Kmon：带有 eBPF 的微服务系统的内核内透明监控系统

I. 简介

受高灵活性和快速交付特点的吸引，微服务目前在许多现代公司中被广泛使用，如谷歌和微软。为了监控和管理微服务系统，系统运营商需要收集反映系统状态的性能指标，并将这些指标持久地存储在数据库中。

一些传统的监控工具（如 cAdvisor1）可以帮助系统运营商收集和汇总指标。然而，它们中的大多数只关注资源使用的基本指标（例如，CPU 利用率）。另一种监测工具，如 Istio2，可以监测更多的指标，如 L7 网络层的请求延迟，但他们需要改变底层基础设施。为了在不修改现有系统的情况下获得多样化的指标，我们提出了 Kmon，一个用于微服务系统的内核内传输监测系统。

Kmon 可以更准确地捕获常规指标。此外，它可以收集细粒度的内核内性能指标（例如，系统调用的数量）。内核内指标很有用，因为它们可以反映出深层次的隐藏问题（例如，第二节中的 livelock 或 limplock[14]）。目前的监控工具几乎不可能发现这些深层次的问题，因为它们无法获得内核内事件。因此，收集内核内指标对于微服务的可观察性非常重要。

虽然以前的工具如 strace3 可以捕捉内核内的指标，但它们牺牲了系统的性能，而且对于分布式系统来说，需要手动汇总。其他方法（如 OpenTelemetry4）需要改变用户的程序源代码，这就引入了额外的复杂性。Kmon 选择使用 eBPF，这是 Linux 内核的一个组件，用于收集没有仪器的程序的度量。它降低了使用 eBPF 的难度，使 eBPF 能够感知应用层的变化，特别是对于微服务的变化。用户可以通过编写配置简单地使用它，并将 Kmon 部署到每个主机。然后，Kmon 可以自动收集指标并将其存储到数据库中。该配置指定了用户希望获得的指标和需要监测的微服务中的服务器实例，这不需要修改源代码或提供程序的内部逻辑信息。

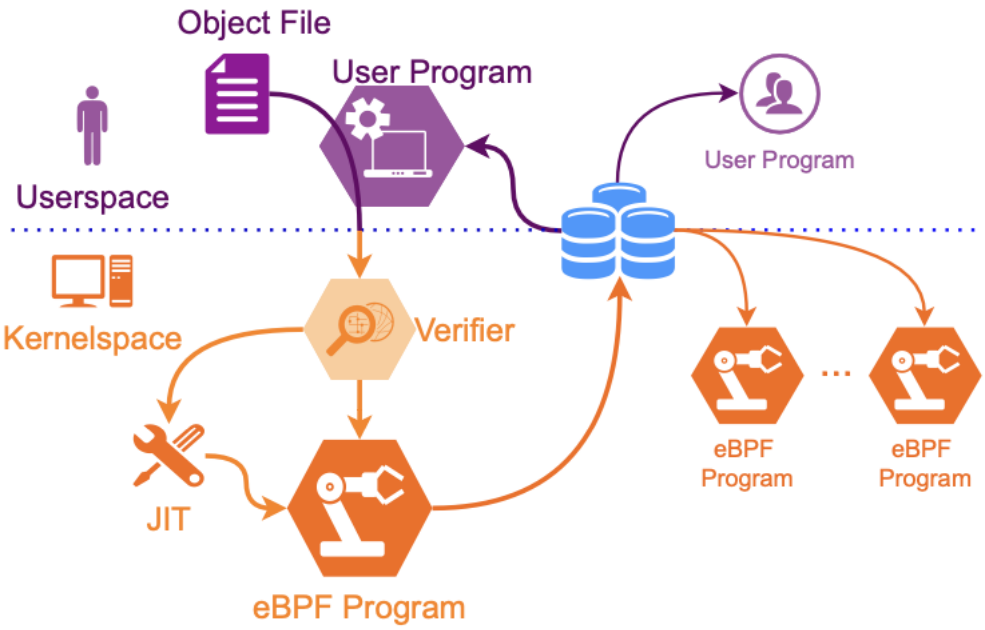

图 1.eBPF 的基本结构。

作为 Linux 内核的一部分，eBPF 提供了一种扩展 Linux 内核的可能性。系统操作者可以使用它以较低的成本捕获内核内的指标。然而，直接编写 eBPF 程序是困难的。图 1 显示了 eBPF 的基本结构和程序。首先，开发者应该用 C 语言编写 eBPF 代码，然后用 llvm 将其编译成一个对象文件，该文件将被加载到内核空间；其次，开发者应该编写一个程序来加载 eBPF，这个程序通常在用户空间运行。在对象文件被加载之前，内核运行一个验证器以确保它不会损坏内核。然后 JIT 将对象文件从字节码转移到机器指令。最后，eBPF 程序和用户程序通过 eBPF 地图相互通信，用户程序可以从地图中提取内核运行时的信息，或者要求 eBPF 程序通过向地图发送信号来改变内核的行为或自身。

