A programmable Qdisc with eBPF

Cong Wang
xiyou.wangcong@gmail.com
Agenda

- What is a packet scheduler
- A quick summary of existing Qdisc’s
- Programmable Qdisc’s
- eBPF quick overview
- Prototypes of sch_bpf
Packet scheduling

- Ordering packets
- Traffic shaping/policing
- Flow classification and isolation
- Fairness
- Latency control
Qdisc overview

- FIFO: pfifo, bfifo, pfifo_fast
- Fair queueing: fq, sfq, qfq, drr
- Traffic shaper: tbf, htb, hfsc
- AQM: choke, codel, pie
- Multiqueue: mq, mqprio
- And more...
Why programmable?

- One qdisc for each algorithm
- Choosing qdisc’s is not easy
- Much more flexible
- Safer, easier to port and update via eBPF (CO-RE)
Push-In-First-Out

- Enqueue in arbitrary position, dequeue from head
- Rank based priority queue, $O(\log N)$
- Relative ordering with scheduling trees
- No arbitrary reordering
Eiffel

- FFS-based priority queue
- On-dequeue scheduling
- Per-flow ranking and scheduling
- Arbitrary traffic shaping on any node
eBPF overview

- Map based data structures
- Array map
- Queue/stack maps
- Map in map
- One or multiple programs for each attach point
- cls_bpf, act_bpf are already available
Why sch_bpf is harder?

- At least two eBPF programs: enqueue() and dequeue()
- Work together via a shared data structure
- dequeue() is essentially harder than enqueue()
- Who owns the data structure? Kernel or user?
- How flexible is it?
- Interaction with TC filters and actions
- Hierarchy
Design considerations

- Flexibility vs. Usability
- Kernel vs. User
- Efficiency
- Fits into existing TC infrastructure
A lazy prototype

enqueue

BPF program

BPF data structure

dequeue

BPF program

BPF program
struct bpf_map_def SEC("maps") queue = {...};
SEC("sch_bpf/enqueue")
int enqueue(struct __sk_buff *skb)
{
    if (bpf_map_push_elem(&queue, skb, 0))
        return DROP;
    return SUCCESS;
}
SEC("sch_bpf/dequeue")
int dequeue(struct sch_ctx *ctx)
{
    void *skb;
    if (bpf_map_pop_elem(&queue, &skb))
        return NONE;
    ctx->skb = skb;
    return SUCCESS;
}
Problems

- Too flexible? Packets could be held infinitely.
- Hard to fit in Qdisc APIs: ->init(), ->peek(), ->reset()
- Multiple flows, map-in-map and map creation
- skb ownership
Second prototype

• Based on PIFO, a priority queue owned by kernel
• Invisible from eBPF program
• Enqueue(): calculate the rank, decide whether to drop
• Dequeue(): nothing
BPF program \(\xrightarrow{\text{enqueue}}\) Priority Queue \(\xrightarrow{\text{dequeue}}\)
int counter;
SEC("sch_bpf_enqueue")
int enqueue(struct sch_ctx *ctx)
{
    if (ctx->total_packets > 1000)
        return DROP;
    ctx->skb->tc_rank = counter;
    counter++;
    return SUCCESS;
}
Third prototype

- Multiple priority queues owned by kernel
- Map each queue to a flow/class
- Invisible from eBPF programs
- Enqueue(): classify packet to a queue, calculate rank within the queue
- Dequeue(): decide how many packets to dequeue from which queues
TC cmdline

- tc qdisc add dev X root handle 1: bpf flows N enqueue obj bpf.o sec enqueue dequeue obj bpf.o sec dequeue
- tc filter add dev X parent 1:0 [...] flowid 1:Y
- tc class add dev X parent 1:0 classid 1:1 bpf rate 10Mbit
int counter;
SEC("sch_bpf/enqueue")
int enqueue(struct sch_ctx *ctx)
{
    int classid;
    if (ctx->total_packets > 1000)
        return DROP;
    classid = bpf_tc_classify(ctx); // Wrapper for tcf_classify()
    ctx->skb->tc_classid = classid;
    ctx->skb->tc_rank = classid + counter;
    counter++;
    return SUCCESS;
}
int current;
int quota = QUANTUM;
SEC("sch_bpf/dequeue")
int dequeue(struct sch_ctx *ctx)
{
    quota = quota - ctx->skb->len;
    if (quota <= 0) {
        quota = quota + QUANTUM;
        return SCH_BPF_REQUEUE | SCH_BPF_DONE;
    } else
        return SCH_BPF_OK | SCH_BPF_CONTINUE;
}
while (1) {
    ctx.flow = prio_dequeue(&q->flows);
    if (!ctx.flow)
        break;
    ctx.skib = prio_dequeue(&ctx.flow->queue);
    ret = BPF_PROG_RUN(&prog->filter, &ctx);
    if (ret & mask == SCH_BPF_DROP)
        kfree_skb(ctx.skib);
    else if (ret & mask == SCH_BPF_REQUEUE)
        prio_enqueue(&ctx.flow->queue, ctx.skib);
    else
        dev_hard_start_xmit(ctx.skib, dev, ...);
    if (ret & mask == SCH_BPF_DONE)
        break;
    if (!prio_empty(&ctx.flow->queue))
        prio_enqueue(&q->flows, ctx.flow);
}
Problems

- `sch_ctx` becomes complicated
- dequeue() is stateful, harder to implement. Use multi-prog?
- $O(M\log N)$
- Need to provide per-flow information?
- Need arbitrary flow/packet access?
- Would eBPF verifier be happy?
Ideas? Questions?
References

- https://blogs.oracle.com/linux/notes-on-bpf-3