OpenAnolis
龙蜥社区

毛文安

阿里云高级技术专家
龙蜥社区系统运维SIG

基于eBPF的ping探测

OpenAnolis
龙蜥社区

议程

1 **Ping毛刺的产生**
介绍网络延时抖动的背景和分类

2 **eBPF的开发方式**
介绍eBPF的各种开发方法

3 **基于LCC进行ping探测实践**
如何实现ping探测

01

Ping毛刺的产生

认识Ping毛刺

```
64 bytes from [redacted] icmp_seq=12 ttl=64 time=0.179 ms
64 bytes from [redacted] icmp_seq=13 ttl=64 time=0.201 ms
64 bytes from [redacted] icmp_seq=14 ttl=64 time=0.188 ms
64 bytes from [redacted] icmp_seq=15 ttl=64 time=0.193 ms
64 bytes from [redacted] icmp_seq=16 ttl=64 time=0.186 ms
64 bytes from [redacted] icmp_seq=17 ttl=64 time=0.214 ms
64 bytes from [redacted] icmp_seq=18 ttl=64 time=0.158 ms
64 bytes from [redacted] icmp_seq=19 ttl=64 time=130 ms
64 bytes from [redacted] icmp_seq=20 ttl=64 time=731 ms
64 bytes from [redacted] icmp_seq=21 ttl=64 time=0.213 ms
64 bytes from [redacted] icmp_seq=22 ttl=64 time=0.181 ms
64 bytes from [redacted] icmp_seq=23 ttl=64 time=0.191 ms
64 bytes from [redacted] icmp_seq=24 ttl=64 time=0.182 ms
64 bytes from [redacted] icmp_seq=25 ttl=64 time=0.171 ms
64 bytes from [redacted] icmp_seq=26 ttl=64 time=0.187 ms
```

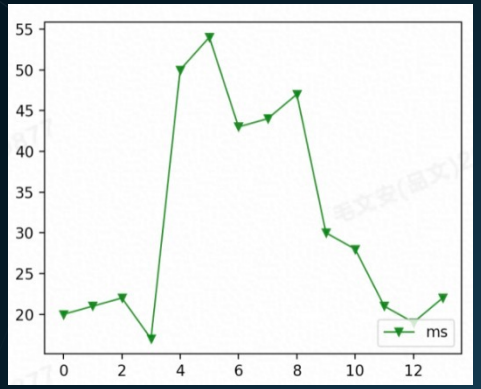
认识网络抖动

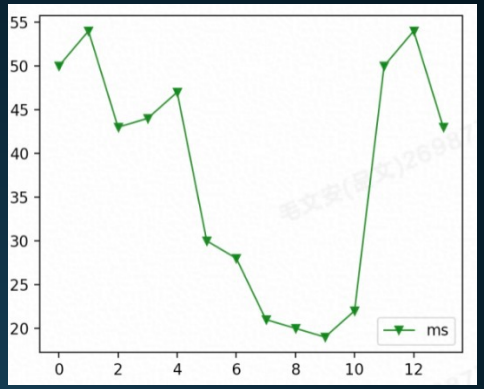
ECS服务HTTP请求平均延迟正常水平维持在10ms，在某个时间点突然发生抖动，整体延迟增加至100ms，随后马上恢复。

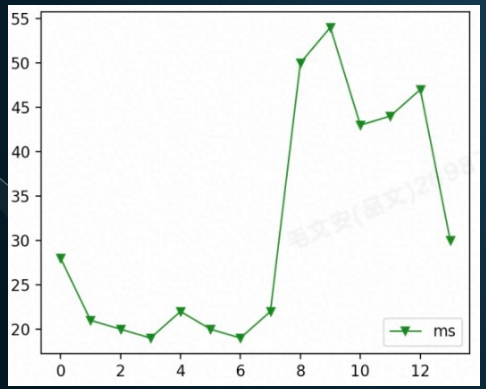
ECS访问另一台RDS，在某个时间点突然出现大量业务日志打印的timeout，持续时间为10秒，随后马上恢复。

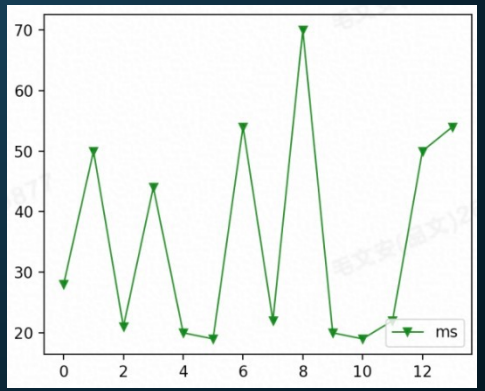
ECS服务的pps正常水平维持在800K，在某个时间点突然发生抖动，pps降低至400K，随后马上恢复。

