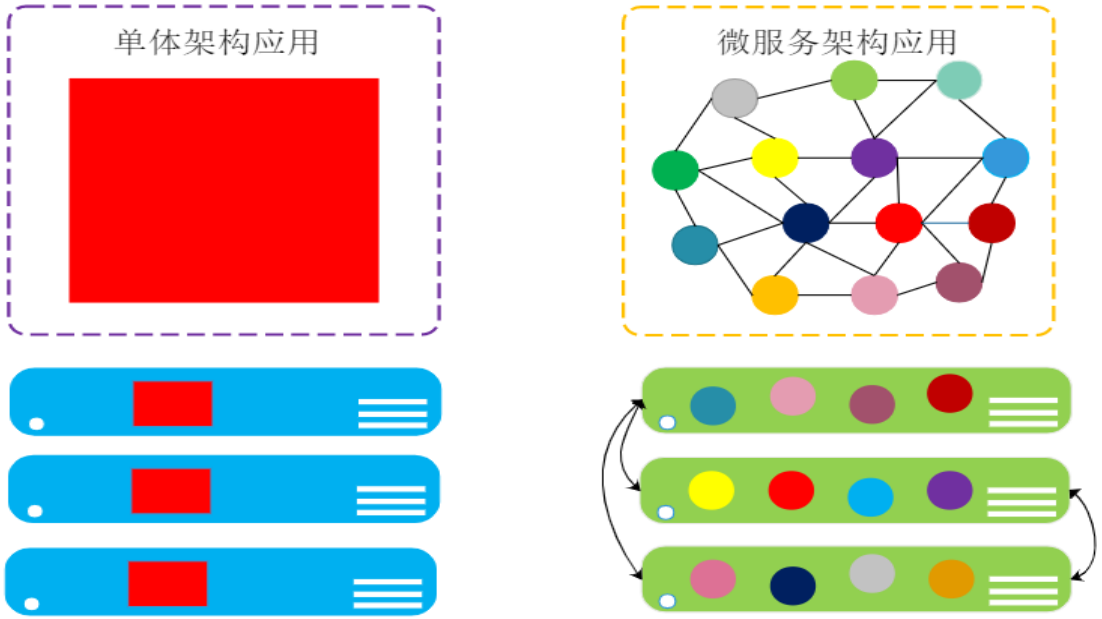
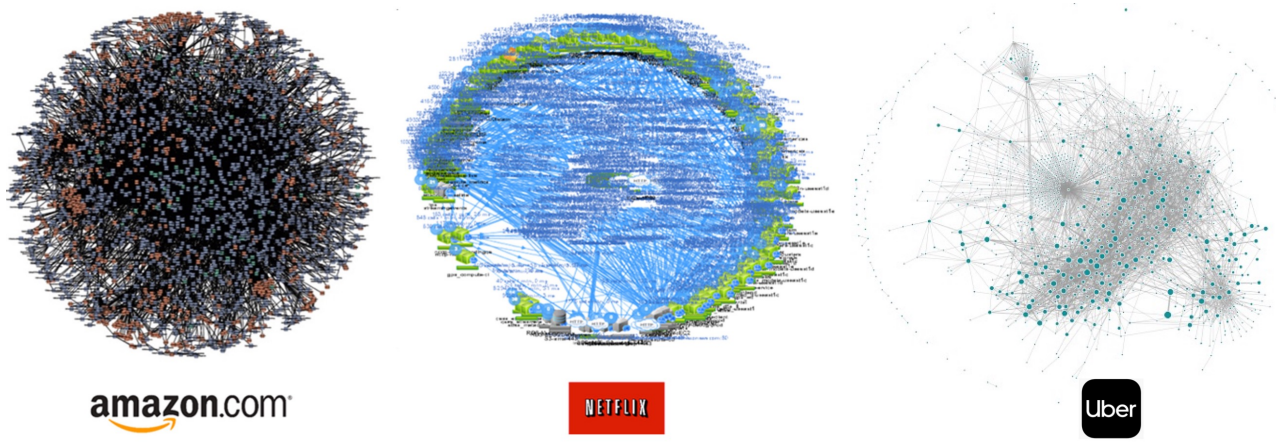
软硬协同加速云原生网络的技术实践

中山大学
演讲人：陈泓仰
导师：陈鹏飞

背景&动机



现代云原生软件系统的规模呈现指数级增加，动辄上千个微服务，例如：WeChat包含近3000个微服务， Netflix超过 700种微服务， Uber包含的 微服务多达2200个， ……；



