TCP-BPF: Programmatically tuning TCP behavior through BPF

Lawrence Brakmo

Facebook
Menlo Park, USA
brakmo@fb.com

Abstract

TCP-BPF, also known as BPF sock_ops, is a new mechanism that supports setting TCP parameters through BPF programs. In addition to the flexibility of using BPF programs to determine the optimal parameters for a TCP connection, TCP-BPF also adds support for new parameters such as SYN RTO and SYN-ACK RTO. A TCP-BPF program can use socket information, such as IP addresses and port numbers, for determining optimal values for TCP parameters. For example, setting buffer sizes to 80KB, SYN and SYN-ACK RTOs to 10ms and TCP's congestion control to DCTCP when both hosts reside within the same ECN supporting datacenter.

Keywords

BPF, TCP, Linux.

Introduction

Linux provides global, per name-space and per-connection parameters to fine tune TCP's behavior. Examples of these parameters are buffer sizes, congestion control, and window clamp. Global parameters are useful for setting default parameters, but cannot be tuned for each connection's characteristics. For example, large RTT connections require more buffer space than short RTT connections. Although network namespaces support finer parameter granularity, it requires that all TCP connections in a namespace have similar characteristics in order to optimize parameter values.

Linux provides two mechanisms for setting per-connection TCP parameters: the setsockopt function and ip-route. The first mechanism, setsockopt, requires changes to the application, and more importantly, the naïve implementation binds the policy to the application. The second mechanism, ip-route, can set the parameters based on route prefixes. This is much more limited that what can be achieved in a BPF program that has access to potentially more connection information.

Once there is a framework for programmatically setting parameters, one starts to see many new opportunities for parameters. SYN and SYN-ACK RTOs are prime examples

In addition to support setting TCP parameters statically based on initial connection information, such as IP addresses and port numbers, TCP-BPF can also support dynamical approaches. For example, consider the initial congestion window (INIT_CWND). One could write a TCP-BPF program to explore a range of INIT_CWND values in order to arrive to per-subnet optimal INIT_CWND values. This could be more than just a one-time experiment; the probing and exploration could always be present (at appropriate rate) in order to adapt to changing workloads or hardware.

Of course, the uses of TCP-BPF go beyond its initial goal of programmatically tuning TCP parameters. It can support experimentation where one can try different parameter values as well as collecting the necessary data in order to evaluate the experiment. For example, fine tuning INIT_CWND per IP prefix.

Overview

TCP-BPF is a new BPF program type that attaches to a cgroupv2. As a result, it can support having different policies for applications by running the applications in different groups.

Existing BPF program types follow the model that there is a BPF program per entry (or calling) point. Thus, the BPF program typically knows where it is being called from. In contrast, a TCP-BPF program can be called from many different points from within the TCP execution path. An *op* value is used to indicate either where it is being called from or what is expected from the TCP-BPF program (more on this later). One reason for this approach is that in order to achieve some results, the TCP-BPF program needs to coordinate among separate calls. For example, if we want to optimize a connection whose endpoints are within the same DC, we need to set SYN RTO and SYN-ACK RTOs (a new parameter introduces by TCP-BPF) to small values as well as set small socket buffer sizes. Using separate BPF programs would introduce the possibility that we use programs that are not compatible with each other. In addition, it would be a big pain having to load multiple TCP-BPF programs.

In addition to the *op* type, TCP-BPF program can also get direct access to a subset of the socket state. Finally, there are also the bpf helper functions getsockops and setsockops that can be used to get and set some TCP parameters.