网络抖动的图形表示

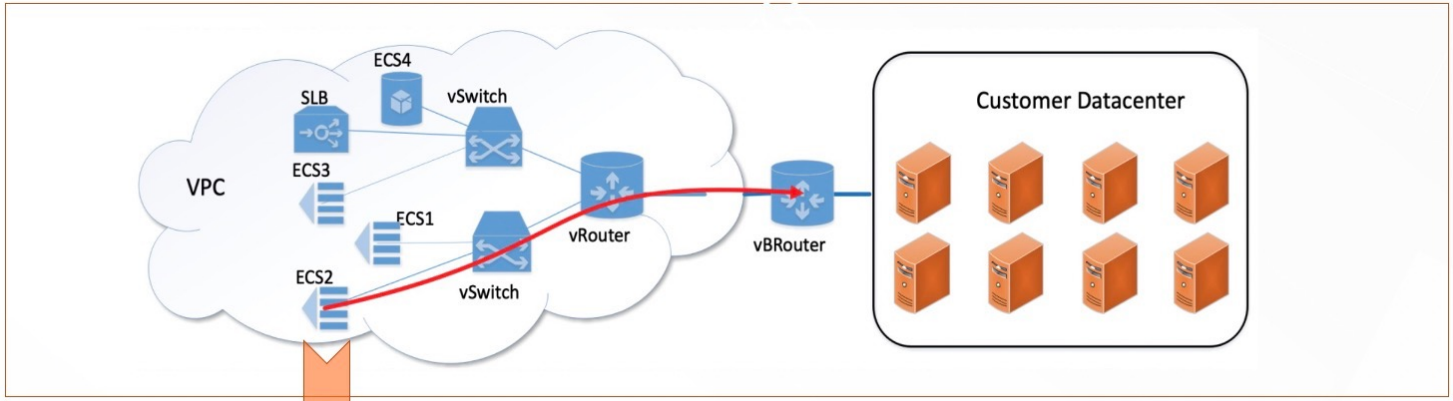

55
50
45
40
35
30
25
20
0
2
4
6
8
10
12
ms


55
50
45
40
35
30
25
20
0
2
4
6
8
10
12
ms

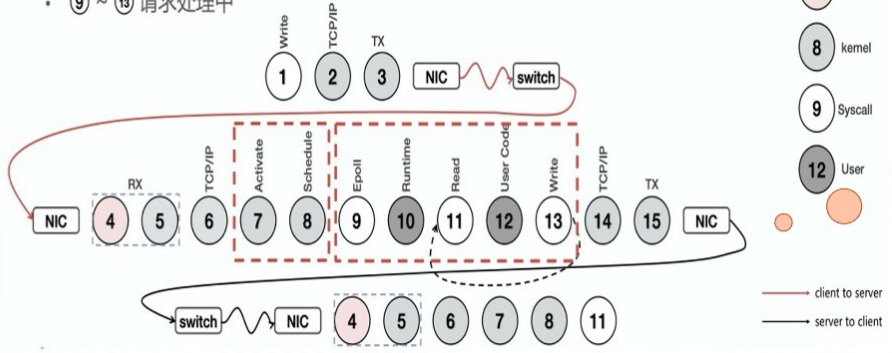

55
50
45
40
35
30
25
20
0
2
4
6
8
10
12
ms


70
60
50
40
30
20
0
2
4
6
8
10
12
ms

影响网络抖动因素



- ⑦ ~ ⑧ Socket 数据 ready，进程激活等待运行
- ⑨ ~ ⑬ 请求处理中



主机内部可能产生延迟的几个阶段：中断/软中断处理，唤醒进程/调度，qdisc排队等

抖动的分类和解法

监控　　探测　　观测&诊断

02

eBPF的开发方式

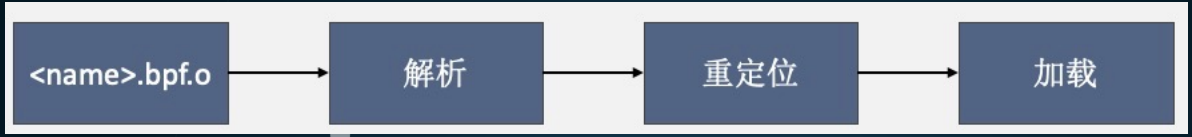
BPF开发常用方式

原生libbpf，无CO-RE（内核samples/bpf示例）

BCC（bpf compile collection）

BPF CO-RE（include *skel.h,自己编写的bpf_core_read代码）

Libbpf核心功能



1. 二进制解析：解析每个段，包括：
   .btf、.<prog_text>、.map、.data、.rodata、.bss等
2. 重定位：根据bpf.o的btf信息和内核现有的btf信息进行重定位、其它段的重定位
3. 加载：主要是map和bpf程序

BPF支持 CO-RE(Compile Once, Run everywhere)

为什么需移植

1. 需要访问原始的内核数据，不同内核版本数据的内存布局不同
2. 需要基于一个内核的两种配置来运行程序

关键组件：

1. BTF: 描述内核镜像，获取内核及BPF程序类型和代码的关键信息(http://pylcc.openanolis.cn/)
2. Clang释放bpf程序重定位信息到.btf段
3. Libbpf CO-RE根据.btf段重定位bpf程序

重定位:

1. 结构体重定位，BTF， Clang通过__builtin_preserve_access_index()记录成员偏移量

```
u64 inode = task->mm->exe_file->f_inode->i_ino;
u64 inode = BPF_CORE_READ(task, mm, exe_file, f_inode, i_ino);
```

2. map fd、全局变量（data、bss、rodata）、extern的变量重定位，依赖ELF

```
skel->rodata->my_cfg.feature_enabled = true;
skel->rodata->my_cfg.pid_to_filter = 123;
extern u32 LINUX_KERNEL_VERSION  __kconfig;
extern u32 CONFIG_HZ             __kconfig;
```

3. 子函数重定位，依赖ELF，CO-RE将所有子程序拷贝到主程序所在区域, 如always_inline函数

Libbpf CO-RE开发步骤

OpenAnolis 龙蜥社区

1.生成带所有内核类型的头文件vmlinux.h

bpftool btf dump file vmlinux format c > vmlinux.h

2.使用Clang(版本10或更新版本)将BPF程序的源代码编译为.o对象文件；

3.从编译好的BPF对象文件中生成BPF skeleton 头文件 bpftool gen命令生成；

4.在用户空间代码中包含生成的BPF skeleton 头文件；

5.编译用户空间代码，这样会嵌入BPF对象代码，后续就不用发布单独的文件。

生成的BPF skeleton 使用如下函数触发相应的阶段：
<name>__open() – 创建并打开 BPF 应用，之后可以设置skel->rodata 变量
<name>__load() – 初始化，加载和校验BPF 应用部分;
<name>__attach() – 附加所有可以自动附加的BPF程序 (可选，可以直接使用libbpf API作更多控制);
<name>__destroy() – 分离所有的 BPF 程序并使用其使用的所有资源。

BCC到libbpf的转换 头文件


```
#include <bcc/proto.h>
#include <uapi/linux/ip.h>
#include <uapi/linux/icmp.h>
#include <uapi/linux/tcp.h>
#include <uapi/linux/udp.h>
#include <net/inet_sock.h>
#include <linux/netfilter/x_tables.h>
#include <linux/virtio.h>
#include <linux/netdevice.h>
```

