BPF Host Network Resource Management

Lawrence Brakmo, Alexei Starovoitov

Facebook
Menlo Park, USA

Abstract

Linux already supports the allocation and management of many of its resources. Examples are CPU and memory, were one can allocate these resources per cgroup. However, the network subsystem lacks good mechanisms for managing its resources. For example, it is not easy (or in some cases possible) to allocate egress and ingress bandwidth per cgroup.

In contrast to CPU and memory that are local resources, networking is a global shared resource. For example, if we want to limit ingress bandwidth per cgroup, we need to modify the sender to slow down. Dropping packets at the receiver when the ingress bandwidth is exceeded may penalize the sender but cannot recover the wasted bandwidth.

In this paper we propose a new BPF based mechanism for managing bandwidth that is efficient, eliminates standing queues and is flexible.

Keywords

BPF, Linux, TCP, Networking

Introduction

The goal of the BPF Nework Resource Management (NRM) project is to provide the necessary mechanisms for managing network resources, such as bandwidth, through BPF programs. In other words, to create a BPF based framework for efficiently supporting policing of both egress and ingress traffic based on both local and global network allocations. For example, limiting per-cgroup egress and ingress bandwidths.

By efficient we mean things like not wasting bandwidth (i.e. flows can reach the imposed limits) and not increasing latencies (i.e. not increasing RTTs or not increasing tail latencies of RPCs). Just dropping packets can cause both issues, so the framework supports other mechanisms that are covered in later sections.

Linux currently provides traffic control (tc) and queue disciplines (qdisc) that can be used for limiting egress rates. However, there is a history of performance issues when using the HTB qdisc for this purpose. In addition, using qdisc for rate limiting can create standing queues in the queue discipline that increase connection RTTs. Finally, using tc qdiscs lacks the flexibility inherent in BPF programs.

In addition to providing mechanisms for local network resource management, we are also adding support for managing global resources, such as bandwidths external to the host. Examples of this would be managing bandwidths at external links, such as backbone links, or ingress bandwidths at other hosts (which require notifying the sender to slow down).

Overview

The NRM framework uses existing BPF cgroup skb hooks (egress and ingress) to keep track of bandwidth use and to enforce bandwidth limits. Rather than drop packets as most tc qdisc do to enforce limits, we provide a richer set of tools to achieve this. For example, the NRM BPF program can use ECN congestion marking for flows supporting ECN. In addition, for TCP flows, it can call tcp_enter_cwr() to reduce the cwnd of the TCP flow or it can also set the cwnd directly. And finally, it can also just drop the packet which is necessary to insure enforcement of bandwidth limits.

In order to support more intelligent BPF NRM program, these programs can also read current and minimum RTTs as well as the value of the current cwnd. The ability to know the flow's RTT means that one can write NRM BPF programs that could increase fairness between small RTT and long RTT flows when enforcing egress or ingress bandwidth limits.

In order to achieve support for enforcing bandwidth limits based on global network allocations, we propose the use of resource scopes. For example, we could have a per cgroup scope that is used to police egress and/or ingress traffic, a scope for a particular backbone link (i.e. policing all traffic from this host on that link), etc.

For the rest of the paper we will only consider per cgroup egress and ingress scopes. Support for more complex scopes is currently under development.

Bandwidth Management

We use virtual token bucket queues to manage bandwidth; each scope has its own virtual queue managed by the NRM BPF program. The minimum state required to maintain the virtual queue is:

```
struct vqueue {
     long credit;          // in bytes
     long long last_time;  // in ns
};
```

The state is updated whenever a packet is sent as follows:

```
vqueue.credit += credit_per_ns(cur-
rent_time - last_time, limit_rate);

vqueue.credit = max(vqueue.credit,
MAX_CREDIT);

vqueue.credit -=
wire_length_in_bytes(skb);

vqueue.last_time = current_time;
```

Where:

`credit_per_ns()` returns new credit accrued during the interval of time since the last packet was sent.

`MAX_CREDIT` represents the maximum credit that can be accrued. Credit that is not used cannot continue to accumulate forever.