云原生技术流行

- 单体架构应用 => 微服务架构应用（多个功能不一的服务）
- 容器化部署
- 可独立部署、开发效率高
- 可扩展性强

系统复杂度高

- 服务、服务实例数量庞大
- 服务调用关系复杂，调用链长

保证云原生应用内的网络通信很重要

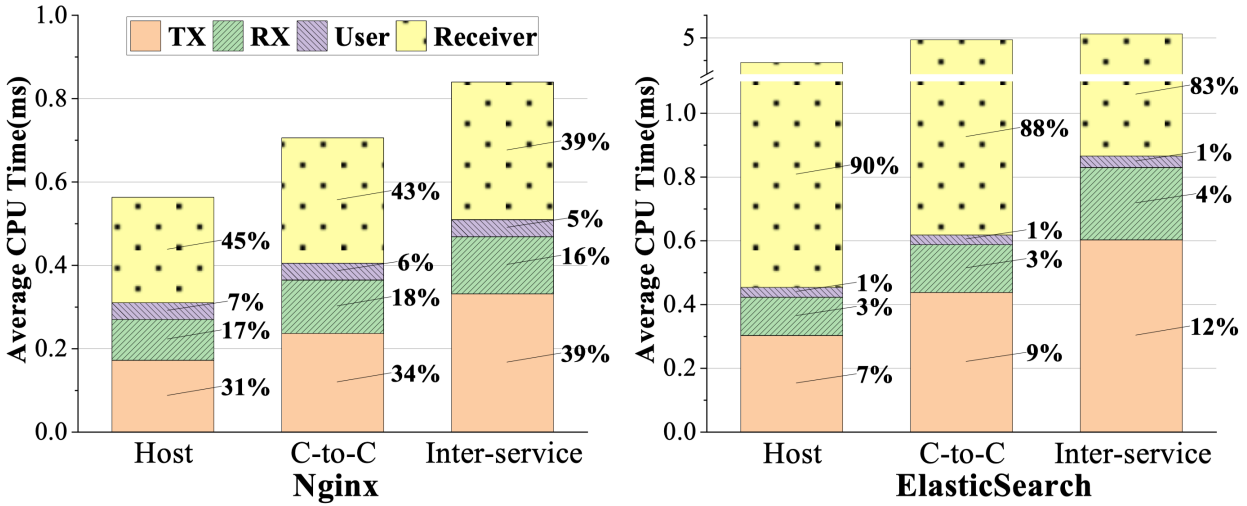
背景&动机

现有的云原生网络的通信模式

- 通过 Overlay 网络提供**容器间通信能力**
  - 基于如VxLAN等的隧道协议进行跨主机的容器通信
- 依赖 KubeProxy + IPTables 提供**服务间通信能力**
  - Linux Netfliter 机制触发

服务间通信存在开销

- Overlay网络的使用，**导致发送端需要进行更多的CPU处理**来完成数据包的发送（TX）和接收（RX）
- 相较于容器间通信，**在服务间通信中，发送端需要在发送阶段（TX）消耗更多的CPU时间**



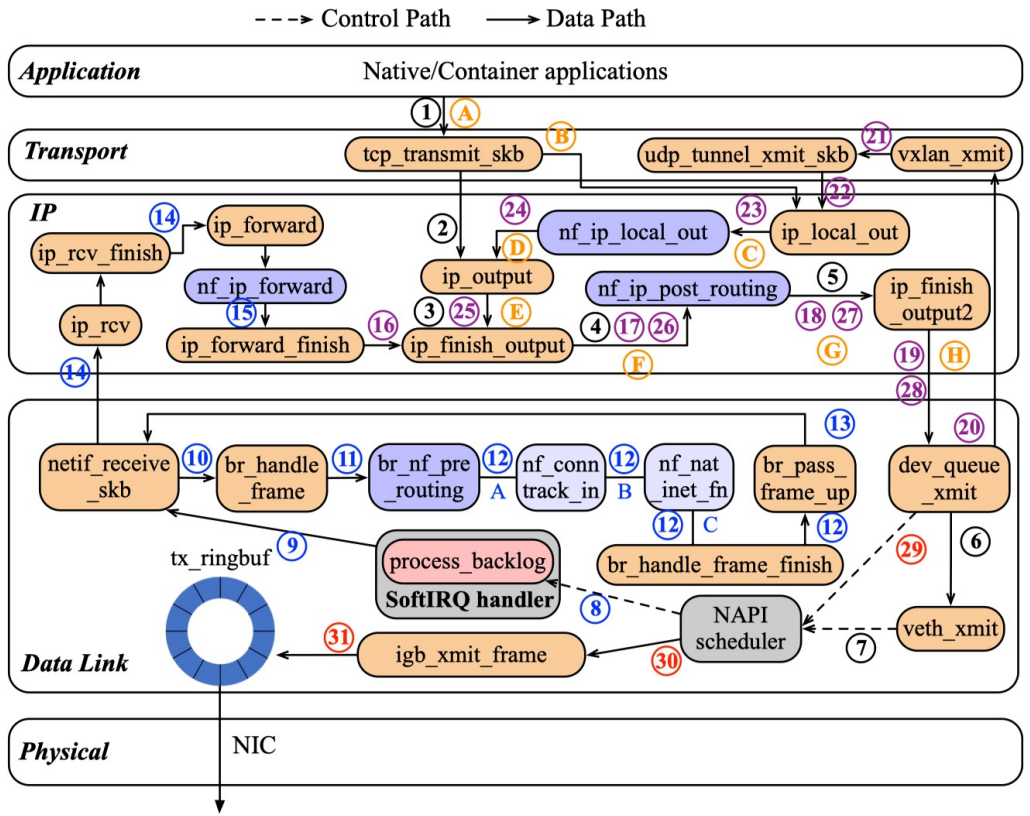
三种不同网络模式（主机网络、容器间网络、服务间网络）下，Nginx和ElasticSearch应用单次请求消耗的CPU时间