BCC

需要手动添加所需的内核头文件

```
#include <vmlinux.h>
// 常用的宏定义，比如bpf_printk、offsetof等
#include <bpf/bpf_helpers.h>
// 读取内核地址的宏定义方法，比如BPF_PROBE_READ
#include <bpf/bpf_core_read.h>
// 用于从struct pt_regs获取参数的方法集合
#include <bpf/bpf_tracing.h>
// 其它的自定义头文件
#include ”rtrace.h"
```

libbpf

vmlinux.h包含了所有的内核类型，但是不包含#define宏定义

BCC到libbpf的转换 bpf maps定义和访问


```
BPF_HASH(cur_ipt_do_table_args, u32, struct
ipt_do_table_args);

some_map.operation(some, args)
```

BCC

BCC提供了map的默认大小，即10240

```
struct bpf_map_def SEC("maps") cur_ipt_do_table_args = {
        .type = BPF_MAP_TYPE_HASH,
        .key_size = sizeof(u32),
        .value_size = sizeof(struct ipt_do_table_args),
        .max_entries = 10240,
};
   bpf_map_operation_elem(&some_map, some, args);
```

libbpf

需要显示地指定map的大小

BCC到libbpf的转换 结构体成员访问

OpenAnolis 龙蜥社区

tsk->parent->pid

BCC

BCC自动将其转换成两次bpf_probe_read函数调用

支持CO-RE：

BPF_CORE_READ(tsk, parent, pid)

不支持CO-RE：

struct task_struct pt;

bpf_probe_read(&pt,sizeof(struct task_struct), tsk->parent);

pid_t pid = bpf_probe_read(&pid,sizeof(pid_t),pt->pid)

libbpf

需要显示地指定map的大小

BCC到libbpf的转换 tracepoint和 kprobe


```
TRACEPOINT_PROBE(net, netif_rx)

int kprobe__netif_rx(struct pt_regs *ctx,
struct sk_buff *skb)
```

BCC

```
SEC("tracepoint/net/netif_rx")
int tp_netif_rx(struct trace_event_raw_netif_rx  *args)

SEC("kprobe/netif_rx")
int BPF_KPROBE(kprobe__netif_rx, struct sk_buff
*skb)
```

libbpf

BCC到libbpf的转换 bpf程序配置

bcc对bpf程序配置主要包括两方面：

1. #if控制函数
2. #if控制函数内语句

```
#if TRACE_L2
int kprobe__netif_rx(struct pt_regs *ctx, struct sk_buff *skb)
{
    return do_trace(ctx, skb, __func__+8, NULL);
}
#endif
```

```
#if TRACE_CPU
    event->cpuid = bpf_get_smp_processor_id();
#endif
```

BCC

1. bpf_object__find_program_by_name和bpf_program__set_autoload来控制函数是否加载
2. 在不支持bpf全局变量的内核版本上可通过map的方式控制函数内语句

```
for (i = 0; i < ENABLE_HOOK_FUNC_NUM; i++)
{
    prog = bpf_object__find_program_by_name(obj->obj, ENABLE_HOOK_FUNC[i]);
    if (prog)
    {
        bpf_program__set_autoload(prog, true);
    }
}
```

```
if (arg->cpuid != 0)
    event->cpuid = bpf_get_smp_processor_id();
```

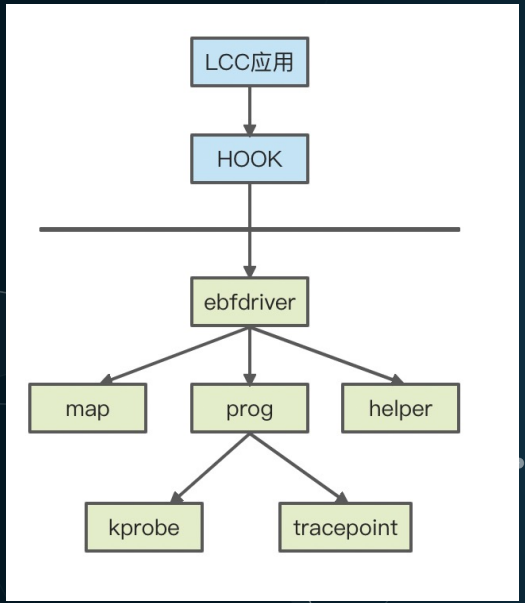
libbpf

目标：一个eBPF程序，高中低内核版本运行无忧

我们是这么干的：

- 低版本内核：3.x 移植eBPF能力
- 中版本内核：4.x 自动导出vmlinux-btf (http://pylcc.openanolis.cn/)，可配置的btf路径，支持libbpf CO-RE
- 高版本内核：5.x 自动化CO-RE代码框架，支持容器化的本地和远程编译，高级语言极简编写能力

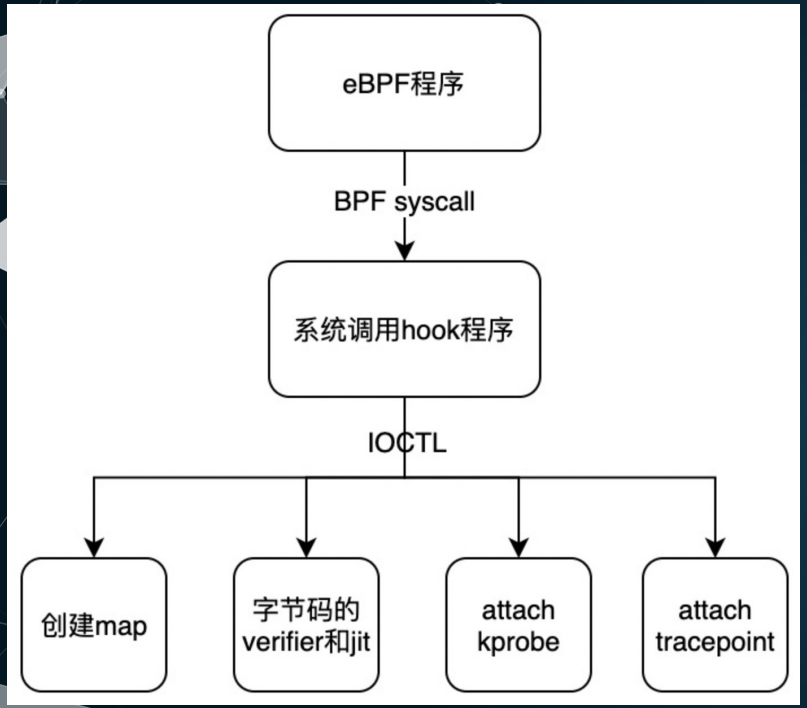
3.10 eBPF移植



1. 目前基于eBPF编写的程序只能在高版本内核（支持eBPF的内核）上运行，无法在不支持eBPF功能的内核上运行；
2. 线上有很多Alios或者centos低版本内核需要维护
3. 存量BPF工具或项目代码，希望不做修改能跨内核运行

