

*tcp*rt: 面向网络监控的 *eBPF* 实践

卢 烈

阿里云内核网络

CONTENT 目录

01 *tcprt* 简介

是什么？有什么用？

02 过去: *kernel module* 实现

如何实现？带来什么问题？

03 现在: *eBPF* 实现

使用 *eBPF* 有什么好处？改造带来什么新问题？

04 总结与展望

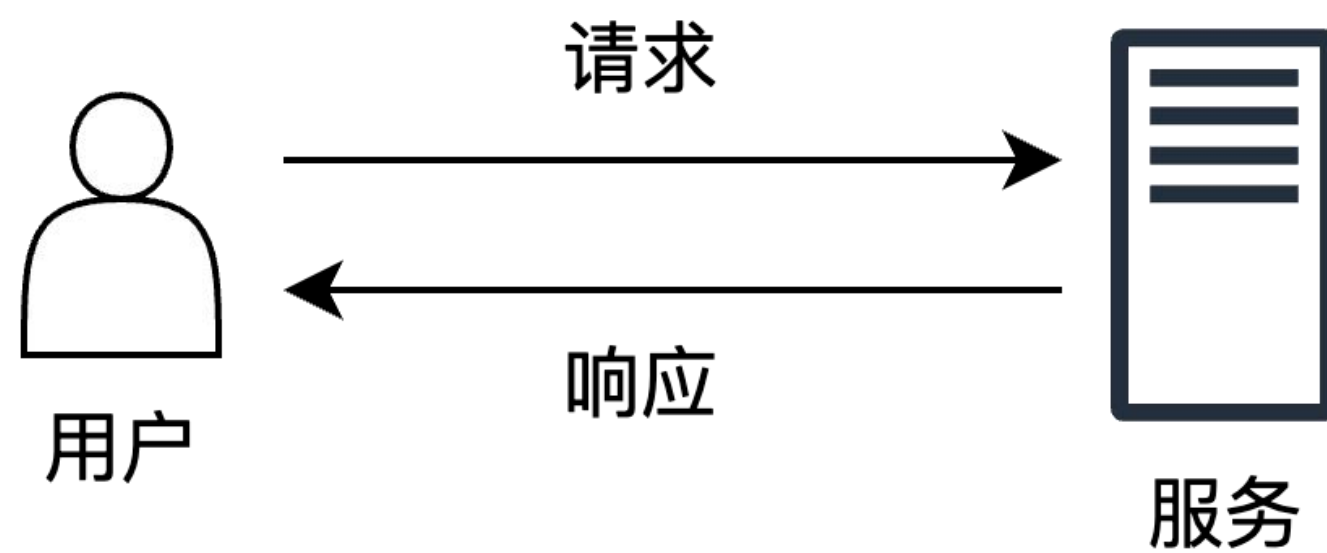
tcprt 简介

背景

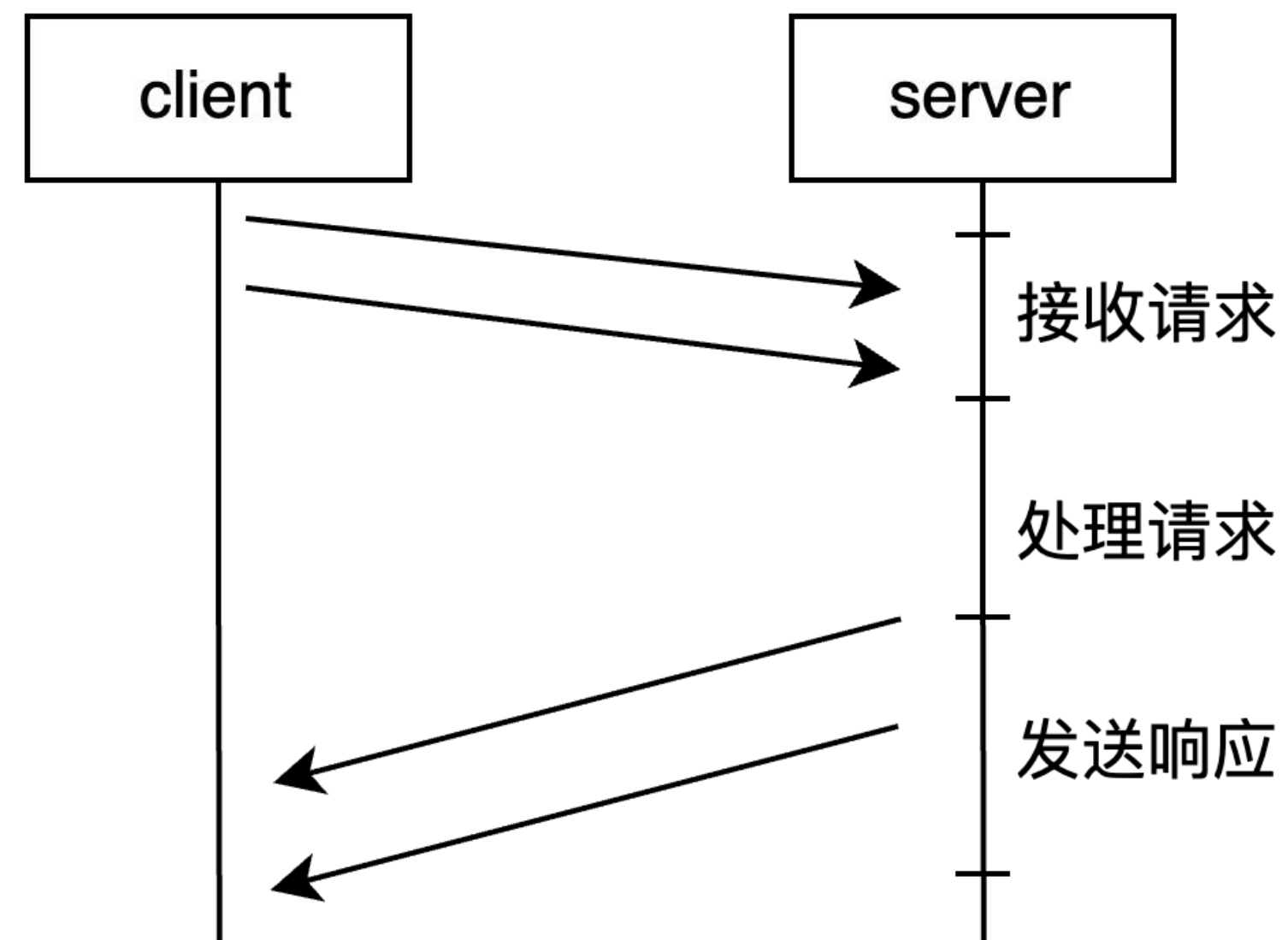
一次请求过程慢了，是谁的锅？

应用 or 网络？

一次请求的处理过程



e.g., HTTP/1.1、MySQL、Redis...



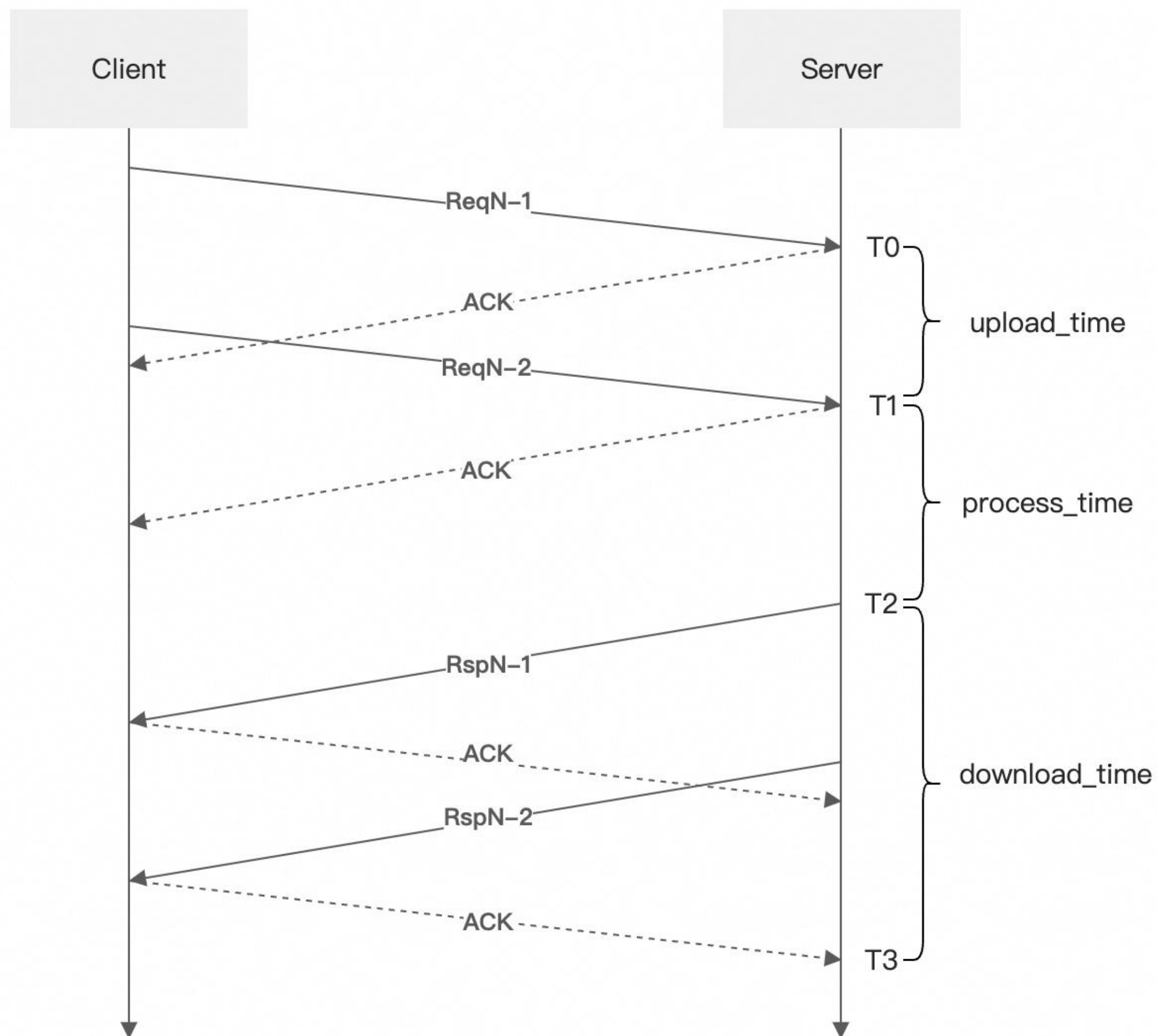
- 网络：接收请求，发送响应
- 应用：处理请求

tcprt 简介

是什么

[请求-响应] 粒度的 TCP 监控

- 一次 [请求 + 响应] 为一个 *TASK*
- 监控每个 *TASK* 的 3 个重要时间段
- *upload time*: 上传请求的耗时
- *process time*: 应用处理请求的耗时
- *download time*: 发送响应的耗时



tcprt 简介



效果举例

V6	R	1726651956	15727	10.	62:41290	10.	07:80	690	3303	2240	0	10	24	0	141	0	1448
V6	R	1726651956	19031	10.	62:41290	10.	07:80	690	3376	2240	0	11	21	0	141	0	1448
V6	R	1726651956	22407	10.	62:41290	10.	07:80	690	3315	2240	0	12	19	0	141	0	1448
V6	R	1726651956	25723	10.	62:41290	10.	07:80	690	3300	2240	0	13	19	0	141	0	1448
V6	R	1726651956	29024	10.	62:41290	10.	07:80	690	3314	2240	0	14	21	0	141	0	1448
V6	R	1726651956	32339	10.	62:41290	10.	07:80	690	3310	2240	0	15	20	0	141	0	1448
V6	R	1726651956	35649	10.	62:41290	10.	07:80	690	3274	2240	0	16	21	0	141	0	1448
V6	R	1726651956	38924	10.	62:41290	10.	07:80	690	3280	2240	0	17	19	0	141	0	1448
V6	R	1726651956	42204	10.	62:41290	10.	07:80	690	3286	2240	0	18	26	0	141	0	1448