如何优化Kubernetes网络中，
发送端进行服务间通信的性能?

背景&动机

性能开销来源

1. 数据包在内核中的传输路径**冗长**



主机网络模式下，发送数据包在内核中的传输路径（字母标记）；
服务间通信模式下，发送数据包在内核中的传输路径（数字标记）

TCP/IP 相关的内核函数调用次数多，且耗时长

Component	Major functions	Time (us)	Count	Overall time (us)
Transport / IP	ip_output*	50.13	3	**243.17(28.51%)**
	ip_finish_output*	28.53	3	
	ip_finish_output2*	40.86	3	
	ip_forward*	28.85	1	
	ip_forward_finish*	33.64	1	
	ip_local_out	15.66	1	
	ip_rcv*	18.35	1	
	ip_rcv_finish*	27.15	1	
	Netfilter: nf_conntrack_in	17.39	2	**133.48(15.65%)**
	Netfilter: nf_nat_ipv4_fn	27.66	2	
	Netfilter: iptables rules match	88.43	62	
	vxlan_xmit*	78.33	1	78.33(9.18%)
Bridge	br_handle_frame*	20.72	1	**163.06(19.12%)**
	br_nf_pre_routing*	55.21	1	
	br_nf_pre_routing_finish*	34.17	1	
	br_handle_frame_finish*	34.61	1	
	br_pass_frame_up*	18.36	1	
Device	veth_xmit	9.68	1	234.80(27.53%)
	__dev_queue_xmit*	187.29	1	
	__netif_receive_skb*	37.74	1	

服务间通信模式下，发送一次数据包时，各个内核函数被调用的次数和总耗时

背景&动机

性能开销来源

1. 数据包在内核中的传输路径**冗长**
2. 网桥处理和隧道协议（VxLAN）封包处理**耗时**

Component	Major functions	Time (us)	Conut	Overall time (us)
Transport / IP	ip_output*	50.13	3	**243.17(28.51%)**
	ip_finish_output*	28.53	3	
	ip_finish_output2*	40.86	3	
	ip_forward*	28.85	1	
	ip_forward_finish*	33.64	1	
	ip_local_out	15.66	1	
	ip_rcv*	18.35	1	
	ip_rcv_finish*	27.15	1	
	Netfilter: nf_conntrack_in	17.39	2	**133.48(15.65%)**
	Netfilter: nf_nat_ipv4_fn	27.66	2	
	Netfilter: iptables rules match	88.43	62	
	vxlan_xmit*	78.33	1	78.33(9.18%)
Bridge	br_handle_frame*	20.72	1	**163.06(19.12%)**
	br_nf_pre_routing*	55.21	1	
	br_nf_pre_routing_finish*	34.17	1	
	br_handle_frame_finish*	34.61	1	
	br_pass_frame_up*	18.36	1	
Device	veth_xmit	9.68	1	234.80(27.53%)
	__dev_queue_xmit*	187.29	1	
	__netif_receive_skb*	37.74	1	