为此我们提出了一种在低版本内核运行eBPF程序的方法，使得二进制程序无需任何修改即可在不支持eBPF功能的内核版本上运行

3.10 eBPF运行流程



1. 利用hook程序将BPF的syscall转换成ioctl形式，将系统调用参数传递给eBPF驱动；

```
#define IOCTL_BPF_MAP_CREATE _IOW(';', 0, union bpf_attr *)
#define IOCTL_BPF_MAP_LOOKUP_ELEM _IOWR(';', 1, union bpf_attr *)
#define IOCTL_BPF_MAP_UPDATE_ELEM _IOW(';', 2, union bpf_attr *)
#define IOCTL_BPF_MAP_DELETE_ELEM _IOW(';', 3, union bpf_attr *)
#define IOCTL_BPF_MAP_GET_NEXT_KEY _IOW(';', 4, union bpf_attr *)
#define IOCTL_BPF_PROG_LOAD _IOW(';', 5, union bpf_attr *)
#define IOCTL_BPF_PROG_ATTACH _IOW(';', 6, __u32)
#define IOCTL_BPF_PROG_FUNCNAME _IOW(';', 7, char *)
#define IOCTL_BPF_OBJ_GET_INFO_BY_FD _IOWR(';', 8, union bpf_attr *)
```

2. eBPF驱动收到Ioctl请求，会根据cmd来进行相应的操作，如；

a. IOCTL_BPF_MAP_CREATE：创建map；

b. IOCTL_BPF_PROG_LOAD：加载eBPF字节码，进行字节码的安全验证和jit生成机器码；

c. IOCTL_BPF_PROG_ATTACH：将该eBPF程序attach到指定的内核函数，利用register_kprobe和tracepoint_probe_register功能完成eBPF程序的attach

03

基于pyLCC进行ping探测实践

BPF开发常用方案对比

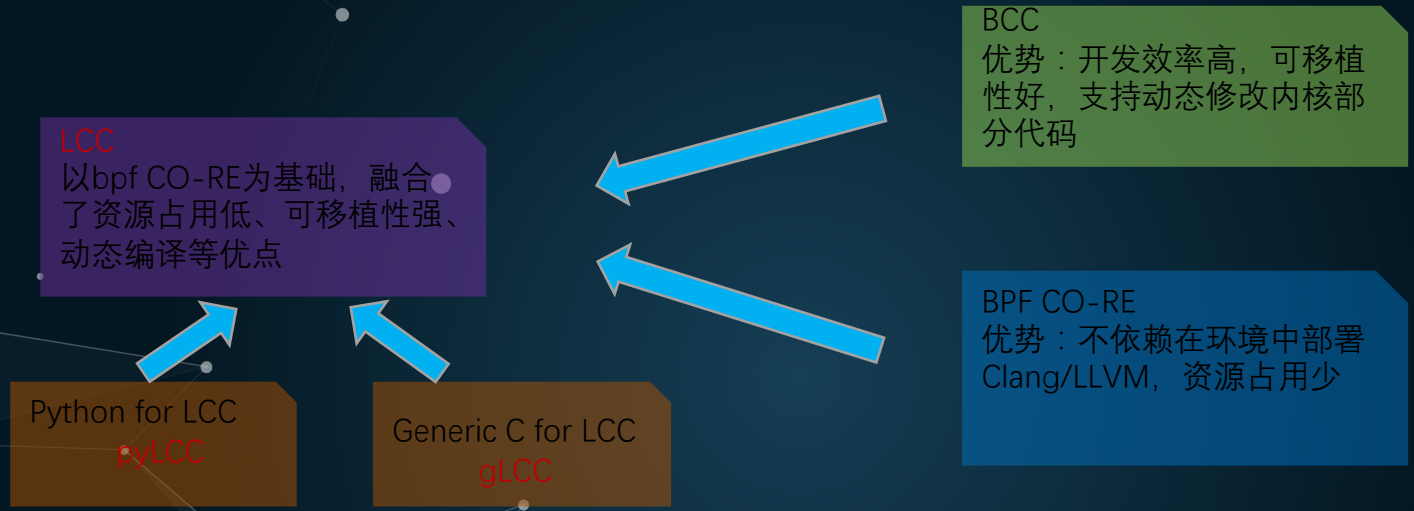
原生libbpf, 无CO-RE(内核samples/bpf示例)
优势：资源占用量低
缺点：
1、需要搭建代码工程、开发效率低；
2、不同内核版本兼容性差；

BCC（bpf compile collection）
优势：开发效率高，可移植性好，支持动态修改内核部分代码
缺点：
1、部署依赖的Clang/LLVM；
2、每次运行都要执行Clang/LLVM编译，争抢内存CPU内存等资源；
3、依赖目标环境头文件；