日志时间戳

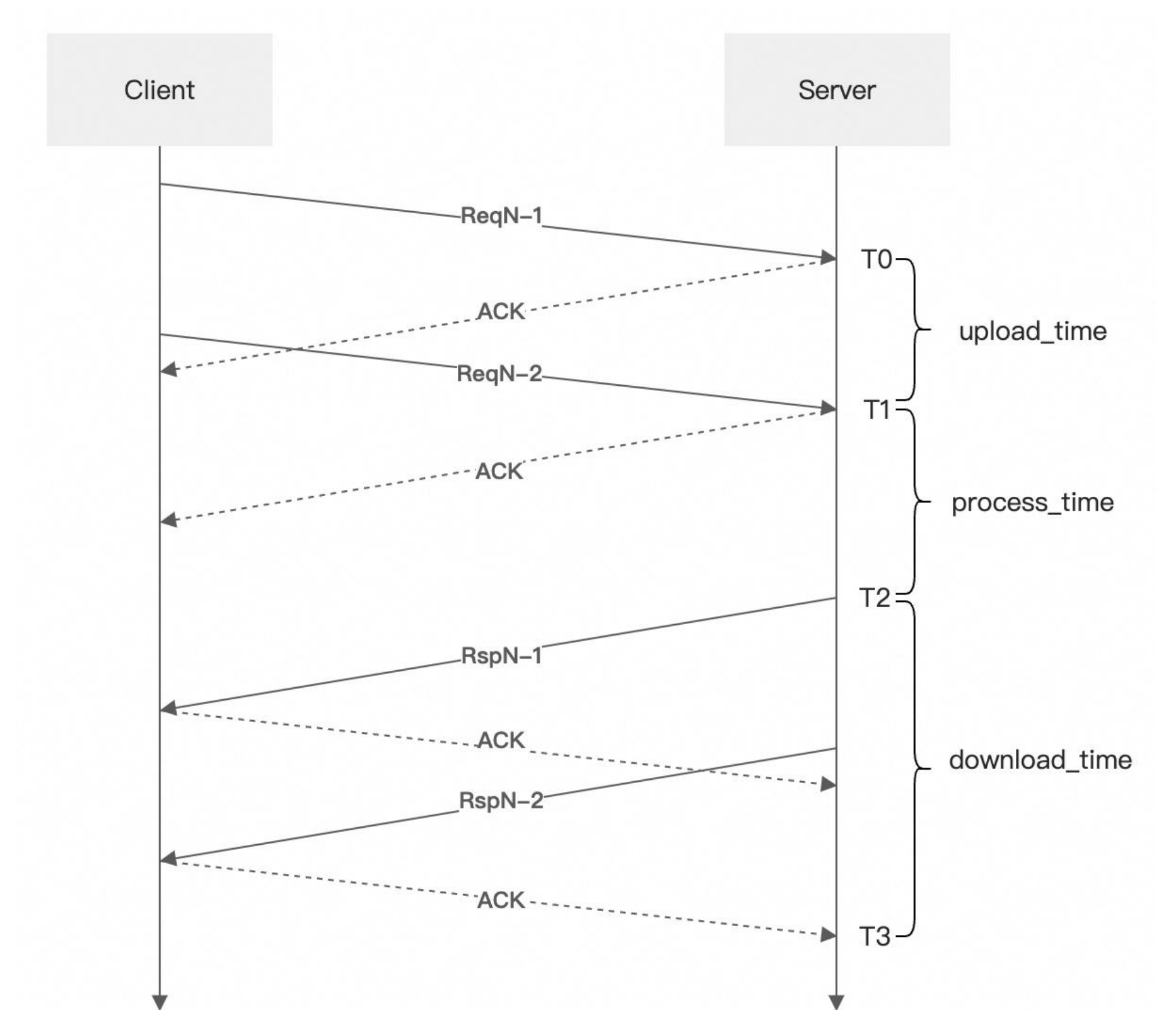
四元组

发送数据量	总耗时	最小RTT	重传数	任务序号	请求处理耗时	请求发送用时	接收数据量	是否乱序	最大段长度
-------	-----	-------	-----	------	--------	--------	-------	------	-------

*仅展示部分常用功能，完整说明可参考文档：<https://help.aliyun.com/zh/alinux/user-guide/tcp-rt-configurations>