服务间通信模式下，发送一次数据包时，各个内核函数被调用的次数和总耗时

背景&动机

性能开销来源

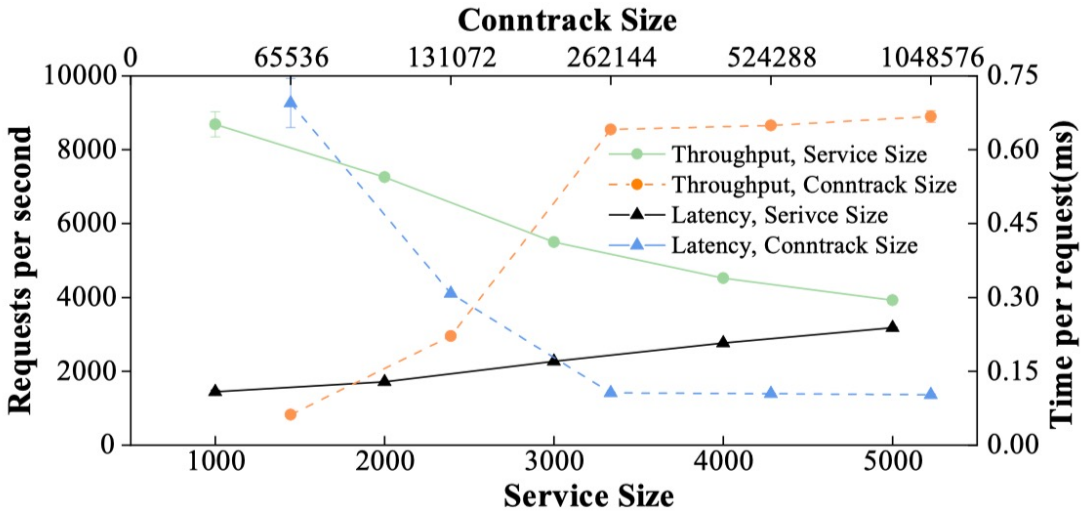
1. 数据包在内核中的传输路径**冗长**
2. 网桥处理和隧道协议（VxLAN）封包处理**耗时**
3. Linux Netfliter 机制**低效**（IPTables, ConnTrack）

IPTables规则匹配函数被频繁调用，相关的内核函数耗时长

Component	Major functions	Time (us)	Conut	Overall time (us)
Transport / IP	ip_output*	50.13	3	243.17(28.51%)
	ip_finish_output*	28.53	3	
	ip_finish_output2*	40.86	3	
	ip_forward*	28.85	1	
	ip_forward_finish*	33.64	1	
	ip_local_out	15.66	1	
	ip_rcv*	18.35	1	
	ip_rcv_finish*	27.15	1	
	Netfilter: nf_conntrack_in	17.39	2	133.48(15.65%)
	Netfilter: nf_nat_ipv4_fn	27.66	2	
	Netfilter: iptables rules match	88.43	62	
	vxlan_xmit*	78.33	1	78.33(9.18%)
Bridge	br_handle_frame*	20.72	1	163.06(19.12%)
	br_nf_pre_routing*	55.21	1	
	br_nf_pre_routing_finish*	34.17	1	
	br_handle_frame_finish*	34.61	1	
	br_pass_frame_up*	18.36	1	
Device	veth_xmit	9.68	1	234.80(27.53%)
	__dev_queue_xmit*	187.29	1	
	__netif_receive_skb*	37.74	1	

服务间通信模式下，发送一次数据包时，各个内核函数被调用的次数和总耗时

IPTables规则 (服务) 越多，延迟越大，吞吐量越低
ConnTrack表中表项越多，延迟越大，吞吐量越低



服务间通信模式下，随着服务数量、ConnTrack表大小的增大，请求延迟和吞吐量大小的变化

ConnTrack表中表项越多，内存占用越大

Conntrack table size	65536	131072	262144	52428	1048576
Memory Usage (MB)	20.63	41.25	82.5	165	330