`wire_length_in_bytes(skb)` Returns the size in bytes that the skb will take in the link. Because the skb can be larger than the maximum packet size (due to TSO), we need to account for the extra bytes taken by each packet header (E.g. TCP, IP and ETH).

Then, the NRM BPF program makes decisions based on the credit, skb and socket information when the NRM BPF program executes, triggered by an skb write.

Our sample program uses 2 negative valued thresholds, a `MARK_THRESHOLD` and a `DROP_THRESHOLD` for determining what actions to take. Figure 1 shows how the current value of credit triggers actions. Note that the credit is allowed to be negative in order to support bursts without dropping packets. If the credit is greater than `MARK_THRESHOLD`, then the skb goes through. If the credit is between the thresholds, then the packet is "marked".

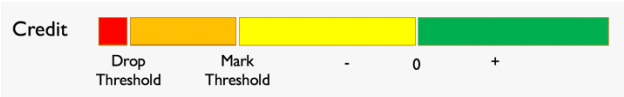

Figure 1: Credit thresholds

The action taken on marked packets by the sample program is dependent on flow details. If the flow is ECN enabled (i.e. its IP packets are marked with either ECT0 or ECT1), then the packet is ECN marked as having experienced congestion. Otherwise, if the flow is a TCP flow, we call the BPF helper function bpf_enter_cwr(skb), which calls tcp_enter_cwr() to reduce the congestion window (cwnd) of the flow, based on a linear probability function. Figure 2 shows the response function used to determine if the flow will be made to reduce its cwnd.

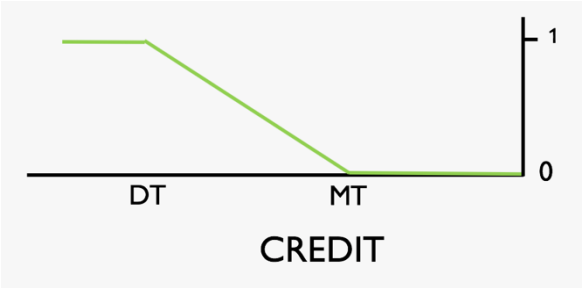

Figure 2: Probability of calling bpf_enter_cwr()

The closer the credit is to `DROP_THRESHOLD`, the more likely that its cwnd will be decreased. If the credit is less than `DROP_THRESHOLD`, then the packet is dropped. In practice, there is a small area (equal to 15*1500 bytes) that is reserved for small packets (less than 100 bytes) in order to reduce the likelihood that small packets will be dropped. This protects SYN and SYN-ACK TCP packets as well as pure ACK packets.

Performance Issues

While developing the NRM framework we became aware of issue that were affecting performance.

1. TCP not aware when packets dropped

TCP is not aware when packets dropped by the NRM BPF program. When packets are dropped by a qdisc, or a BPF program associated with traffic control (tc), queue_transmit (called from __tcp_transmit_skb()) returns a special value to nofity TCP that the packet was dropped. TCP then "forgets" that it sent the packet (so it will be transmitted again) and tcp_enter_cwr() is called to reduce the cwnd.

In contrast, when packets are dropped by a cgroup skb BPF program, there is not special return value to notify TCP. As a result, TCP assumes the packet was sent and it will need to go through the expensive (in terms of flow performance) loss detection and recovery code.

Our solution was to use a flag to notify TCP that the packet was dropped by the NRM BPF program so TCP will "forget" it sent the packet.

2. High tail latencies due to dropped packets

When a packet is dropped, and there are no packets in flight, it is possible that nothing will trigger sending a new packet

until a timeout occurs (no ACKs will arrive to trigger sending new packets). The timeout that usually triggers new packets to be sent is the probe timer (around 200ms).

Our solution was to decrease the probe timer to 20ms when a packet is dropped and there are no other packets in flight. We created a new sysctl to control the value to use for the probe timer. It defaults to 20ms and setting it to 0 disables decreasing the probe timer when the NRM BPF program drops packets.

Table 1 shows the benefits of our solution. We use a rate limit of 1Gbps for a cgroup with between 1 and 9 concurrent flows running back-to-back RPCs. The first 2 columns show the aggregate goodput (payload throughput) in Mbps with the upstream kernel and with a kernel patches with our solution (small timer). The last 2 columns show the 99.9% latencies for the 1MB RPCs before and after implementing our solution.