tcpert 简介

应用举例：数据库

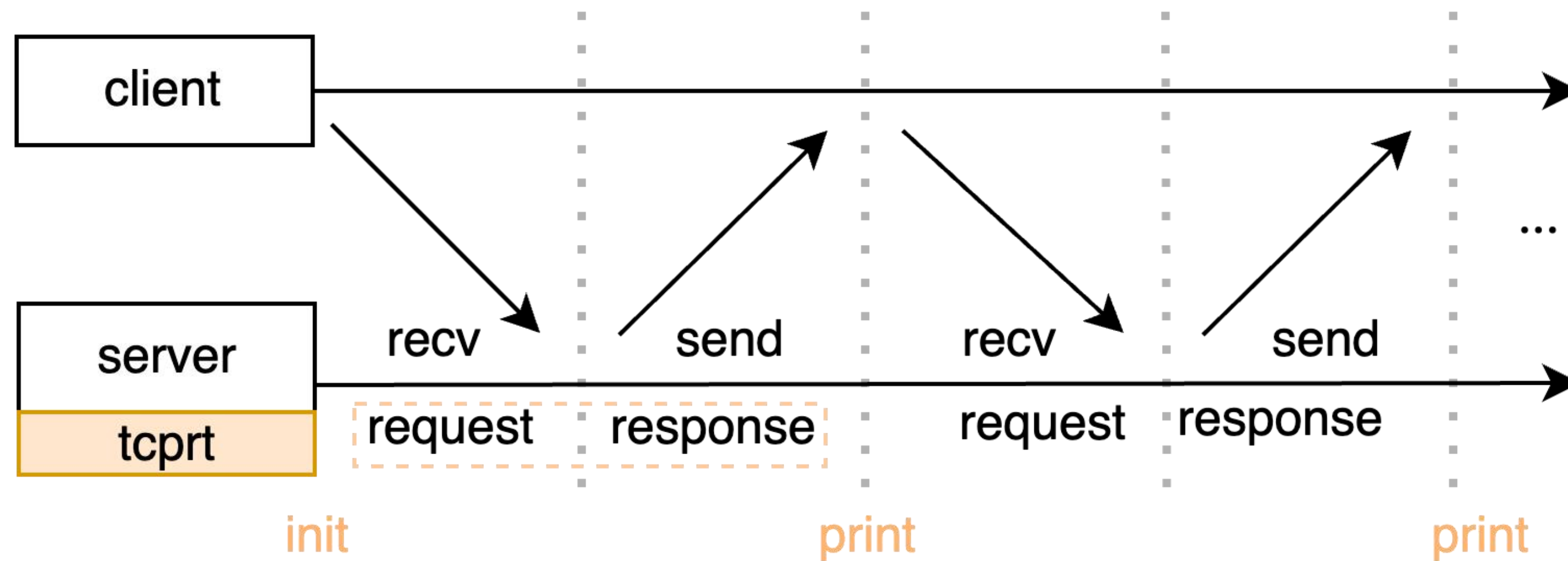


- 蓝线：总耗时， $T0 \sim T3$
- 黑线：请求处理耗时， $T1 \sim T2$

过去: *kernel module* 实现

如何获取各段时间信息?

=> 在数据收发位置添加 *hook*, 即可收集到 *TASK* 信息

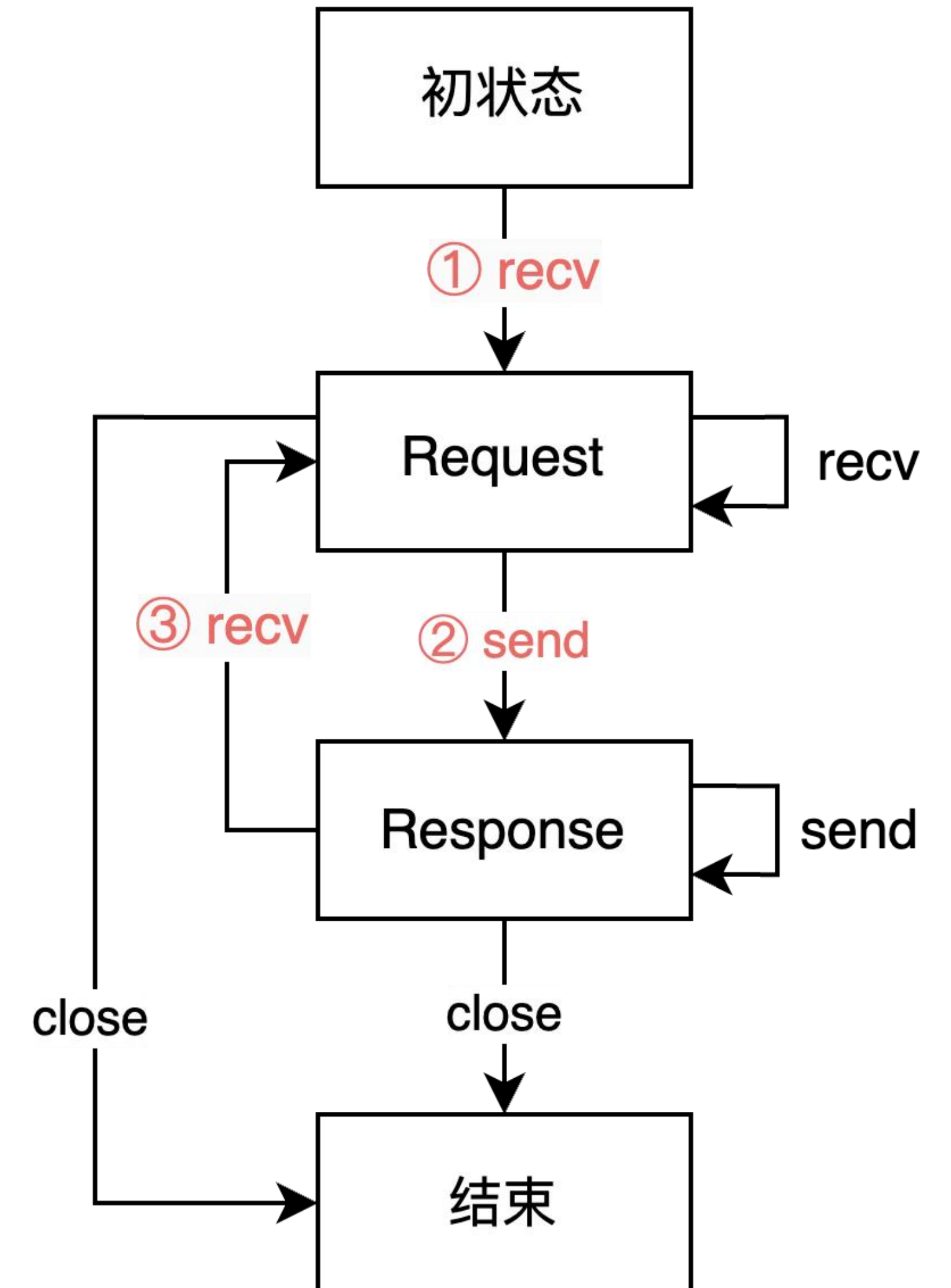


过去: *kernel module* 实现

如何获取各段时间信息?