编写 eBPF 代码是很难的，因为它需要理解 Linux 内核的源代码并遵守验证器的规则。开发环境也很复杂。近年来，出现了一些 eBPF 工具（如 BCC5），以减少开发 eBPF 程序的难度。但它对用 eBPF 监控微服务有以下两个挑战。

-eBPF 程序收集了许多不能直接利用的非数字指标（例如，§III-C 中显示的堆栈地址）。因此，我们必须将这些不可读的非数字数据翻译成人类可读的指标。

-eBPF 程序捕获的是与微服务容器的 PID 相对应的内核内指标。因此，eBPF 不能从语义上感知微服务的变化。我们必须赋予 eBPF 适应微服务和其他高层变化（例如，用户配置变化）的能力。

对于这些挑战，Chang 等人[2]使用 eBPF 对微服务剖析的指标进行 col- lect，但他们没有考虑微服务的变化。Shiraishi 等人[3]通过 eBPF 对微服务实现了动态传感器，但他们在调整监控项目时需要

创建或删除 eBPF 程序，这就造成了额外的过载。Viperprobe[1]是一个微服务收集框架，它只关注数字指标而忽略了非数字指标。

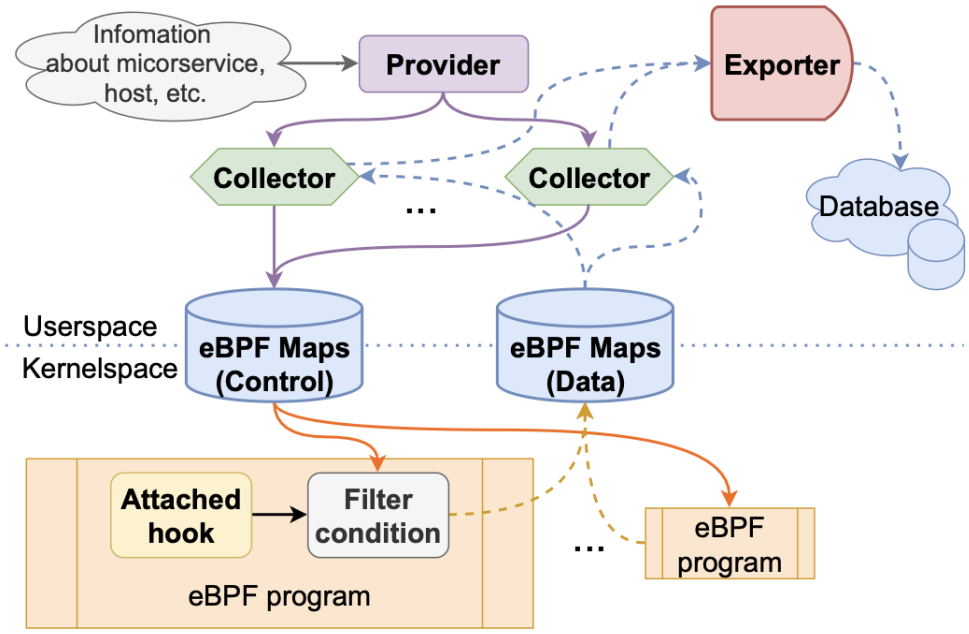

图 2.Kmon 的结构

Kmon 使用可视化工具，如 flamegraph 和 heatmap 来增加可视化高密度和非数字指标的可读性。Kmon 由提供者、收集者、输出者和 eBPF 组成。提供者收集高层次的信息，并将其发送给收集者以进一步转换指标。收集器通过 eBPF 地图控制 eBPF 程序的行为，对 eBPF 程序的指标进行转换和汇总。出口器形成指标并将其存储到数据库中。实验表明，Kmon 能够以较低的成本收集细粒度的指标。