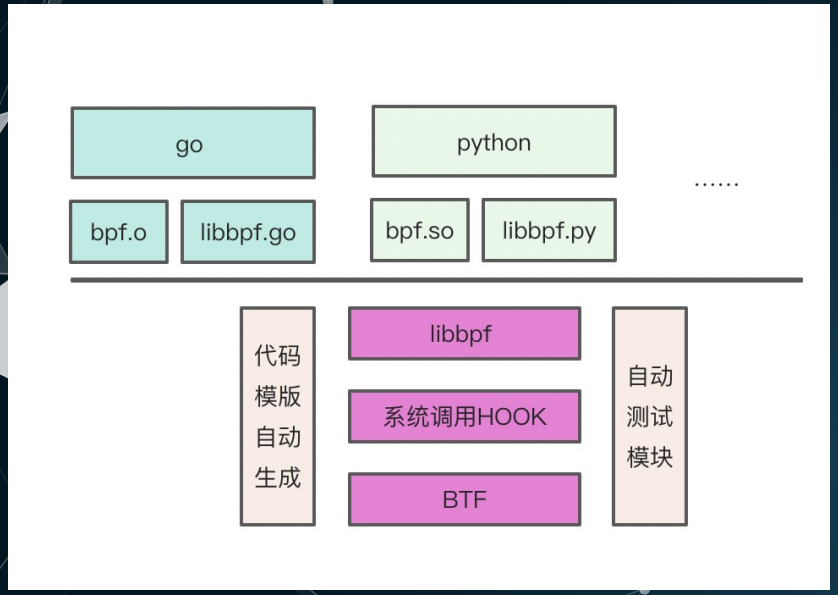
BPF CO-RE（自己编写的bpf_core_read代码）
优势：不依赖在环境中部署Clang/LLVM，资源占用少
缺点：
1、仍需要搭建编译编译工程；
2、部分代码相对固定，无法动态配置；
3、用户态开发支持信息较少，缺乏高级语言对接；

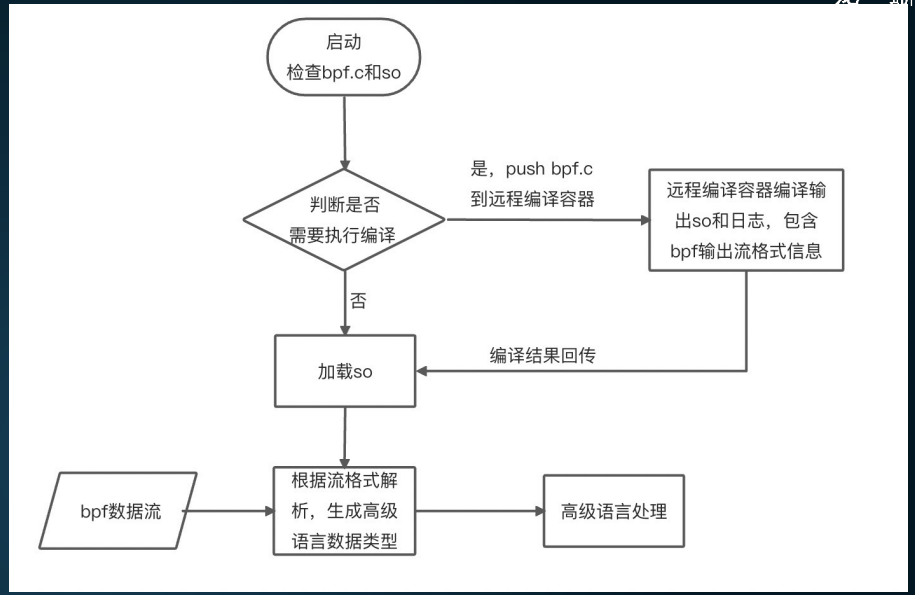
上述方案，不能很好适配生产环境中，多内核并存、快速批量部署等需求

LCC（Libbpf Compiler Collection）技术优势



pyLCC：通过将复杂编译过程交由远程容器执行，支持高级语言的极简代码编写和优秀的数据处理能力，一次编译，到处运行，节省资源损耗，使得初学者能快速入门。
gLCC：基于C语言的自动化代码框架生成，支持本地容器编译，真正libbpf CO-RE支持，免去复杂环境搭建，专注功能开发。





1. pyLCC在libbpf基础上进行封装，将复杂的编译工程交由容器执行，可选本地编译和远程编译；
2. 本地编译也会在docker容器里运行，无需搭建复杂的环境（安装clang等库）；
3.对生产环境没有类似BCC py脚本运行瞬时资源冲高的现象，没有资源损耗情况
4. 封装一些基础的执行函数，类似BCC的代码易用度

pyLCC event.py示例


```python
import ctypes as ct
from pylcc.lbcBase import ClbcBase

bpfPog = r"""
#include "lbc.h"
#define TASK_COMM_LEN 16
struct data_t {
    u32 c_pid;
    u32 p_pid;
    char c_comm[TASK_COMM_LEN];
    char p_comm[TASK_COMM_LEN];
};

LBC_PERF_OUTPUT(e_out, struct data_t, 128);
SEC("kprobe/wake_up_new_task")
int j_wake_up_new_task(struct pt_regs *ctx)
{
    struct task_struct* parent = (struct task_struct *)PT_REGS_PARM1(ctx);
    struct data_t data = {};

    data.c_pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&data.c_comm, TASK_COMM_LEN);
    data.p_pid = BPF_CORE_READ(parent, pid);
    bpf_core_read(&data.p_comm[0], TASK_COMM_LEN, &parent->comm[0]);

    bpf_perf_event_output(ctx, &e_out, BPF_F_CURRENT_CPU, &data, sizeof(data
    return 0;
}

char _license[] SEC("license") = "GPL";
"""
```

```python
class CeventOut(ClbcBase):
    def __init__(self):
        super(CeventOut, self).__init__("eventOut", bpf_str=bpfPog)

    def _cb(self, cpu, data, size):
        stream = ct.string_at(data, size)
        e = self.maps['e_out'].event(stream)
        print("current pid:%d, comm:%s. wake_up_new_task pid: %d, comm: %s"
            e.c_pid, e.c_comm, e.p_pid, e.p_comm
        ))

    def loop(self):
        self.maps['e_out'].open_perf_buffer(self._cb)
        try:
            self.maps['e_out'].perf_buffer_poll()
        except KeyboardInterrupt:
            print("key interrupt.")
            exit()

if __name__ == "__main__":
    e = CeventOut()
    e.loop()
```

1.三步搞定一个BPF应用程序，只要pip install pylcc
2.init之后，bpf已经加载到内核，只需关注数据处理。
3.不用安装clang，不用安装kernel-dev头文件，资源消耗小
4.常用有内核版本都可以支持