=> 在数据收发位置添加 *hook*, 即可收集到 *TASK* 信息

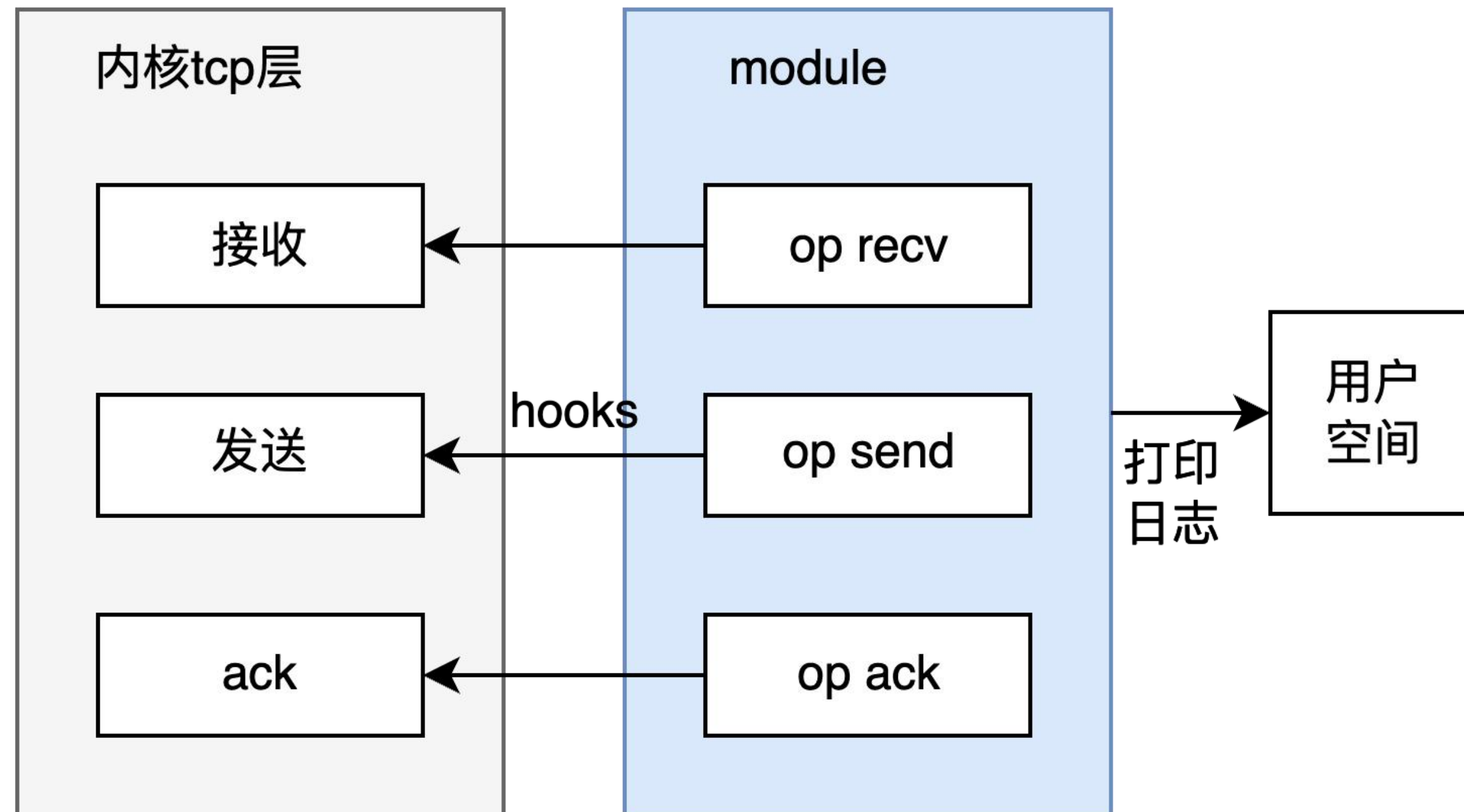
- ① 初始状态下, 收到数据包: 开始接收请求, 初始化一条 *TASK* 记录。
(进入 *REQUEST* 状态)
- ② *REQUEST* 状态下, 发送数据包: 请求处理完成, 开始发送响应。记录相关信息。
(进入 *RESPONSE* 状态)
- ③ *RESPONSE* 状态下, 收到数据包: 响应发送完毕, *TASK* 结束。记录信息并输出一条 *TASK* 日志, 然后初始化一条新的 *TASK* 记录。
(回到 *REQUEST* 状态)



过去: *kernel module* 实现

如何获取各段时间信息?