不同大小的ConnTrack表的内存使用量

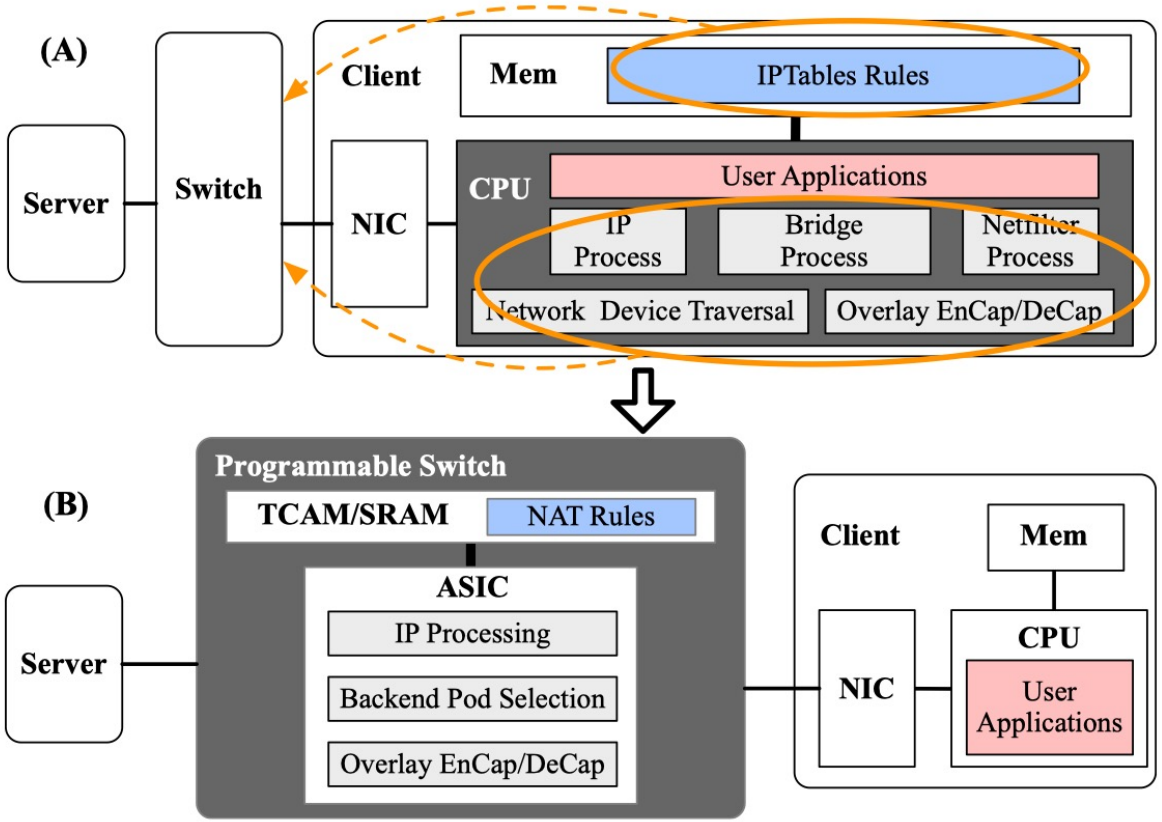
方法设计

设计目标：

- 避免上述性能开销，提高现有网络的性能，提升云原生应用的性能
- 降低云原生网络对主机的资源开销

设计核心：

- 主机侧尽可能多的绕过内核网络栈
- 将更多的网络处理逻辑下沉到可编程交换机中
  - 可编程交换机：**快速访存**、**快速数据包处理**
  - 在网负载均衡、连接追踪