1. 执行pip install pylcc安装

2. xx.bpf.c编写:

```
bpfPog = r"""
#include "lbc.h"
LBC_PERF_OUTPUT(e_out, struct data_t, 128);
LBC_HASH(pid_cnt, u32, u32, 1024);
LBC_STACK(call_stack,32);
```

3.xx.py编写:

```
import time from pylcc.lbcBase import ClbcBase
class Pingtrace(ClbcBase): def __init__(self): super(Pingtrace, self).__init__("pingtrace")
```

lbc.h内容：

```
#include "vmlinux.h"
#include <linux/types.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>
#include <bpf/bpf_tracing.h>
```

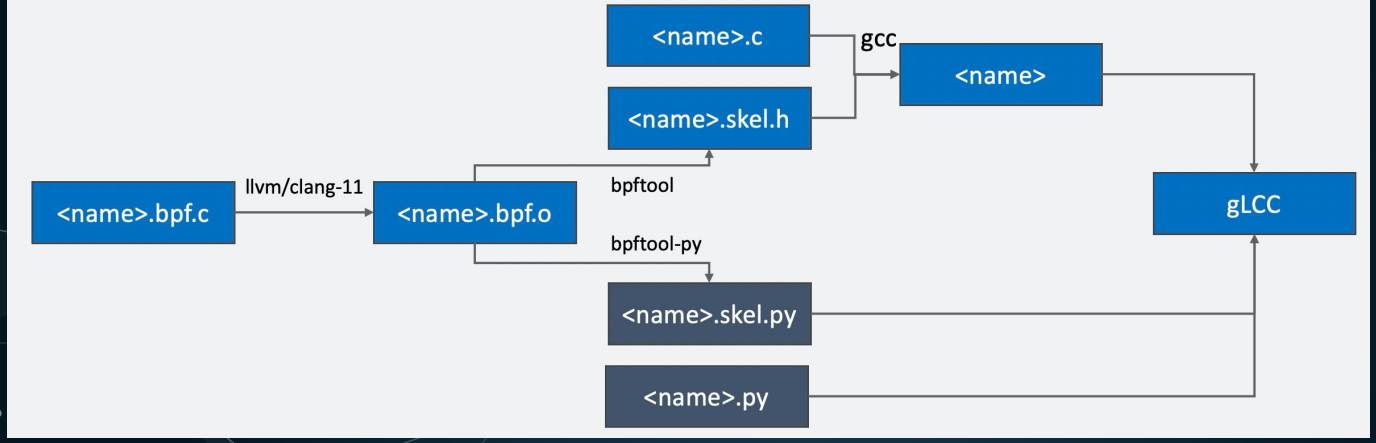
pyLCC生态支持

支持cpu 架构	x86_64/aarch64
当前支持公开发行版	4
支持公开发行版内核	1200+

Alibaba Cloud Linux

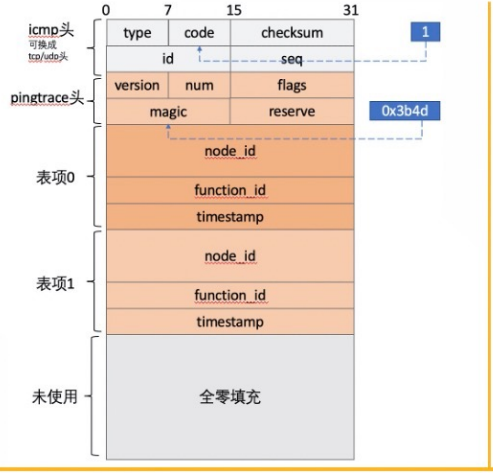
gLCC组成和框架

- 提供libbpf的python wrapper，实现将原有的bcc大部分功能自动转换成libbpf
- 能支持远程编译，提供<name>.bpf.c，远程直接编译成bpf.o程序，避免llvm/clang-11的docker环境的安装（容器大概4GB）

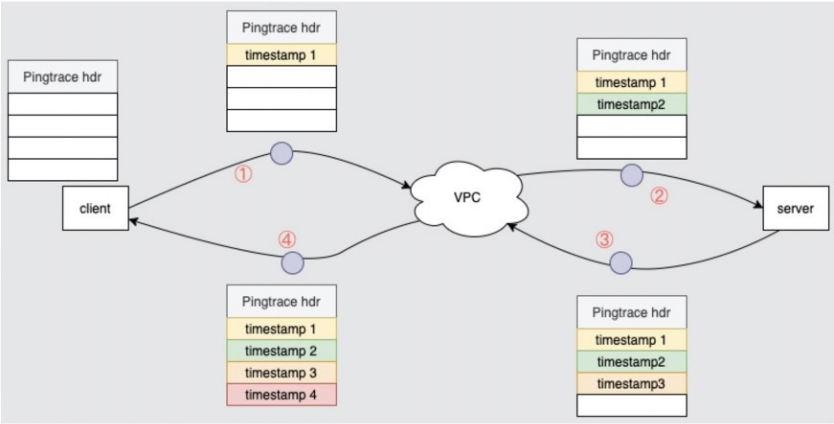


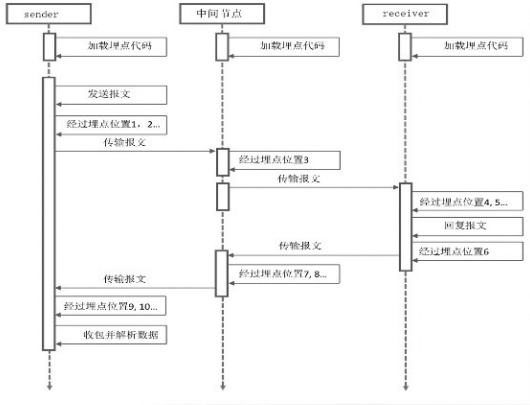
1. gLCC也是在libbpf基础上进行封装，将复杂的编译工程交由容器执行，节省资源
2. 用户空间程序为C语言，通过自动生成代码框架，让用户专注真正的功能代码开发
3. 可以动态指定btf文件路径，在低版本内核上实现CO-RE特性
4. 全量内核版本vmlinux-btf库生成
5. 把map pin到指定bpf 文件系统，延长生存周期