=> 在数据收发位置添加 *hook*, 即可收集到 *TASK* 信息



过去：kernel module 引入维护成本

作为内核模块实现，为长期、稳定地维护，带来挑战

内核符号缺少稳定性保证

内核符号不提供 *API* 稳定性保证，可能发生变化。

模块维护者通常不会检查所有符号的兼容性。

一旦符号变化（比如函数语义变化），很容易引入 *bug*，排查费时费力。

适配各种内核版本

内核大版本 + 小版本，数量已过百，未来还会持续增加。

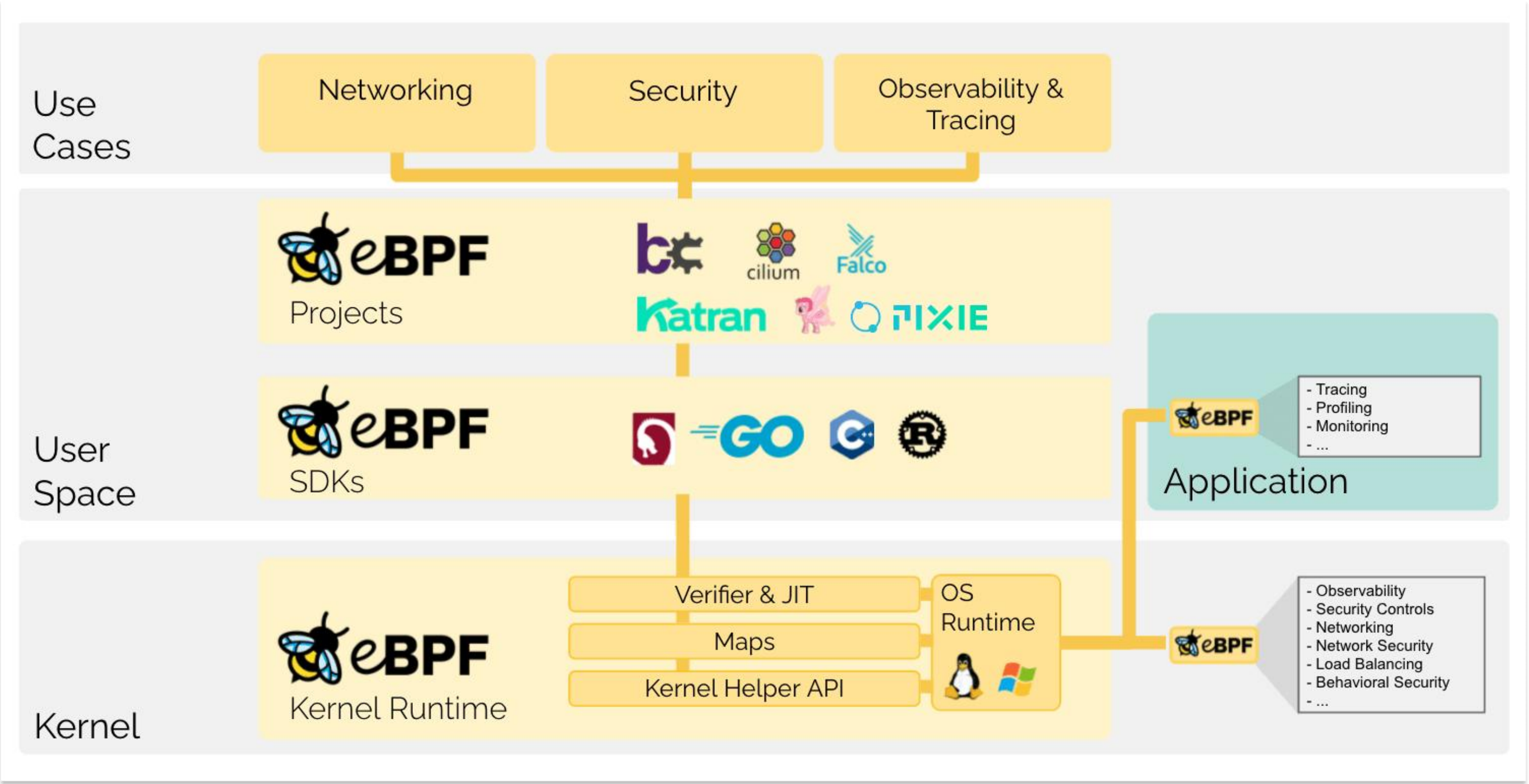
适配各种内核版本，工作量大，风险高。

现在：利用 eBPF 实现 tcprrt

eBPF 简介

动态扩展内核功能，提供较强的安全、稳定保证。

近几年来，发展迅速，功能不断增强。



*图源: <https://ebpf.io/zh-hans/what-is-ebpf/>

现在：利用 *eBPF* 实现 *tcprrt*

eBPF 实现，带来哪些收益？

1. **维护成本低**。动态扩展内核功能，与内核版本解耦。
2. 提升**稳定性**和**安全性**。*eBPF* 通过 *verifier* 的检查，从原理上是保证安全的。
3. 便于**功能扩展**。基于 *eBPF* 可以实现一些难以在内核实现的功能，并且有安全性保证。
4. **使用简单**。*eBPF* 版本作为用户态包，使用方法上更符合用户习惯。
5. 繁荣 ***eBPF* 生态**。从实际应用出发，反哺 *bpf* 社区，形成良性循环。

现在： 利用 *eBPF* 实现 *tcprt*



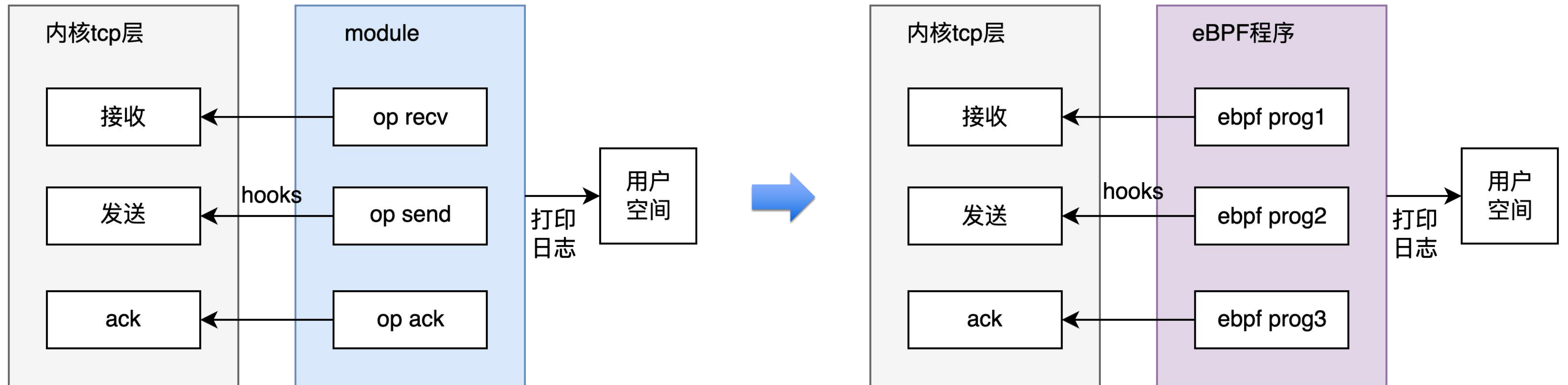
两种实现方式的对比

	内核模块（旧）	eBPF（新）
功能	TCP 基础监控	兼容原版，并扩展部分功能
兼容性	内核符号需看护	bpf helper 保证兼容性
维护方式	随内核版本迭代	与内核解耦
intree 代码量 (lines)	1119	556
使用方式	modprobe	rpm/systemctl