本文的贡献可以归纳为以下几点：

-它　不仅使 eBPF 能够感知 mi-croservice 的变化，而且使它能够感知其他高层信息，如用户定义的配置变化。

-这是一个尝试，为配电系统诊断　存储、处理和可视化非数字的内核指标。

-它将多个内核内指标统一在一个系统中，数据格式相同，减少了指标收集和翻译的工作量。

II. 动力

首先，用统一的数据结构捕捉所有类型的指标，可以降低指标汇总的成本。现在有各种类型的监测工具。毫无疑问，它们在一些特定的领域中效果不错。但是，只用一种监测工具是很难获得一个系统的整体视图的。此外，每个工具的输出格式都有很大不同。因此，很难在一个视图中结合所有的指标。

其次，一个收集细粒度指标的监控系统可以帮助人们发现更多的问题。这里有一个关于活锁的例子：如果一个进程被卡在活锁中，它仍然会消耗 CPU 资源，而对其任务没有任何作用。它不能只用 CPU 使用率来反映活锁问题。进程之间发生的活锁将增加上下文切换的数量。如果我们能够获得上下文切换的数量或每个上下文切换的执行栈，我们就更有可能识别这样的问题。

第三，一个不需要仪器的系统可以降低开发和部署的成本。微服务中的一些指标，如 TCP 消息的延迟，是很难收集的。它们可以通过代码工具来捕获，或者通过部署软件，导致用户程序的额外性能损失，如 Istio。Kmon 的设计没有任何代码工具。因此，它对新的应用程序和运行时应用程序的影响很小。

III. 解决办法

A.系统结构概述

Kmon 在三个层面上监测微服务，这些微服务在实施时共享相同的架构。这是因为所有的指标都可以通过 eBPF 类似地收集。在描述每个层次的监测方法之前，首先介绍共享架构。图 2 说明了 Kmon 的架构。

-提供者 ：它收集高层次的信息，并发送给收集器以更新监测策略。

-采集器 ：他们加载 eBPF 程序，并通过 eBPF 地图与他们沟通。每个采集器负责一种类型的指标。

- eBPF 地图：eBPF 地图是内核中基于键/值的存储结构。在 Kmon 中，它们被用来从采集器向 eBPF 程序发送控制信息（称为控制图），并从 eBPF 程序向采集器收集度量（称为数据图）。

- eBPF 程序（s）：它们在内核中运行和钩住一些特定的事件和函数，以收集和存储数据到数据地图中。

-出口器 ：它从收集器接收指标。然后它汇总这些指标并将其发送到数据库。

B.提供者

提供者（图 3）负责收集高层信息。最初的 eBPF 是在内核层面使用的，所以很难捕捉到高层信息，如微服务信息和用户定义的配置。然而，由于 Kmon 是为微服务设计的，因此有必要为 eBPF 提供这些信息。如果没有提供 PID 来识别容器中的程序，eBPF 不知道它需要监控哪个程序；在用户改变配置的通知下，Kmon 节点可以改变其监控策略而不需要重新启动节点或 eBPF 程序。

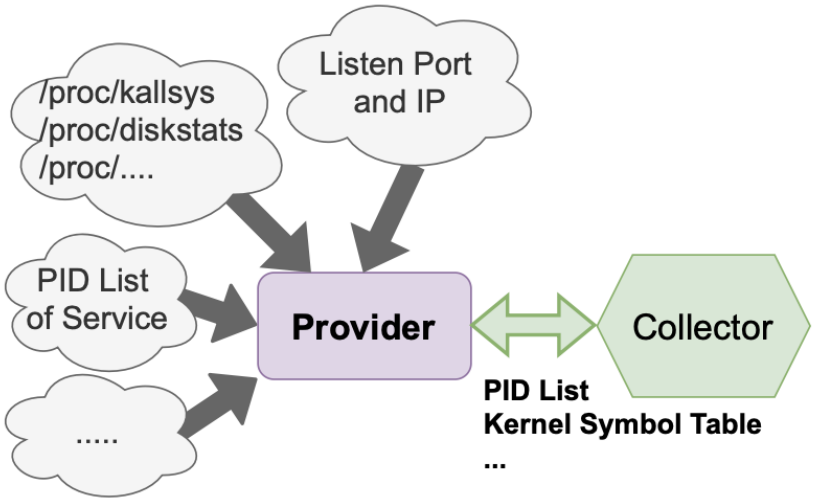

图 3.提供者为数据解析所探测到的高层信息

提供者还需要为收集器提供必要的信息来解析数据。例如，eBPF 可以在钩子触发器中收集内核堆栈，并以无符号长整数的形式存储地址，这对人类来说是不可读的，而且对于没有内核符号表的算法来说也很难分析。因此，如果一个指标包含内核堆栈，提供者应该为收集器收集内核符号表，将地址转换成符号名称。