# Flows	Cubic Aggr BW		Cubic 99.9% Lat	
		small timer		small timer
1	496M	794M	250ms	84ms
2	856M	922M	260ms	44ms
5	935M	989M	465ms	92ms
9	999M	1006M	600ms	345ms

Table 1: Effect of decreasing probe timer on Cubic traffic

The table shows improvements in both the aggregate goodput and in the 99.9 percentile latencies. The goodput increases are especially large when there is only one flow.

Table 2 shows the respective results when we use DCTCP (which uses ECN marking) instead of Cubic. The improvements are not as large except for the 9 flows case. Finally, note that the aggregate goodput for both Cubic and DCTCP is larger than 1Gbps in some cases. This was caused by the following issue.

# Flows	DC-TCP Aggr BW		DC-TCP 99.9% Lat	
		small timer		small timer
1	953M	953M	9ms	9ms
2	962M	962M	22ms	21ms
5	1003M	1004M	52ms	48ms
9	1029M	1020M	308ms	78ms

Table 2: Effect of decreasing probe timer on DCTCP traffic

3. Updating credit and last_time is a critical section

In multiprocessors, the NRM BPF code that updates both the credit and last_time forms a critical section that needs to be protected. Otherwise, the NRM BPF program cannot enforce the bandwidth limits. Our solution was to protect the whole NRM BPF program with spinlocks, making the whole program a protected critical section.

Obviously, this is not an ideal solution since it increases CPU usage due to contention of the spinlocks. There are two other solutions we are working on:

1. Add support for spinlocks in BPF programs. This is currently being worked on.
2. Explore using a data structure for managing the virtual queue that does not have a critical section.

Once we protected the critical section the bandwidth used by the cgroup was always within the desired limit. In addition, the tail latencies or the RPC flows also decreased.

Experimental Results

The experimental setup was as follows:

1. We only used 1 cgroup
2. One server sends to another in the same rack
3. We had 1 to 9 concurrent flows doing back to back RPCs:
    a. 1 flow: 1 – 1MB RPC
    b. 2 flows: 1 – 1MB RPC and one 10KB RPC
    c. 5 flows: 4 – 1MB RPCs and one 10KB RPC
    d. 9 flows: 8 – 1MB RPCs and one 10KB RPC
4. Bandwidth is limited either by using TC with HTB or using our NRM framework
5. Limits of 1Gbps or 5 Gbps
6. In some cases we used netem to increase link latency to 10ms.
7. We compare 4 cases:
    a. Cubic[3] using TC and HTB for rate shaping
    b. Cubic using NRM BPF for policing
    c. Cubic with ECN and NRM BPF for policing
    d. DCTCP[2] and NRM BPF for policing

We used Netesto[1] to create the traffic and collect flow statistics and experiment metrics.

Egress, 1Gbps no added delay

The first experiment consists of a 1Gbps rate limit. Figure 3 shows the aggregate rate for all concurrent flows as well as the RTT seen by the flows (or by a different process within the cgroup doing pings).

The axis on the left goes with the bars, while the axis on the right goes with the diamonds. The graph shows that when using TCP the aggregate bandwidths are a little smaller, except for the case of 1 Cubic flow without ECN

where the aggregate bandwidth is about 10% lower. In addition, the RTTs are much larger when using TC and HTB for rate shaping.

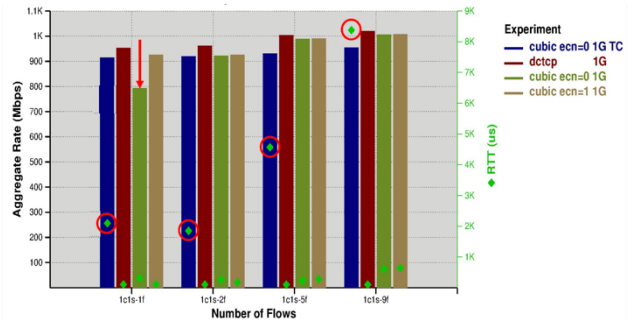

Figure 3: Aggregate Rate and RTT for 1Gbps limit

The larger RTTs are caused by the standing queues (queues that do not completely dissipate) caused by using HTB for rate limiting. Note that we used the default value of 260KB for the sysctl tcp_limit_output_bytes. At 1Gbps, sending 260KB takes about 2ms, which matches the RTT results.