现在：利用 *eBPF* 实现 *tcpprt*

如何改用 *eBPF* 实现？

核心逻辑几乎不变



现在：利用 *eBPF* 实现 *tcprt*

扩展 *tcprt* 功能

- *eBPF* 的高安全性，方便二次开发
- 功能在逻辑上更适合作为 *out-of-tree* 扩展（而非 *intree* 的内核模块）

HTTP/TLS 支持

通过数据包 *payload* 解析，新增部分 *HTTP/TLS* 支持，以便匹配应用层监控信息

- *TLS* 握手阶段识别
- *HTTPS* 挥手 *close_notify Alert* 报文识别

现在：利用 *eBPF* 实现 *tcprt*

完善 *eBPF* 子系统

tracepoint nullable argument

before: *bpf* 默认 *tracepoint* 的参数是非空的。实际上却可为 *NULL*，可导致内核 *panic*

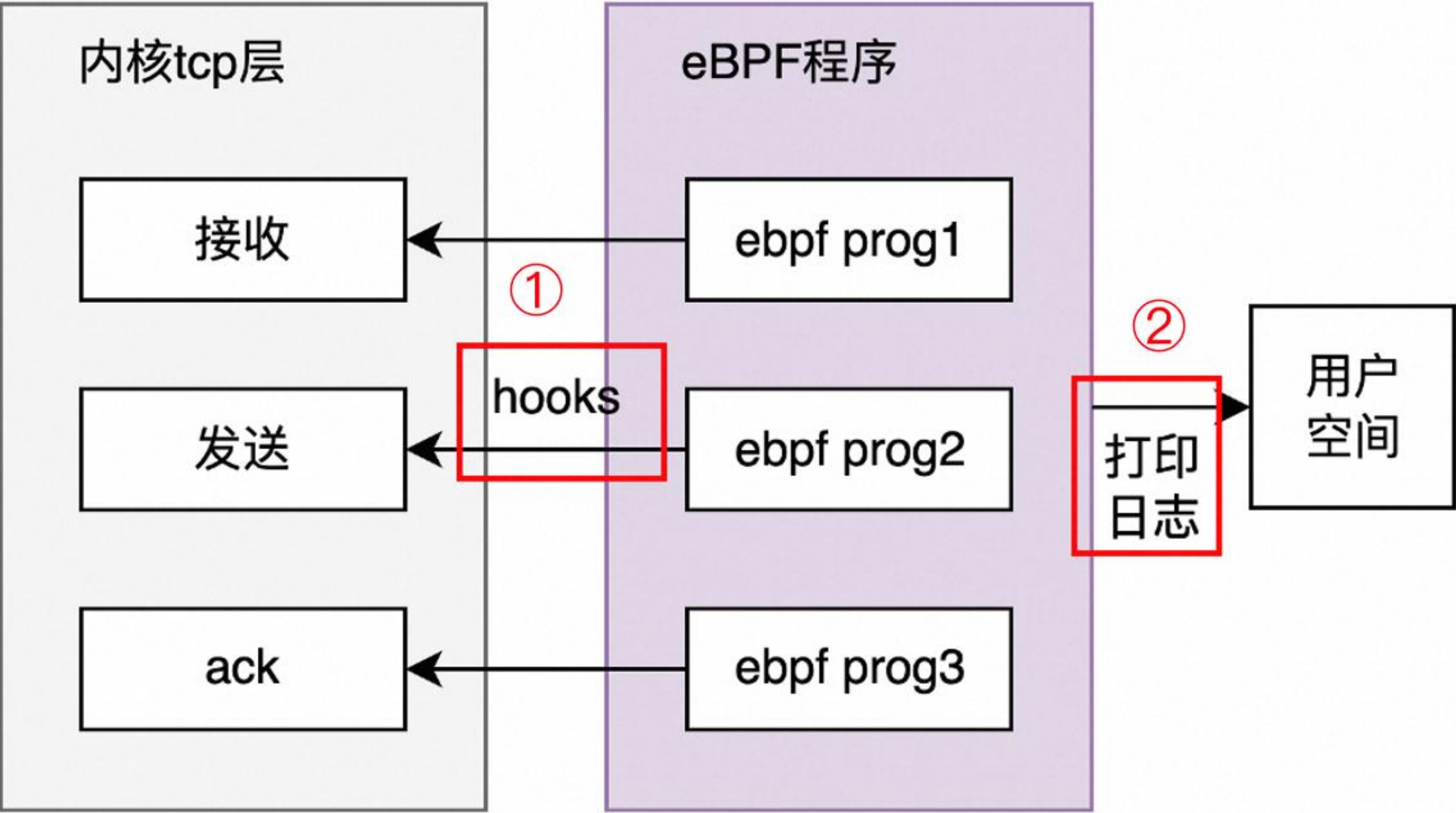
```
→ /* skb of trace_tcp_send_reset() keeps the skb that caused RST,
→ * skb here is different to the troublesome skb, so use NULL
→ */
→ trace_tcp_send_reset(sk, NULL, reason);
→ }
```

after: *tracepoint* 参数添加“*_nullable*”后缀，*bpf* 可以通过 *BTF* 识别出可能为 *NULL* 的参数

```
TRACE_EVENT(tcp_send_reset,
    TP_PROTO(const struct sock *sk,
-           const struct sk_buff *skb,
+           const struct sk_buff *skb_nullable,
            const enum sk_rst_reason reason),
-   TP_ARGS(sk, skb, reason),
+   TP_ARGS(sk, skb_nullable, reason),
    TP_STRUCT__entry(
        __field(const void *, skbaddr)
```