C.采集器

采集器（图 4）与 eBPF 程序进行交互。它将 eBPF 程序加载到内核，并通过 eBPF 地图与 eBPF 程序进行通信。通信包含两部分：控制 eBPF 程序的行为和接收 eBPF 程序收集的数据。

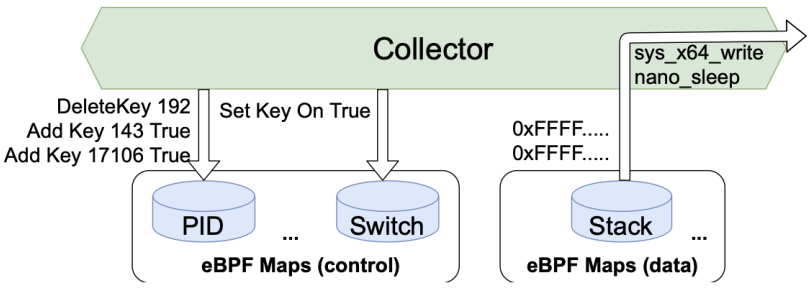

图 4.出口器的主要工作流程

例如，如果服务被删除或创建，收集器应立即通知 eBPF 程序改变其监控 PID 列表。如果用户改变了与某些采集器相关的配置，这些采集器需要将这些变化反映到控制图上。对于后者，采集器从数据地图中接收数据。这种原始数据（位图、字节数据、地址等）通常是人类无法阅读的，所以收集器应将它们转换为适当的形式。这些指标被发送到出口器并存储在数据库中

D. eBPF 地图和 eBPF 计划

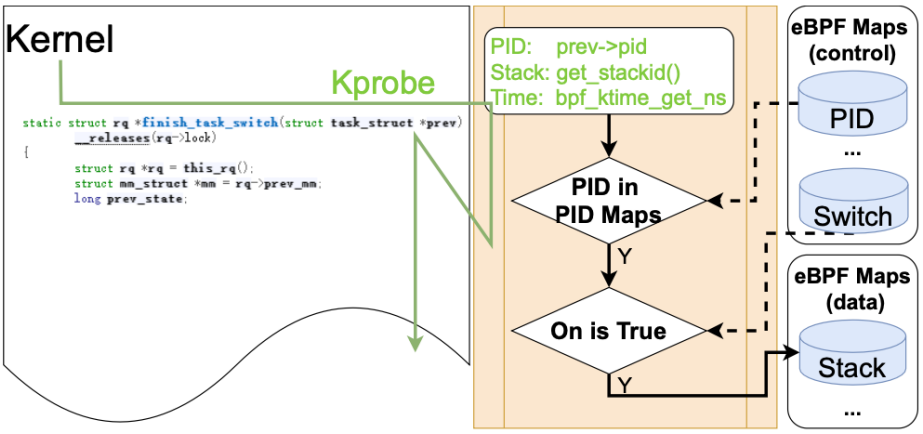

图 5.Kmon 的控制图和数据图

为了使 Kmon 成为一个灵活的系统，eBPF 程序（图 5）需要在微服务或配置改变时调整其行为。

eBPF 程序调整其行为的一种方法是重新编译和重新加载，这在 BCC 中使用。它需要在生产环境中安装重型编译环境（如 LLVM、Clang 等）。由于微服务变化频繁，每次变化都要重新编译 eBPF 程序，造成了很大的过载。

Kmon 以另一种方式进行调整，即把控制信息存储到 eBPF 地图中。eBPF 程序检查选择行政部门的地图。这些 eBPF 地图的用法与存储收集数据的 eBPF 地图不同。因此，我们将前者分别命名为控制地图和数据地图。通过这种方式，eBPF 可以在部署到系统之前进行预编译，避免了沉重的编译环境和运行时的重新编译过载。