(A) 原始网络模式下，主机CPU进行所有的网络处理；
(B) 我们的方法，将必要的网络处理下沉到可编程交换机

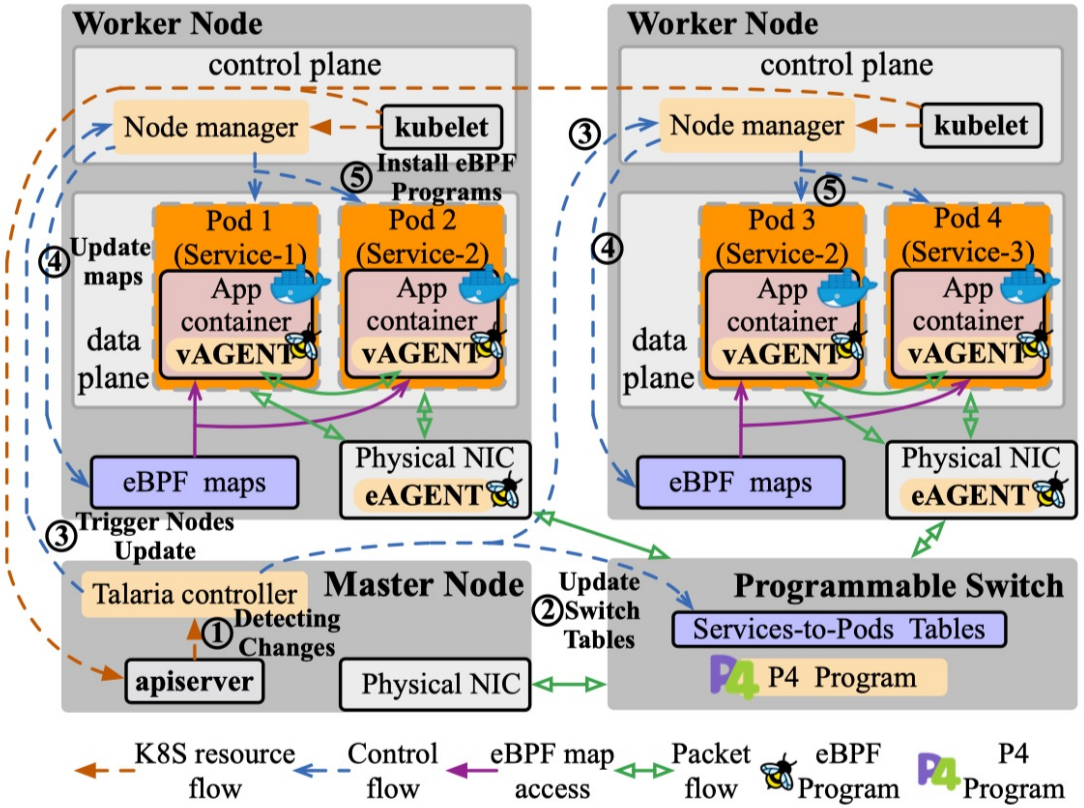
方法设计

设计框架

1. 控制面：进行服务、服务实例资源变化的监控，随时更新交换机中的负载均衡规则表

2. 主机：使用eBPF无侵入的绕过内核网络栈
   - 挂载eBPF TC类型的程序到主机网卡和容器网卡，劫持容器流量

3. 交换机：负载均衡&&连接追踪，保证连接一致性
   - 负载均衡：为服务选择合适的后端实例
   - 连接追踪：保证属于同一条连接的数据包始终发送到同一个后端实例
   - **挑战：交换机存储资源稀缺**
   - **解决方法：将连接信息进行编码，插入到数据包中**



实验验证

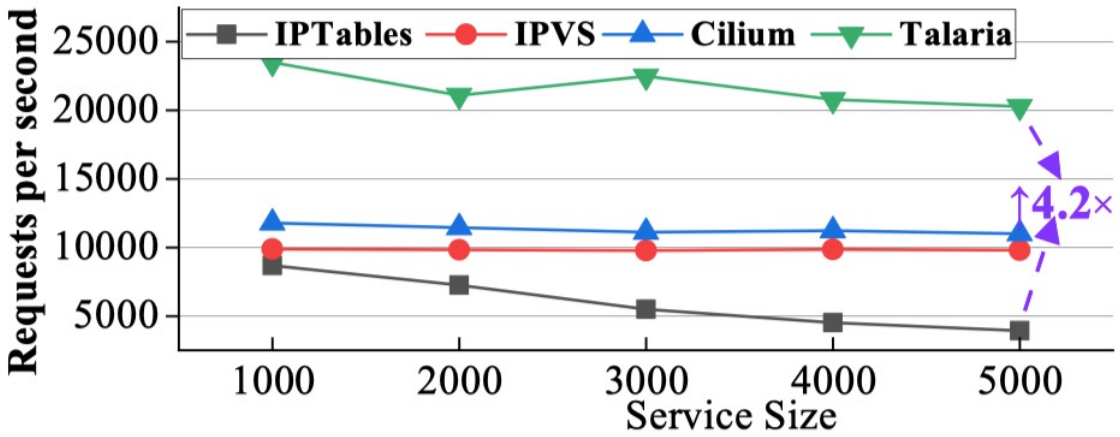
研究问题

1. 性能是否得到提高?
2. 集群节点资源消耗是否减少?
3. 后端实例数量的增多时，已经建立的连接的连接一致性是否会被破坏?

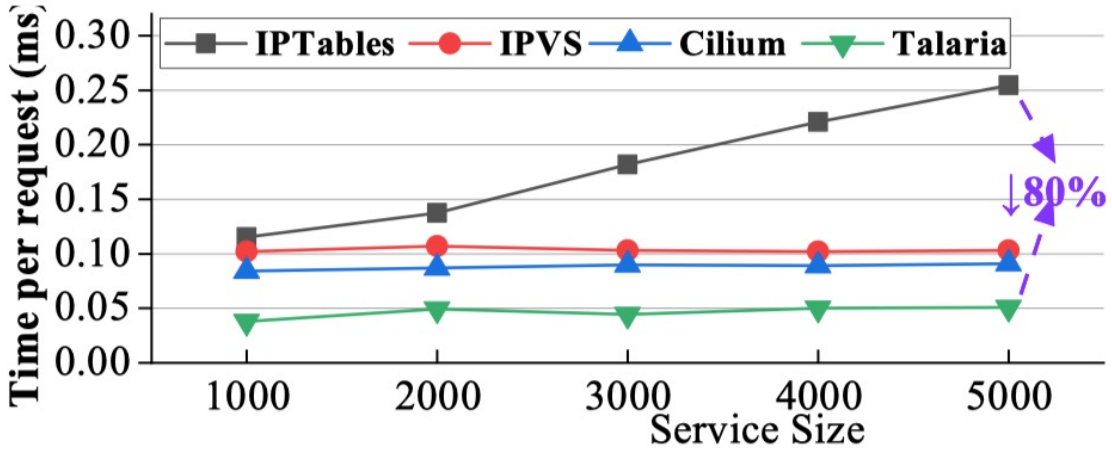
实验验证

研究问题1：性能是否提高?