The next figure, Figure 4, shows the aggregate rates for the 1MB RPCs (bars) and the rate of the 10KB RPC (diamond). Note how the rate of the 10KB RPCs are very small when using HTB for rate control. This is caused by the standing queues and the resulting increase in RTTs. At most, only one RPC can occur per RTT. Hence, at 2ms RTT the 10KB are limited to a rate of at most 10KB/2ms or 40Mbps. As a consequence, using HTB creates unfairness between RPC of different sizes. Larger RPC sizes can achieve a higher rate (1MB RPCs are limited to a rate of 1MB/2ms or 4Gbps.

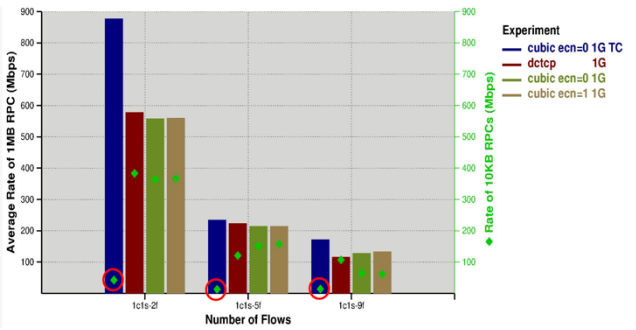

Figure 4: Goodputs of 1MB and 10KB RPCs

In contrast, when using NRM for rate limiting, the 10KB RPC achieves more than 60% of the rates of the 1MB RPCs. The next figure, Figure 5, shows the 99.9 and 50 percentile latencies for the 1MB RPCs. The primary issue is the increased tail latency for Cubic when using NRM. We will be exploring new response functions to see if we can decrease the tail latency.

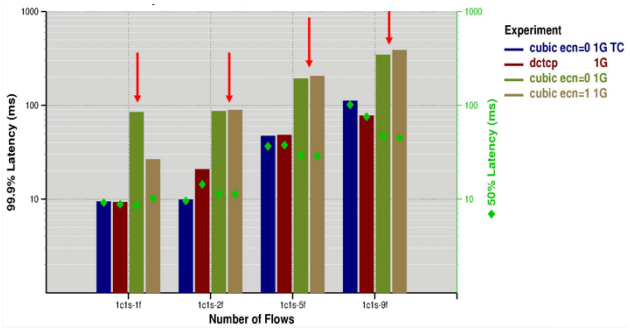

Figure 5: 1MB RPC Latencies

Figure 6 shows the 10KB 99.9 and 50 percentile latencies. DCTCP with NRM has much smaller 99.9 percentile latency as compared to the other cases.

In summary, the aggregate rate is similar among all the cases. Using HTB for rate limiting results in standing queues that increase RTTs. As a result, HTB in unfair between small and large RPCs; 10KB RPCs achieve rates up to 20x less. DCTCP with NRM hwas much lower 10KB tail latency, between 10 and 80x lower.

Egress, 1 Gbps, 10ms netem delay

Table 3 shows the aggregate rates and 1MB 99.9 percentile latencies when we increase the link delay to 10ms through netem. With only 1 flow, they all undershoot the bandwidth limit. HTB with fq qdisc (and pacing) achieves lower rates than without fq. With only one flow, the 99.9% latencies of HTB are worse than for DCTCP and NRM. On the other hand, the tail latencies of the 9 flow experiments are lower for HTB than for the others.