E.出口商

输出器（图 6）将收集器收集的指标发送到数据库。出口器通过传输数据形式支持不同的数据库，以满足数据库的要求。

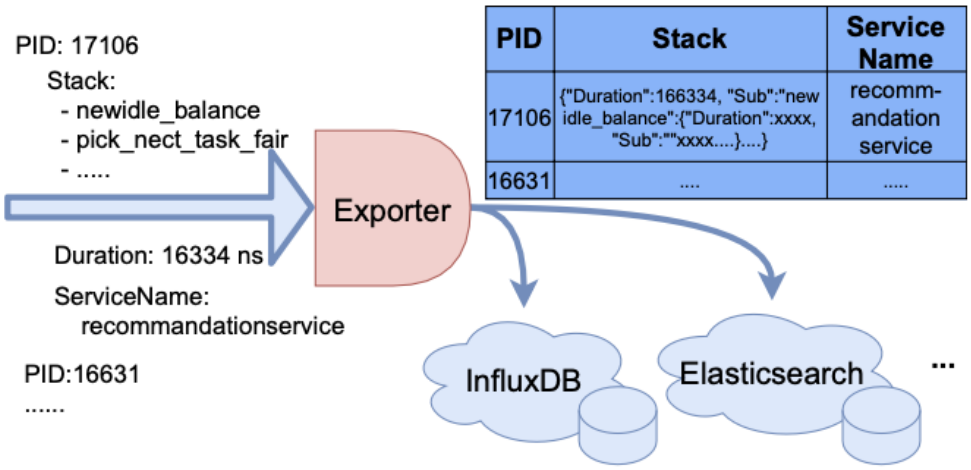

图 6.出口器的主要工作流程

IV. 实施内核内监控

Kmon 可以通过其架构收集各种类型的指标。有太多的类型可以决定在监测时应该关注哪些，所以这些指标的层次结构对于定位问题很有帮助。

Kmon 将指标分为三类（图 7）。第一类是 TCP 请求水平。TCP 请求在微服务中很常见，许多微服务的异常检测算法都依赖于它。在这个层面上，Kmon 捕获了每个 TCP 连接的指标，如四元数和延迟。

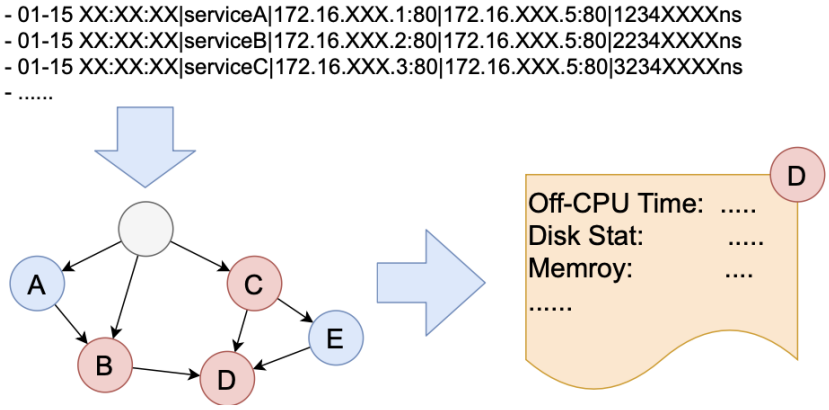

图 7.三个层次的性能指标。

第二层是拓扑结构层。动态服务图是由 TCP 请求数据形成的。图中的节点代表服务实例，而节点之间的链接反映了最近的网络请求流。它汇总了整个微服务系统中每个主机的信息。这个层次的

数据对人类和算法来说是直观的，可以识别服务之间的关系。根据延迟信息和拓扑结构，系统操作者可以方便地估计图中是否有异常节点。

第三层包含其他细粒度的指标供进一步调查。它包括与 CPU、内存、块 I/O 等相关的指标。Kmon 的 eBPF 程序的源代码参考了 BCC 的例子，并做了一些修改以符合 Kmon 的架构。

A.TCP 请求监控

这里是一个收集 TCP 请求指标的例子。Tbl.I 显示了提供者应该提供的东西，Tbl.II 显示了 eBPF 程序应该监控的内核函数。请求延迟是通过从最后发送的消息时间戳中减去第一个收到的消息时间戳来计算的（图 8），所以 eBPF 应该钩住相对的函数调用来捕获时间。

TABLE I
HIGH-LEVEL INFORMATION

PID	Identify receiver of requests.
Service name	Convert PID to service name for readability.
IP listen list	Identify server side for persistent connection.

TABLE II
HOOKED KERNEL FUNCTIONS

tcp_sendmsg ***tcp_cleanup_rbuf***	Capture the time when TCP message sent and received, for latency timing.
security_socket_accept ***tcp_close***	Capture the time when TCP connection accept and close, for identify server side for short connections.

