
        bpf_spin_lock(&v->lock);
        bool same_dir = (v->ipRev == ipRev && v->portRev == portRev);
        if (same_dir) {
            v->ttl = now + UDP_FLOW_TIMEOUT;  // Extend unidirectional
            pkt.connStatus = ESTABLISHED;
        } else {
            v->state = ESTABLISHED;           // Promote to bidirectional
            v->ttl   = now + UDP_ESTAB_TIMEOUT;
            pkt.connStatus = ESTABLISHED;
        }
        bpf_spin_unlock(&v->lock);
        goto PASS;
    }

    // New UDP flow
    struct ct_v newEntry = {};
    newEntry.state    = NEW;
    newEntry.ttl      = now + UDP_FLOW_TIMEOUT;  // 5 minutes
    newEntry.ipRev    = ipRev;
    newEntry.portRev = portRev;
    newEntry.sequence = 0;
```

```
    bpf_map_update_elem(&connections, &key, &newEntry, BPF_ANY);
    pkt.connStatus = NEW;
    goto PASS;
}
```

Listing 1.5: UDP flow tracking implementation

Timeout Values

- `UDP_FLOW_TIMEOUT`: 5 minutes (unidirectional)
- `UDP_ESTAB_TIMEOUT`: 10 minutes (bidirectional)

Impact

- Unidirectional UDP flows pass; return traffic promotes to `ESTABLISHED`.
- `iperf3` UDP @1 Gbps → ~817 Mbps sender, ~782 Mbps receiver, ~4% loss (expected at line-rate).

1.3. Experimental Results

1.3.1. Workflow

1. **Terminal 1:**

```
sudo ./conntrack -1 veth1 -2 veth2 -l 5 &
```

Listing 1.6: Launch conntrack

2. **Terminal 2 (Trace):**

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

Listing 1.7: Kernel trace

   - Verified packet parsing, state transitions, and `bpf_redirect()`.

3. **Terminal 3 (`iperf3`):**

   - **TCP Server:**

```
sudo ip netns exec ns2 iperf3 -s &
```

- **TCP Client:**

```
sudo ip netns exec ns1 iperf3 -c 10.0.0.2 -t 10
```

- **UDP Client:**

```
sudo ip netns exec ns1 iperf3 -c 10.0.0.2 -u -b 1G -t 10
```

1.3.2. TCP Throughput

Before Fixes (Baseline)

Server [ns2]:

```
[  5]  0.00-1.00   sec  339 MBytes  2.84 Gbits/sec
[  5]  1.00-10.00  sec  0.00 Bytes  0.00 bits/sec
[  5]  0.00-10.04  sec  339 MBytes  283 Mbits/sec  (receiver)
```

Client [ns1]:

```
[  5]  0.00-1.00   sec  341 MBytes  2.86 Gbits/sec   1 retr   1.41 KB cwnd
[  5]  1.00-10.00  sec  0.00 Bytes  0.00 bits/sec    4 retr   1.41 KB cwnd
[  5]  0.00-10.00  sec  341 MBytes  286 Mbits/sec    5 retransmits  (sender)
```

Summary

- **Average throughput:** ∼283 Mbps (server), ∼286 Mbps (client).
- Handshake dropped after the first second → no steady flow.

After Fixes

Server [ns2]:

```
[  5]  0.00-1.00   sec  456 MBytes  3.82 Gbits/sec
[  5]  1.00-2.00   sec  416 MBytes  3.49 Gbits/sec
...
[  5]  9.00-10.00  sec  432 MBytes  3.62 Gbits/sec
[  5]  0.00-10.04  sec  4.18 GBytes  3.57 Gbits/sec  (receiver)
```

Client [ns1]:

```
[  5]  0.00-1.00   sec  474 MBytes  3.96 Gbits/sec   0 retr   274 KB cwnd
[  5]  1.00-2.00   sec  418 MBytes  3.50 Gbits/sec   0 retr   321 KB cwnd
...
[  5]  9.00-10.00  sec  433 MBytes  3.63 Gbits/sec   0 retr   284 KB cwnd
[  5]  0.00-10.00  sec  4.18 GBytes  3.59 Gbits/sec   0 retransmits  (sender)
[  5]  0.00-10.04  sec  4.18 GBytes  3.57 Gbits/sec  (receiver)
```

Summary

- **Sustained throughput:** 3.57 Gbps (server), 3.59 Gbps (client).
- **Retransmissions:** 0.
- **Loss:** 0%.

Insight Achieved high-performance IPv4 forwarding with proper connection tracking.

1.3.3. UDP Throughput

Client [ns1]:

```
[  5]  0.00-1.00   sec  92.8 MBytes  779 Mbits/sec  67,223 datagrams
[  5]  1.00-2.00   sec  93.9 MBytes  788 Mbits/sec  68,033 datagrams
...
[  5]  9.00-10.00  sec  95.5 MBytes  801 Mbits/sec  69,171 datagrams
[  5]  0.00-10.00  sec  974 MBytes  817 Mbits/sec  0 lost  (sender)
```

Server [ns2]:

```
[  5]  0.00-10.04  sec  935 MBytes  782 Mbits/sec  0.023 ms jitter  27,873/705,285
```

Summary

- **Offered rate:** 1 Gbps.
- **Sustained:** 817 Mbps (client), 782 Mbps (server).
- **Receiver loss:** 4% (expected near line-rate).
- **Jitter:** 0.023 ms.

Insight UDP flows tracked correctly; TTL-based expiration prevents map growth.

1.4. Engineering Trade-Offs

Table 1.1: Chosen approaches and rationale

Aspect	Chosen Approach	Rationale
TCP Flag Checks	Exact bitmask comparisons	Ensures RFC 793 compliance; eliminates logical bugs.
RST/FIN Cleanup	Immediate deletion on RST or final ACK	Prevents stale entries; enforces correct teardown.
Sequence Validation	Relaxed for SYN+ACK (flag-only)	Compatible with Linux TCP ISN behavior; avoids false drops.
Timeout Strategy	Per-state TTL values	Balances memory cleanup vs. long-lived flows.
Map Lookups	Single lookup per packet + spin-lock	Halved map operations; reduced lock contention.
UDP Flow Tracking	Two-state model (NEW → ESTABLISHED)	Simple flow tracking; TTL handles idle flows.
Debug Logging	Verbose at `-l 5`, otherwise silent	Aids development without runtime overhead.

1.5. Conclusion

1. **Functionality Restored:**
    - **TCP:** RFC 793–compliant handshake and teardown, with RST/FIN cleanup.
    - **UDP:** Flows tracked; bidirectional detection via TTL.
2. **Performance Achieved:**
    - **TCP:** Sustained **3.57 Gbps** (vs. ∼0.28 Gbps baseline) with **0% loss**.
    - **UDP: 817 Mbps** (sender) / **782 Mbps** (receiver) at 1 Gbps offered, **4% loss**.
3. **Resource Management:**
    - **Automatic TTL:** State entries expire; BPF map remains stable.
    - **Optimized Lookups:** Single lookup per packet reduces BPF overhead by ∼50%.

4. **Relevance:**

   - Ready for any XDP-capable Linux 5.x+ environment.
   - High-performance IPv4 connection tracking across network namespaces.
   - Easily extended to additional protocols or namespaces.