Ping探测流程


0
7
15
31
icmp头
可换成
tcp/udp头
type
code
checksum
1
id
seq
pingtrace头
version
num
flags
magic
reserve
0x3b4d
表项0
node_id
function_id
timestamp
表项1
node_id
function_id
timestamp
未使用
全零填充


用户态
发送端
接收端
用户态
内核态
内核态
硬件
硬件
l_rx_task_queue
l_rx_task_waking
l_tx_kern
l_rx_kern
r_tx_kern
r_rx_ringkern
l_tx_qdisc
l_tx_ring
l_rx_ring
r_tx_ring
l_rx_inlink
外部链路（网关，路由器等）
l_tx_outlink


Pingtrace hdr
timestamp 1
client
VPC
server
timestamp 2
timestamp 3
timestamp 4
timestamp2
timestamp3
①
②
③
④


sender
中间节点
receiver
加载埋点代码
发送报文
经过埋点位置1，2...
传输报文
经过埋点位置3
经过埋点位置4, 5...
回复报文
经过埋点位置6
经过埋点位置7, 8...
经过埋点位置9, 10...
收包并解析数据

Pingtrace.bpf.c 和 探测包构造

```
SEC("kprobe/__dev_queue_xmit")

SEC("tracepoint/net/netif_receive_skb")

SEC("kprobe/ip_rcv")

SEC("tracepoint/sched/sched_wakeup")

SEC("kprobe/raw_local_deliver")

SEC("tracepoint/net/net_dev_queue")
```

```
fd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

ret = setsockopt(fd, SOL_RAW, ICMP_FILTER, &filter, sizeof(filter));

setsockopt(fd, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv,
sizeof tv);
```

Icmp pingtrace & pyLCC

OpenAnolis 龙蜥社区

1、执行pip install pylcc安装

2、编写client和server py代码

内核bpf 代码片段

```
SEC("tracepoint/net/net_dev_xmit")
int net_dev_start_xmit_hook(struct net_dev_xmit_args *args)
{
        int ret;
        struct sk_buff *skb = args->skb;
        ret = tag_timestamp(&tx_map, skb, P_L_TX_DEVOUT, ICMP_ECHO);
        return 0;
}

struct net_dev_queue_args
{
        uint32_t pad[2];
        struct sk_buff *skb;
};

SEC("tracepoint/net/net_dev_queue")
int net_dev_queue_hook(struct net_dev_queue_args *args)
{
        int ret;
        struct sk_buff *skb = args->skb;
        ret = tag_timestamp(&tx_map, skb, P_L_TX_DEVQUEUE, ICMP_ECHO);
        return 0;
}

char _license[] SEC("license") = "GPL";
```

用户python代码片段

```
import ctypes as ct
from pylcc.lbcBase import ClbcBase

class Pingtrace(ClbcBase):
    def __init__(self):
        super(Pingtrace, self).__init__("pingtrace")
```

资源消耗情况

	pylcc	bcc
rss	10352	92288
vmpeak	207444	369672
vmdata	201284	363484

汇总对比如下表，同样的python应用，pylcc在cpu和mem等资源消耗均比bcc有较明显的优势

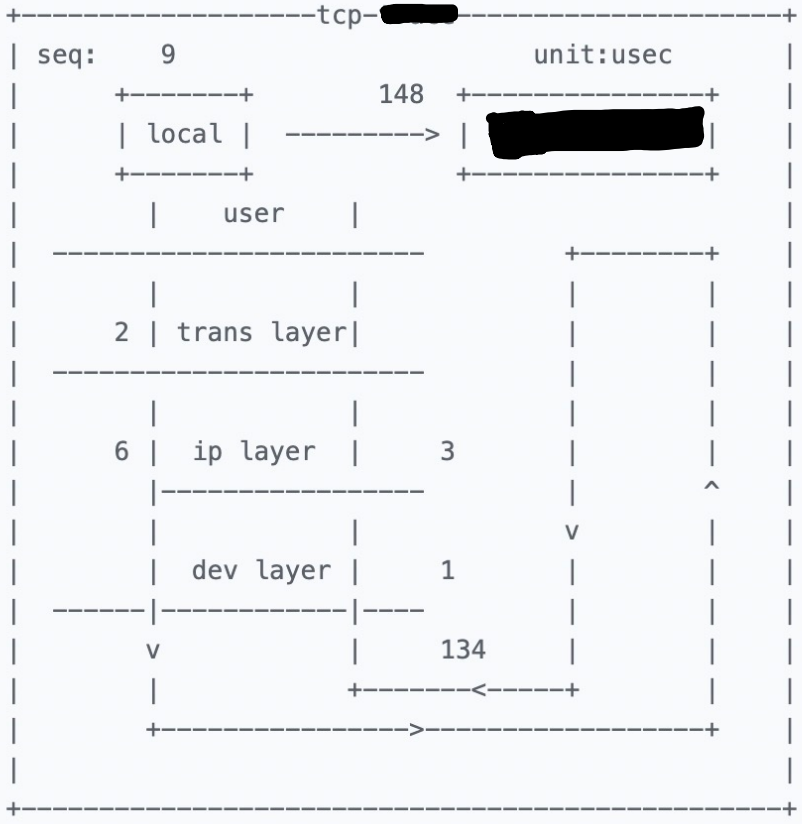
	pylcc	bcc
启动阶段cpu占用	0%	50%+
运行阶段rss占用	1	9