Kmon 只收集发送到服务器的请求，但它很难

来区分一个套接字被接受后的方向。Kmon 使用关于 "接受 "和 "关闭 "操作的钩子来决定短连接方向，它假定服务器端执行了 "接受 "系统调用。

但是，它不适合于持久性连接。如果一个连接是在 Kmon 部署之前建立的，Kmon 就不能感知它。因此，提供者应该收集监听端口信息。Kmon 认为，如果本地主机在服务器端，它的 IP 和端口应该在本地主机的 IP 监听列表中。

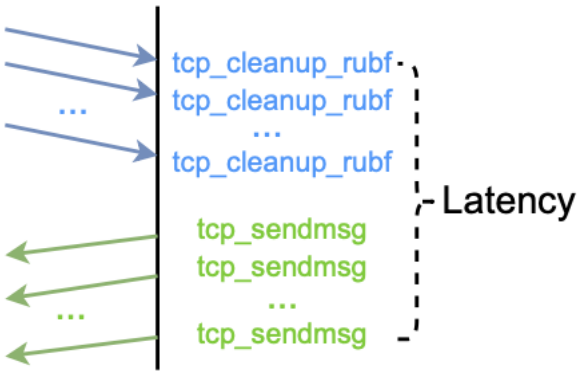

图 8.TCP 延迟的计算。

B.服务拓扑监测

服务拓扑结构可以通过 TCP 请求的数据来构建。它搜索一个时期内发生的所有请求，根据其来源和目的地绘制节点和边缘。这是一个动态的拓扑结构，因为在不同的时期会构建不同的拓扑结构，这适合于微服务的频繁变化。来自 TCP 请求的数据足以构建拓扑结构，所以不需要更多的收集器。

C.细粒度的性能指标

其他指标与 Kmon 的结构相同，但有不同的 eBPF 程序和收集器。Tbl.III 和 Tbl.IV 显示了提供者和 eBPF 程序收集的一些已实施指标的信息。

网络：我们选择 tcp drop 作为一个例子。它可以捕获每个被内核丢弃的数据包，并为每个丢弃的数据包提供一个堆栈。它可以调试掉包的高速率。有许多关于网络的指标，如 tcp connect 和 tcp accept。我们介绍 tcp drop 是因为其他的都是类似的，而我们现在只实现了 tcp drop。

CPU：非 CPU 时间 6 可以捕捉到进程被阻断的内容和时间。这对于分析进程或内核行为的细节是很有用的 7。因为捕获这个指标的成本很高，因为跟踪调度器被频繁调用 8，Kmon 只是存储那些让进程进入 "睡眠 "状态的数据，而不是每个调度（虽然它仍然是高成本。）这个指标需要存储每个调度的堆栈，这需要大量的内存。受 Flame Graph 的启发，Kmon 将堆栈翻译成树状存储。每个函数的调用都是树上的一个节点，它的总持续时间。采集器为每个间隔（例如，5s）构建树，以追踪进程的变化。

块状 I/O：图三中的三个钩子可以捕获许多指标，如 I/O 类型和吞吐量。这些指标主要是通过 "request "结构的指针获得的，该结构是挂钩函数的参数。它还可以通过从 "blk account io start "的时间戳减去 "blk account io done "的时间戳来捕获 I/O 延迟。 这对优化 I/O 性能很有用，例如，将总是在同一时间写入的服务实例放在不同的机器上。

TABLE III
HIGH-LEVEL INFORMATION

Configuration	For changing Kmon's behaviour.
PID	Identify and filter program of monitoring.
Service name	Convert PID to service name for readability.
Kernel symbol table	Convert stack address to symbol name.
Disk name	Convert major and minor number of disk to name.

TABLE IV
HOOKED KERNEL FUNCTION

Type	***Hook***	***descirbtion***
Network	tcp_drop	Capture TCP packets or segments that were dropped by the kernel.
Block I/O	blk_account_io_start blk_mq_start_request blk_account_io_done	Capture indicator about block I/O, such as read-write type, throughput, latancy, I/O.
CPU	activate_task deactivate_task	Capture the on and off CPU counts, time and its stack for specific program.