现在：利用 *eBPF* 实现 *tcp*rt

尚未解决的难点：主要在内核接口



现在：利用 *eBPF* 实现 *tcprt*

内核适配

难点1: *hook* 实现

内核模块采用静态 *hook* 点，当前版本采用私有 *tracepoint* 代替

```
diff --git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
index 6a8d53d6540b..7586349214a7 100644
--- a/net/ipv4/tcp_input.c
+++ b/net/ipv4/tcp_input.c
@@ -760,6 +760,7 @@ static void tcp_event_data_recv(struct sock *sk, struct sk_buff *skb)
     now = tcp_jiffies32;

     tcp_rt_call(sk, recv_data);
+    trace_tcp_pkt_recv(sk, skb);

     if (!icsk->icsk_ack.ato) {
         /* The _first_ data packet received, initialize
@@ -3352,6 +3353,7 @@ static int tcp_clean_rtx_queue(struct sock *sk, u32 prior_fack,
     }

     tcp_rt_call(sk, pkts_acked);
+    trace_tcp_data_acked(sk);

     if (icsk->icsk_ca_ops->pkts_acked) {
         struct ack_sample sample = { .pkts_acked = pkts_acked,
diff --git a/net/ipv4/tcp_output.c b/net/ipv4/tcp_output.c
index 9dfd1642b1e4..7945ebef45a3 100644
--- a/net/ipv4/tcp_output.c
+++ b/net/ipv4/tcp_output.c
@@ -2719,6 +2719,7 @@ static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
     tcp_schedule_loss_probe(sk, false);

     tcp_rt_call(sk, send_data);
+    trace_tcp_data_send(sk);
     return false;
 }

 return !tp->packets_out && !tcp_write_queue_empty(sk);
```

现在：利用 eBPF 实现 tcprt

内核适配

难点2：日志输出

内核模块利用 *relayfs* 实现，*eBPF* 缺少支持 *overwrite* + *pin* 的高效 *map*

bpf ringbuf

设计理念是 *cpu* 共享 + 保序
不支持 *overwrite*

bpf perfbuf

依赖用户进程的内存空间
不支持 *pin*

社区方案讨论

讨论 *ftrace ring buffer*
强同步，性能不佳
可尝试实现 *relay map*

relay map 推社区

per-cpu 定位与 *perfbuf* 冲突
建议改进 *perfbuf*

perfbuf + 守护进程

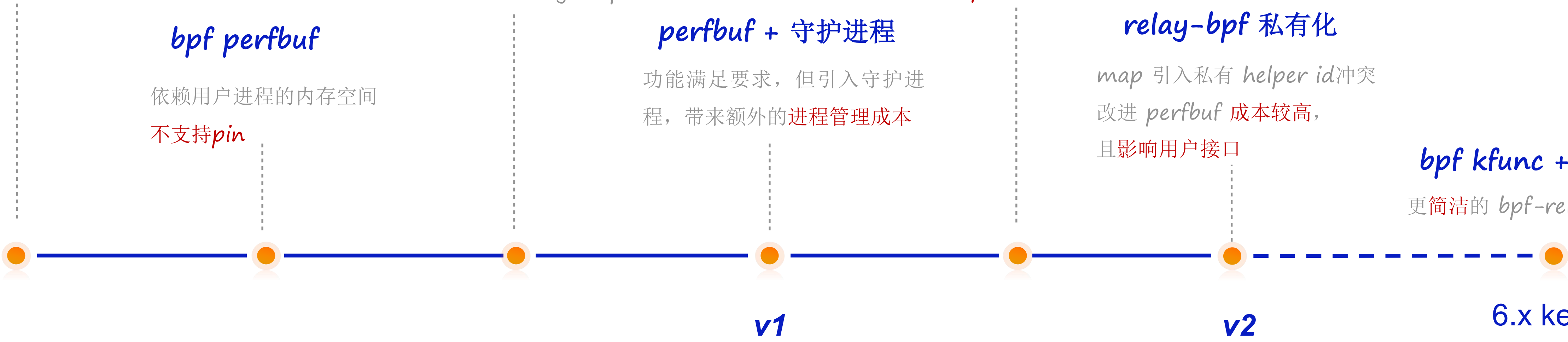
功能满足要求，但引入守护进程，带来额外的进程管理成本

relay-bpf 私有化

map 引入私有 *helper id* 冲突
改进 *perfbuf* 成本较高，且影响用户接口

bpf kfunc + *kptr*

更简洁的 *bpf-relay* 接口



总结

tcprt: 面向 请求-响应 通信模式的 *TCP* 监控



目前 *eBPF* 版本的 *tcprt* 还处在早期阶段，仍有缺憾，有待完善

tcprt: 利用内核 *hook*，采集请求处理过程中，各阶段耗时

内核模块实现

灵活自由，但有代价

eBPF 实现

安全稳定易维护，方便二次开发
带来新的挑战

未来展望


tcprt 功能扩展

- 日志解析工具
- 容器场景支持
- IPv6支持
- ...

实现方式优化

随着 *eBPF* 功能越来越强大，实现方式有了更多/更好的选择

- 用户交互/日志输出: *bpf_arena*
- 内核功能调用: *kfunc*
- *hook*点: *fentry*、*struct_ops*

 阿里云 | 计算,为了无法计算的价值