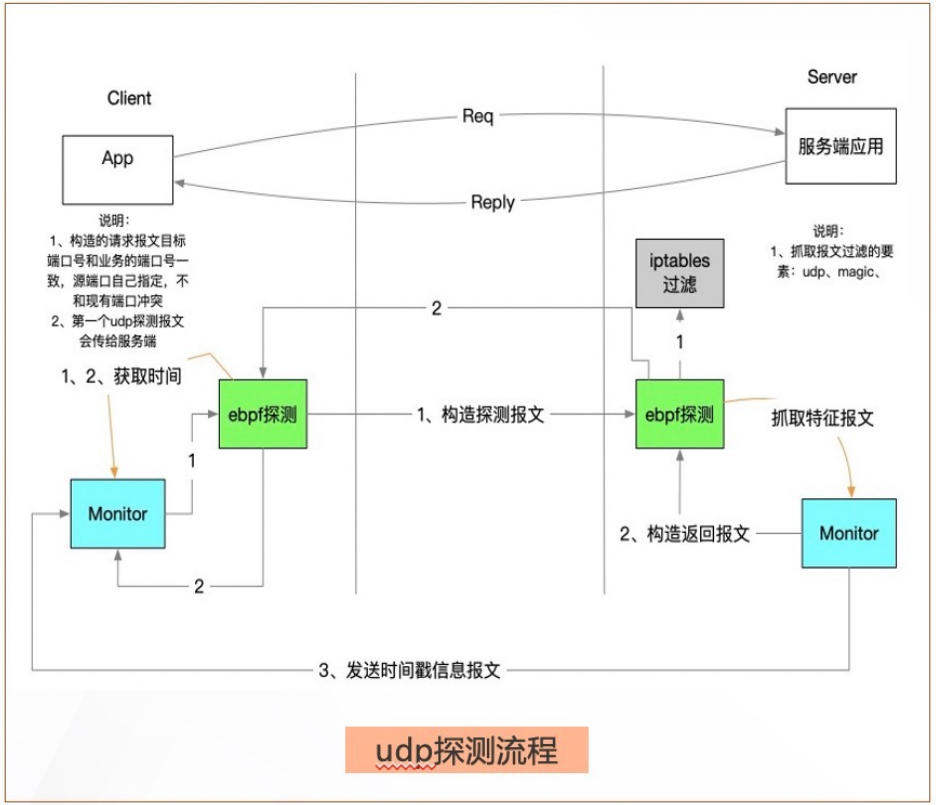
```
top - 09:05:58 up 207 days, 17:16,  2 users,  load average: 1.53, 1.36, 1.26
Tasks: 261 total,   2 running, 258 sleeping,   0 stopped,   1 zombie
%Cpu(s): 10.8 us,  2.2 sy,  0.0 ni, 85.5 id,  1.4 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem : 24130808 total,  2107008 free, 11896700 used, 10127100 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 11816644 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
2730530 root      20   0  345572  61672  28872 R  44.2  0.3   0:01.33 filelife
```

LMP项目：预热，目前我们已发布 tcp syn pingtrace

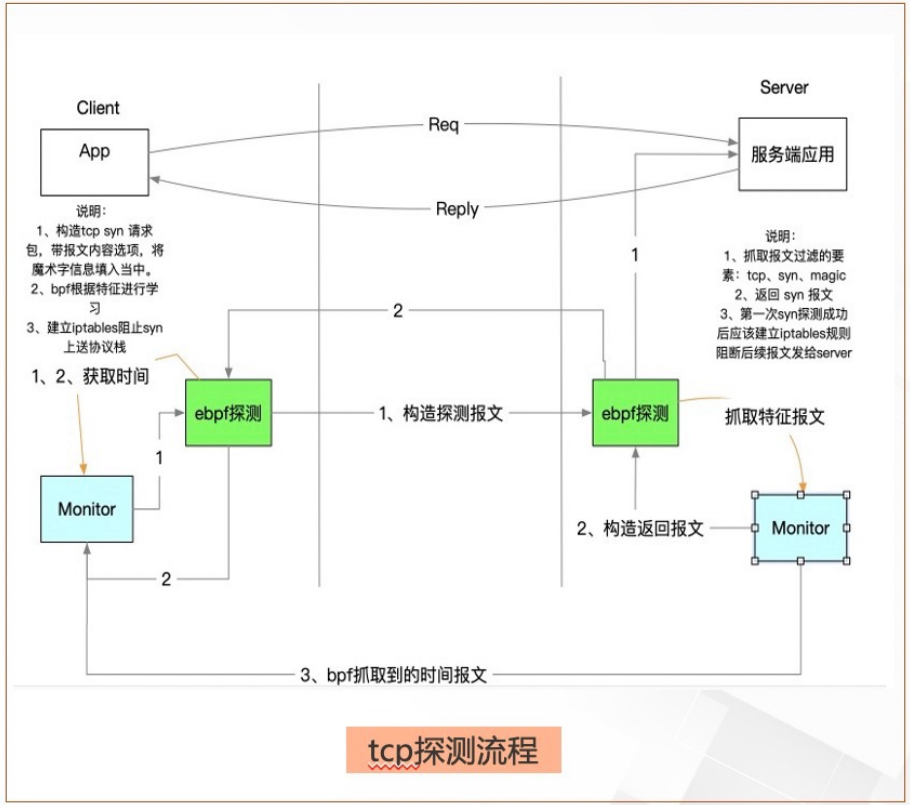
OpenAnolis 龙蜥社区

发送10个报文，输出到控制台,结果如下





udp探测流程



tcp探测流程

欢迎加入龙蜥社区
共同讨论

OpenAnolis
龙 蜥 社 区

龙蜥社区系统运维SIG
https://openanolis.cn/sig/sysom

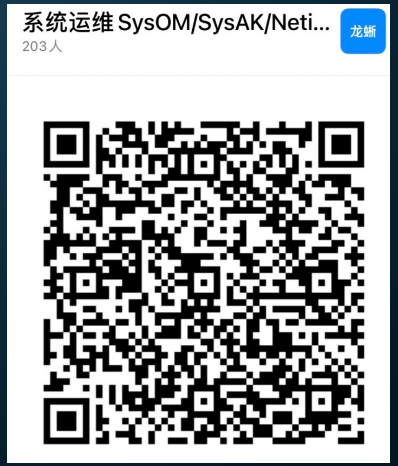
gLCC和pyLCC项目
https://gitee.com/anolis/surftrace

sysAK工具集
https://gitee.com/anolis/sysak

sysOM运维平台
https://gitee.com/anolis/sysom

eBPF学习入门
https://gitee.com/linuxkerneltravel/eBPF

微信： 明月悬空



OpenAnolis
龙蜥社区

谢谢观看