As mentioned earlier, there are two types of TCP-BPF *ops*. The first type requests a particular value and its effect occurs through its return value. Examples of this type of *op* are:

- BPF_SOCK_OPS_TIMEOUT_INIT
- BPF_SOCK_OPS_RWND_INIT
- BPF_SOCK_OPS_NEEDS_ECN
- BPF_SOCK_OPS_BASE_RTT

The second type indicates where the TCP-BPF program is being called from and its effects occur by changes to the connection state. These changes can be achieved through calls to setsockopt or by directly changing a socket state value. Examples of this type of *op* are:

- BPF_SOCK_OPS_TCP_CONNECT_CB
- BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB
- BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB

Implementation

TCP-BPF programs are implemented as a new BPF program type that attaches to a cgroupv2. The data structure associated with the program is called *bpf_sock_ops* when seen by the TCP-BPF program and looks like:

```
struct bpf_sock_ops {
    __u32 op;
    union {
        __u32 reply;
        __u32 replylong[4];
    };
    __u32 family;
    __u32 remote_ip4;       /* Stored in NBO */
    __u32 local_ip4;        /* Stored in NBO */
    __u32 remote_ip6[4];    /* Stored in NBO */
    __u32 local_ip6[4];     /* Stored in NBO */
    __u32 remote_port;      /* Stored in NBO */
    __u32 local_port;       /* stored in HBO */
/* where NBO = Network Byte Order
   and   HBO = Host Byte Order
 */
};
```

Current *ops* return their value through the *reply* field. The *replylong* field is there to support future ops that may require larger return values.

The data structure associated with the kernel is called *bpf_sock_ops_kern* and looks like:

```
struct bpf_sock_ops_kern {
    struct   sock *sk;
     u32     op;
     union {
         u32 reply;
         u32 replylong[4];
     };
};
```

The fields *op*, *reply* and *replylong* in the bpf_sock_ops structure are R/W and map directly to the respective fields in the kernel structure. The other fields are Read Only and are mapped directly by the BPF framework into direct accesses to the respective sock structure fields.

The function *tcp_call_bpf()*, defined in include/net/tcp.h , is used to call the TCP-BPF program. The function declaration is:

```
static inline int tcp_call_bpf(struct sock *sk,
                               int op);
```

There are helper functions for some of the ops defined in include/net/tcp.h

As mentioned earlier, there are two TCP-BPF helper functions.

- **`bpf_setsockopt()`** is similar to the standard Linux setsockopt but only supports a limited number of options. Among the options supported are:
    - `SO_RCVBUF`
    - `SO_SNDBUF`
    - `SO_MAX_PACING_RATE`
    - `SO_PRIORITY`
    - `SO_RCVLOWAT`
    - `SO_MARK`
    - `TCP_CONGESTION`
    - `TCP_BPF_IW`
    - `TCP_BPF_SNDCWND_CLAMP`

    Where the standard options have the same meaning as in Linux. The two new options have the following meanings:

    `TCP_BPF_IW` – set initial snd_cwnd to the specified value. If the connection has already sent packets, then this is a nop.

    `TCP_BPF_SNDCWND_CLAMP` – sets the socket snd_cwnd_clamp and the snd_ssthresh to the specified value.

- **`bpf_getsockopt()`** is similar to the standard Linux getsockopt but currently only supports one option:
    - `TCP_CONGESTION`

Usage

TCP_BPF uses cgroupsv2 BPF framework, so it is necessary to create a group and attach the relevant processes to the group. For example:

```
mkdir -p /tmp/cgroupv2
mount -t cgroup2 none /tmp/cgroupv2
mkdir -p /tmp/cgroupv2/foo
bash
echo $$ >> /tmp/cgroupv2/foo/cgroup.procs
```

Any program started with the current shell will belong the the cgroupv2 *foo*. If you are using netperf/netserver or iperf3 they should be started through the current shell.

To attach a TCP-BPF program to the cgroupv2 *foo*, one can use the following command:

```
load_sock_ops [-l] <cgroupv2> <tcp-bpf program>
```

For our example:

```
load_sock_ops -l /tmp/cgroupv2/foo tcp_iw_kern.o
```

`tcp_iw_kern` is a TCP-BPF program that only affects flows where one of the ports is 5560 and sets TCP parameters that are appropriate for larger RTTs: it set TCP's initial congestion window of active opened flows to 40, the receive windows to 40 and send and receive buffers to 1.5MB so the flow can achieve better throughput.