吞吐量最多提高4.2倍，请求延迟最大减少80%



随着服务数量增多，各个方法吞吐量的变化



随着服务数量增多，各个方法请求延迟的变化

实验验证

研究问题1：性能是否提高?

	Average	Median	90%ile	98%ile	99%ile
IPTables	0.40	0.18	1.30	3.30	3.50
IPVS	0.34 ↓ **0.06 (15%)**	0.15	1.20	2.00	2.10 ↓ **1.40 (40%)**
Cilium	0.12 ↓ **0.28 (70%)**	0.02	0.12	2.20	2.60 ↓ **0.90 (25%)**
Talaria	0.13 ↓ **0.27 (68%)**	0.02	0.12	2.13	2.40 ↓ **1.10 (31%)**

10并发度

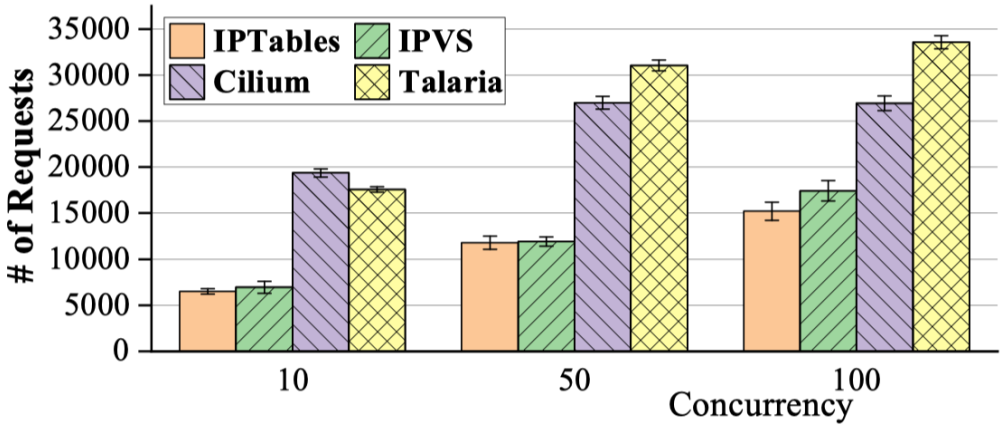
	Average	Median	90%ile	98%ile	99%ile
IPTables	0.99	0.21	1.45	15.0	22.0
IPVS	0.98 ↓ **0.01 (1%)**	0.20	1.37	14.0	17.0 ↓ **5.0 (23%)**
Cilium	0.43 ↓ **0.56 (57%)**	0.07	0.21	7.80	14.3 ↓ **7.7 (35%)**
Talaria	0.38 ↓ **0.61 (62%)**	0.02	0.16	6.18	14.0 ↓ **8.0 (36%)**

50并发度

	Average	Median	90%ile	98%ile	99%ile
IPTables	1.30	0.22	0.11	25.0	33.0
IPVS	1.26 ↓ **0.04 (3%)**	0.23	0.11	24.0	29.5 ↓ **3.5 (12%)**
Cilium	0.85 ↓ **0.45 (35%)**	0.17	0.05	18.0	24.0 ↓ **9.0 (27%)**
Talaria	0.67 ↓ **0.63 (48%)**	0.11	0.03	7.23	23.0 ↓ **10.0 (30%)**