我们还实现了一个通用的系统调用钩子，以计算用户指定的特定内核函数（如 tcp connect, tcp drop, write, and read）的调用数量。一些内核函数如系统调用可以反映程序的类型，例如，对于一个频繁的 I/O 服务，执行 "读 "或 "写 "的数量可能很高。

V. 实验设置

我们的实验使用四个 Linux 虚拟机，Linux 内核 v5.4，组成一个 Kubernetes 集群。我们使用 "Hipster shop "9，一个来自谷歌的云原生微服务应用演示作为基准。它包含 10 层微服务，用户可以在上面浏览商品，将它们添加到购物车，并购买它们。我们把它的负载生成器改为 k6，它可以在负载生成后获得更多的指标。在所有实验中，k6 的负载被设定为 100 个用户。

Kmon 节点通过 Kubernetes 以新的 ClusterRole 部署到每个主机上。ClusterRole 是获得容器中 Kmon 节点执行 eBPF 和收集 Kubernetes 信息（如命名空间、pod 名称、服务名称、节点等）的权限所必需的。服务和 Kmon 节点的资源使用情况由一个指标服务器记录。

A.性能指标

为了估计一个 Kmon 节点的指标的资源使用情况，第一个实验在没有 Kubernetes(K8s)的帮助下手动运行 Kmon 节点程序，并且禁用 Exporter。它避免了存储的网络过载，所以我们可以专注于指标监测部分。

运行 Kmon 节点的主机包含 10 个服务中的 3 个（"paymentervice"、"emailservice "和 "frontend"）。在第一个实验中，Linux 中的工具 "top "被用来测量 Kmon 节点的资源使用。

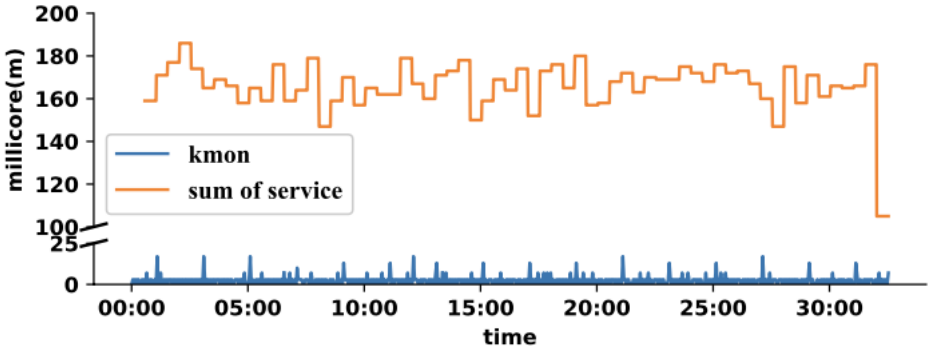

图 9.单个主机中 micrioserive 和 Kmon 节点的 CPU 使用率。

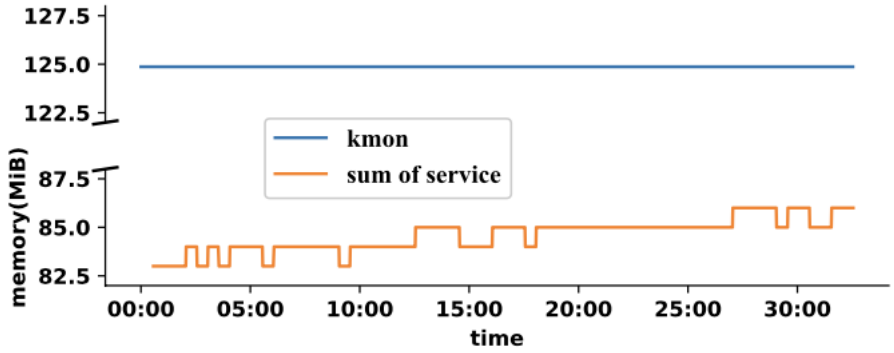

图 10.单个主机中 micrioserive 和 Kmon 节点的内存用量。

图 9 和图 10 显示了 Kmon 节点的 CPU 和内存使用情况。CPU 的使用（不包括 eBPF 程序）可以忽略不计，而内存的使用则相对较高。有可能对内存使用进行更多的优化。我们将在未来做这件事。

在测量完一个 Kmon 节点的资源使用情况后，有必要对 Kmon 在整个微服务中的性能进行评估。在这一部分，Kmon 被部署在每个主机上，收集了 4 种指标，分别是网络（TCP 请求延迟，丢弃信息），块 I/O 状态（吞吐量，计数，类型，堆栈），功能计数器（系统调用 "写 "和 "读"），和非 CPU 时间（堆栈，持续时间）。Tbl.V 显示了导出的 TCP 请求数据的一些结果，其中包含时间戳、延迟源和目的地。