Cong Control	qdisc	Rate Control	1-flow Rate (Mbps)	9-flow Aggr Rate (Mbps)	1-flow 1-MB 99.9% Lat (ms)	9-flow 1-MB 99.9% Lat (ms)
Cubic	HTB – fq	HTB	441	858	58	85
Cubic	HTB	HTB	437	945	20	120
Cubic	mq – fq	NRM-BPF	410	915	46	141
Cubic	mq – fqc	NRM-BPF	754	944	12	218
DC-TCP	mq – fq	NRM-BPF	666	947	13	143

Table 3: Rates and latencies for 1Gbps and 10ms

Egress, 5 Gbps, no added delay

Table 4 shows the rates and latencies the cgroup rates are limited to 5Gbps. The most noticeable result is that, again, HTB penalizes the 10KB RPC. The rate is only 35 Mbps vs. 295 Mbps for DCTCP and NRM. Similarly, the 99.9% latency is 3.7ms vs. 0.8ms for DCTCP.

However, the 99.9% latencies are smaller with HTB than anything else. Especially when using Cubic, when the tail latencies are more than 5x larger. The tail latencies for DCTCP are “only” 50% larger with 9 flows. Looking with more detail at the DCTCP behavior we see some concerning behaviors, where the cwnd periodically decreases. We will

investigate to see if the performance of DCTCP can be improved.

Ingress, 1 and 5 Gbps rate limits

To limit ingress rates we use the same mechanisms as for egress. Since we need a mechanism to notify the sender to slow down (otherwise we cannot enforce the rate limit) we only tested DCTCP and NRM.

Table 4 shows the results. NRM with DCTCP was very effective at limiting the ingress rates achieving 925 Mbps and 4.6 Gbps respectively. The tail latencies were very well behaved, except for the case of 9 concurrent flows at 1Gbps rate limit when the NRM program dropped packets. But when no packets were dropped, the 99.9 and 50% latencies are very close (which is very good).

Limit		Aggregate		99.9% Latency (ms)		50% Latency (ms)	
(Mbps)	# Flows	Rate (Mbps)	Retrans	1MB RPC	10KB RPC	1MB RPC	10KB RPC
1000	1	925	0	9.5		9.0	
1000	2	922	0	19.0	0.7	13.0	0.2
1000	5	931	0	47.9	0.9	43.0	0.5
1000	9	945	1493	336.0	207.0	54.0	0.8
5000	1	4600	0	4.1		1.7	
5000	2	4600	0	4.7	0.6	1.9	0.2
5000	5	4670	0	12.4	1.0	7.7	0.2
5000	9	4630	0	18.5	0.8	15.5	0.2

Table 4: Ingress, 1 and 5 Gbps rate limits, DCTCP

Conclusions

Egress BPF based NRM is able to prevent standing queues and as a result small RPCs get higher rates and lower tail latencies. The best NRM results are achieved when using DCTCP. There are some cases when using HTB for rate limiting achieves better results, but we are only starting to explore NRM policing algorithms.

Finally, NRM BPF is a great platform for experimenting with policing algorithms for network resource management.

Future Work

There is still a lot of work to do. We plan to do the following:

1. Explore new policing algorithms
2. Tests concurrent flows with different RTTs
3. Explore using RTT in the policing algorithms
4. Test using multiple scopes such as multiple cgroups and multiple scopes per flow
5. Test concurrent flows with different TCP variants. For example DCTCP and Cubic.
6. Explore mechanisms for notifying senders when doing ingress NRM and the packets do not support ECN.
7. Explore using a host ingress scope to decrease incast losses.
8. Rather than reacting to the packets once TCP sends them, explore checking how much data could be sent before TCP sends an skb. This would allow us to reduce the skb size in order to prevent just dropping the skb.

References

1. Brakmo, L. 2017. Network Testing with Netesto. *Netdev 2.1 Technical Conference*, Montreal, Canada.
2. Mohammad Alizadeh , Albert Greenberg , David A. Maltz , Jitendra Padhye , Parveen Patel , Balaji Prabhakar , Sudipta Sengupta , Murari Sridharan, Data center TCP (DCTCP), *Proceedings of the ACM SIGCOMM 2010 conference*, August 30-September 03, 2010, New Delhi, India
3. Sangtae Ha, Injong Rhee and Lisong Xu, CUBIC: A New TCP-Friendly High-Speed TCP Variant, *ACM SIGOPS Operating System Review*, Volume 42, Issue 5, July 2008, Page(s):64-74, 2008.