100并发度

不同并发度下，使用不同网络模式的应用的请求延迟



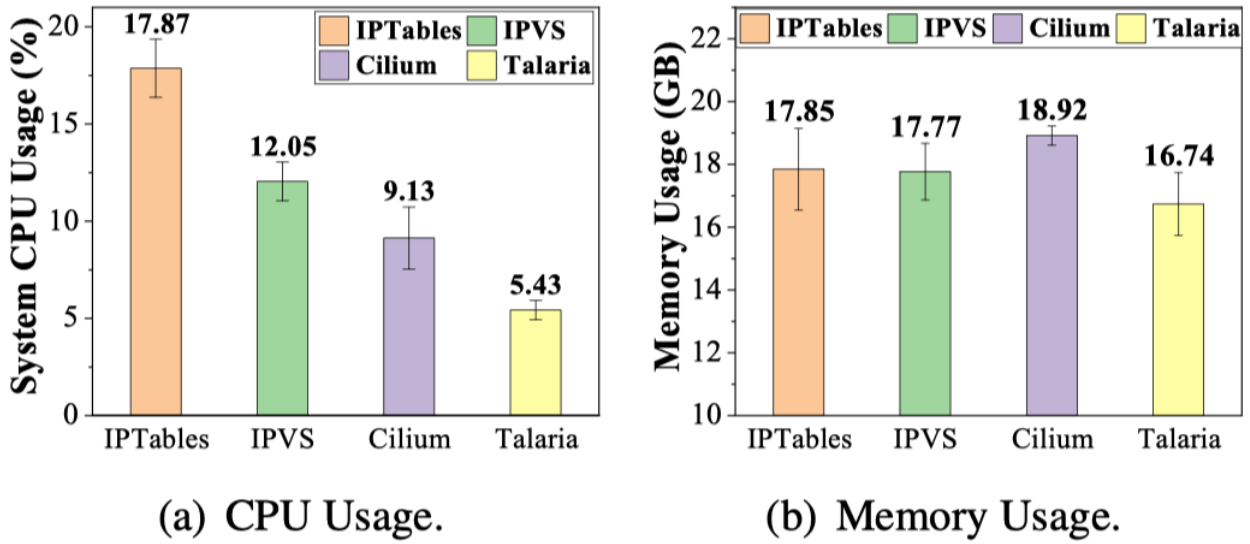
不同并发度下，使用不同网络模式的应用的请求吞吐量

Cilium和Talaria同样都可以提高应用性能（提高吞吐、降低延迟），但是随着并发度的提高，Talaria的性能提升幅度相比Cilium更高。

实验验证

研究问题2：集群节点资源消耗是否减少？

CPU消耗最大减少69%，内存消耗差距不大



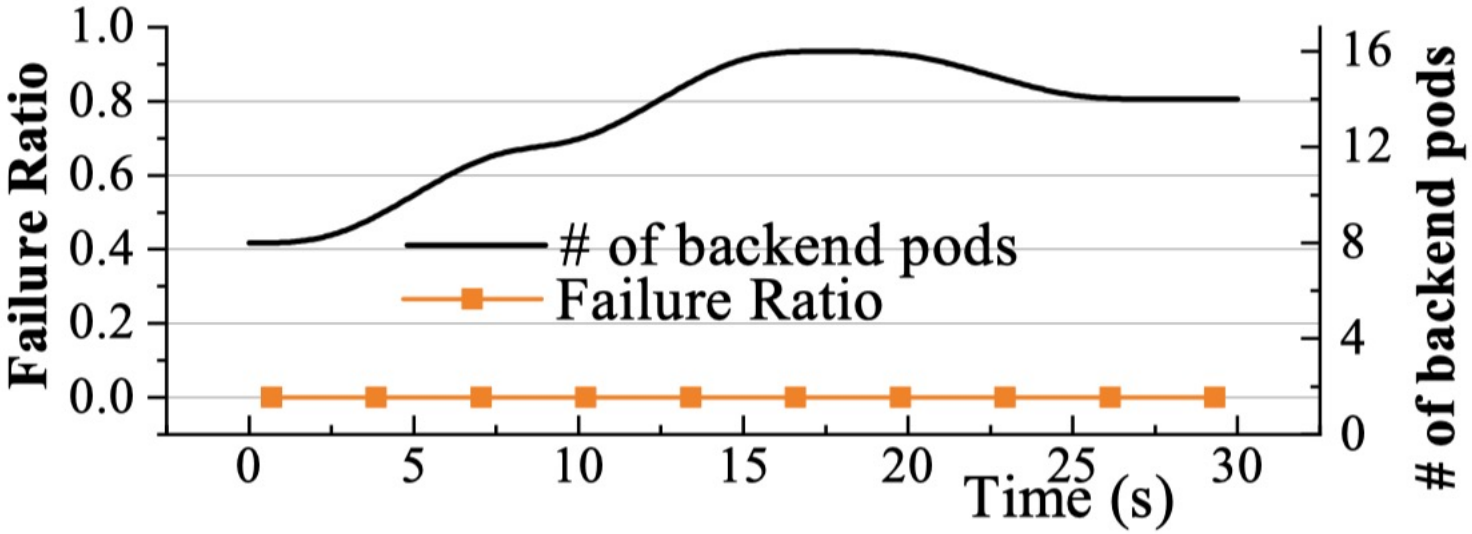
(a) CPU Usage.　　(b) Memory Usage.

使用不同方法后，集群资源（CPU、内存）消耗总量

实验验证

研究问题3：连接一致性保证?

后端实例数量变化，不会影响已有的连接状态



随着后端实例数量变化，已建立的连接的失败率

结论

- 现有的云原生网络模式中存在开销，影响应用性能，消耗集群主机资源
- Talaria提出了一种软硬结合的网络加速方法，尽可能多的避免冗余的内核处理，并将合适的处理放置在可编程交换机。
- 实验表明，Talaria提高了吞吐量平均达2.89倍，减少延迟平均达72.8%，保证了连接一致性，并减少了对集群主机的CPU资源消耗。

THANKS