To remove/unload a TCP-BPF program

```
load_sock_ops -r <cgroupv2>
```

Example: Tuning for DC

`tcp_clamp_kern` is a TCP-BPF sample program to optimize parameters for flows within a DC. The TCP-BPF program assumes that if the first 5.5 bytes of the IPv6 address match, then both hosts are within the same DC. In such a case, the TCP-BPF program set the SYN and SYN-ACK RTOs to 10ms, send and receive buffers to 150KB. The actual program is shown in Appendix A.

To measure the benefits of fine tuning some parameters for intra-DC traffic we had 3 hosts send to one host (see Figure 1). Each host established X 1MB back-to-back RPC flows plus 1 streaming flow plus 2 10KB flows, for X in {1, 2, 4, 8, 16}. Figure 2 shows the average rate of the 1MB RPCs for baseline and using the clamp bpf program. The average 1MB RPC rates are the about the same for baseline and using the clamp TCP-BPF program. However, the number of packet retransmitted is quite different. It is zero when using the TCP-BPF program for most of the experiments. As a result, the 99% Latencies are about half of baseline when running the TCP-BPF program (except for the last experiment where the retransmissions are the same for both).

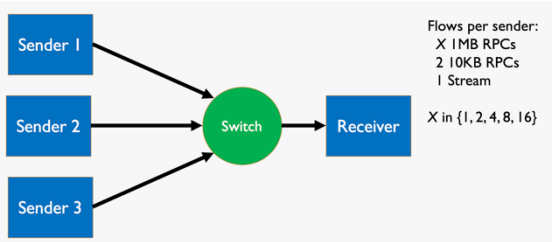

Figure 1: Experiment scenario

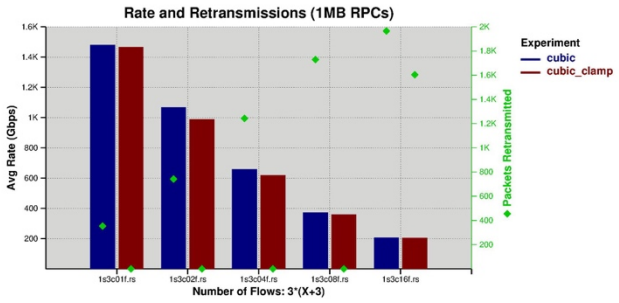

Figure 2: Rate and retransmissions for 1MB RPCs

More interestingly, Figure 3 shows the rate and retransmissions for 10KB RPCs. When using the TCP-BPF program. The 10KB RPCs get much higher bandwidths when running the TCP-BPF program, staring 4x the bandwidth, as compared to baseline, when there are 12 flows.

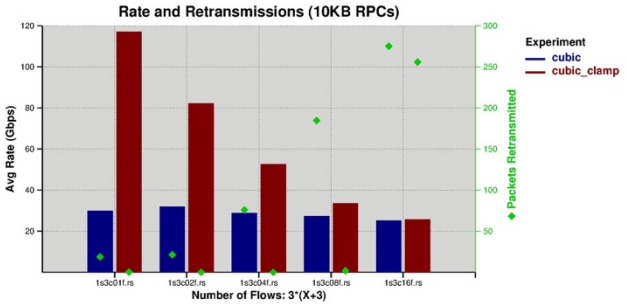

Figure 3: Rate and retransmissions for 10KB RPCs

Finally, Figure 4 shows the median an 99% latencies for the 10KB Flows. Interestingly, the 99% latencies are at least half of baseline as long as the TCP-BPF program is able to prevent packet losses.

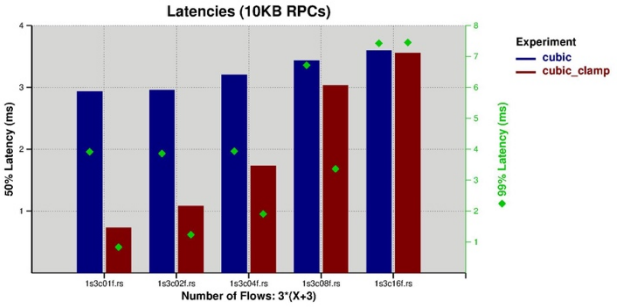

Figure 4: 50% and 99% 10KB Latencies

The parameter mostly responsible for the improvements is the cwnd window clamp. For this example, the TCP-BPF program sets it to 100, large enough to allow one flow to achieve close to 10Gbps, but small enough that allows many concurrent flows to coexist without packet drops. The other parameters used help with memory and the case when the SYN or SYN-ACK packets are dropped.

Although there are other mechanisms for setting the cwnd clamp (setsockopt and ip-route), TCP-BPF is more convenient because it doesn't require any changes to the application and the same TCP-BPF program can be used everywhere, whereas ip-route would need different settings in each DC.

Next Steps

There are various TCP-BPF enhancements in the pipeline. Among them are:

1. Make more TCP state available to TCP-BPF programs. For example cwnd, ssthresh, etc.
2. Add more functionality to bpf_setsockopt and bpf_getsockopt. Examples: setting IPv6 class, setting flowlabels, etc.
3. Add more entry points to TCP-BPF. For example, when an RTO fires, when a packet is retransmitted, when a packet is received or sent, etc. In order to reduce overhead, there will be a bitmap (per socket) associated with these calling points to determine whether they should be called.

   The bitmap is initialized to zeroes, but can be set (per flow) in the BPF program when the connection is established. In some cases, entry points could be enabled statistically (i.e. 0.01% of flows) enabled to allow collection (or analysis) of flow behavior. In many cases the RTO and retransmits could be enabled for all flows (assuming they are rare) and the TCP-BPF program could trigger specific behaviors, such as changing flow labels or congestion algorithm, when the RTO rates or retransmits are too high.

   It would enable new behaviors, such as dynamically, i.e. not only at connection establishment time, tuning parameters based on the effect on the flow. For example, adjusting the initial congestion window per subnet based on whether it leads to losses or not.
4. Add support for handling TCP packet header options. The idea is to be able to implement new TCP header options in BPF programs. This is especially useful in environments where people have their own local options since it decreases the need of maintaining local patches.

Availability

TCP-BPF is available starting at kernel version 4.13

I would like to recognize the helpful feedback provided by Alexei Starovoitov and Daniel Borkmann.

References

1. Alexei Starovoitov, BPF in-kernel Virtual Machine, Netdev 0.1 Technical Conference, Ottawa, Canada. https://netdevconf.org/0.1/docs/starovoitov-bpf_netdev01_2015feb13.pdf

Appendix A

```
SEC("sockops")
int bpf_clamp(struct bpf_sock_ops *skops
{
    int bufsize = 150000;
    int to_init = 10;
    int clamp = 100;
    int rv = 0;
    int op;

    /* Check that both hosts are within same datacenter. For
     * this example it is the case when the first 5.5 bytes of
     * their IPv6 addresses are the same.
     */
    if (skops->family == AF_INET6 &&
        skops->local_ip6[0] == skops->remote_ip6[0] &&
        (bpf_ntohl(skops->local_ip6[1]) & 0xfff00000) ==
        (bpf_ntohl(skops->remote_ip6[1]) & 0xfff00000)) {
        switch (op) {
        case BPF_SOCK_OPS_TIMEOUT_INIT:
            rv = to_init;
            break;
        case BPF_SOCK_OPS_TCP_CONNECT_CB:
            /* Set sndbuf and rcvbuf of active connections */
            rv = bpf_setsockopt(skops, SOL_SOCKET, SO_SNDBUF,
                                &bufsize, sizeof(bufsize));
            rv = rv + bpf_setsockopt(skops, SOL_SOCKET,
                                     SO_RCVBUF, &bufsize,
                                     sizeof(bufsize));
            break;
        case BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB:
            rv = bpf_setsockopt(skops, SOL_TCP,
                                TCP_BPF_SNDCWND_CLAMP,
                                &clamp, sizeof(clamp));
            break;
        case BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB:
            /* Set cwnd clamp and sndbuf, rcvbuf of passive
             * connections
             */
            /* See actual program for this code */
        default:
            rv = -1;
    } else { rv = -1; }
    skops->reply = rv;
    return 1;
}
```