Kmon 节点在 30 分钟内的平均资源使用量是 9.55%（CPU 使用率）和 616.11MB（内存使用率）。

表六显示了对响应时间的影响：

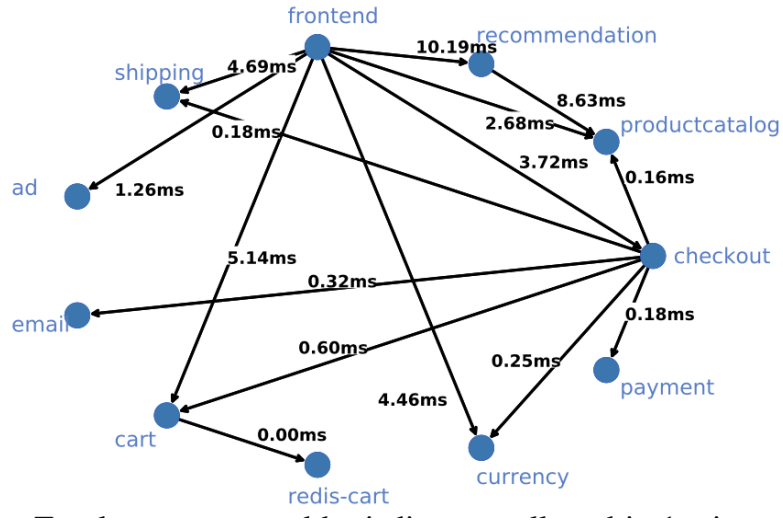

图 11.由 1 分钟内收集的指标构建的拓扑结构。

图 11 显示了从这部分实验中收集的网络数据对请求的拓扑可视化。对于一个 pod 中更精细的指标，如 "推荐服务"，图 13 显示了 K6 开始产生负载时的状态。荚中的程序睡眠时间更长，频率更高。结合图 12，它显示了在每种类型的堆栈中程序离开 CPU 时的持续时间。睡眠（do nano sleep）、同步（futex wait queue me）和 epoll（poll schedule ...）可能导致其增加。

VI. 相关工作

A.侵入性监测框架

侵入性的监控框架需要改变用户程序的源代码或二进制文件，这就引入了额外的开销。Pythia[5]专注于在分布式应用中以直观的方式对哪里、什么、何时进行监控。Seer[6]和 MicroRank[9]对微服务应用的请求信息进行跟踪测量，以匹配服务的模式。工作[10]通过将 API 钩子插入到分布式系统诊断的源代码中来增加可观察性。与之前的工作相比，用户可以在 Komn 的帮助下获得详细的监控信息，而不需要任何仪器。

B.非侵入式监控框架

当代的内核内监测工具，如 ftrace10 和 sysdig[4]，可以获得单个主机上的细粒度指标，但它们很难收集和汇总微服务环境中所有节点的指标。Microscope[11]捕获了微服务系统的网络连接信息，以监测服务依赖图的变化。Microscaler[12]，[13]使用服务网格来监控微服务系统的指标。然而，

Microscope 和 Microscaler 不能获得系统杠杆度量。

Chang 等人，[2]使用贝叶斯模型来分析由 valtrace 收集的 eBPF 的数据。[7]中的研究使用随机森林模型来分析虚拟机中 eBPF 的网络相关指标。这两项研究都集中在算法上，而不是适应动态环境。与之前的研究相比，Kmon 更适合动态的微服务环境，可以有效地获得细粒度的指标。

VII. 总结

为了创建一个具有细粒度指标的透明监测系统，我们介绍了 Kmon，一个基于 eBPF 的系统。它透明地捕获各种类型的指标，并将其组织在三个层次上，这对系统操作者和算法来说是很方便的。实验表明，它的 CPU 使用率很低，对服务响应时间的影响很小。

在未来，我们的目标是找到一种更好的方法，在更少的假设上捕获 TCP 连接信息。目前对 TCP 请求的假设并不适合某些特定类型的服务，如消息队列。Kmon 的内存用量也需要减少。我们考虑了两个方向。一个是使用 libbpf 库而不是 libbcc 库，这样在运行时使用的内存更少，而且具有可移植性；另一个是找到一个更好的指标表示方法来压缩其大小。

TABLE V
RECORD OF TCP MESSAGE SENDING

time	*service*	*latency (ns)*	*myIP:port*	*peerIP:port*
Jan 15, 2021 @ 21:56:13.206	emailservice	375170	172.20.1.132:8080	172.20.2.134:51614
Jan 15, 2021 @ 21:56:13.200	shippingservice	676425	172.20.1.129:50051	172.20.2.134:50330
Jan 15, 2021 @ 21:56:13.199	cartservice	594035	172.20.1.132:7070	172.20.2.134:53604

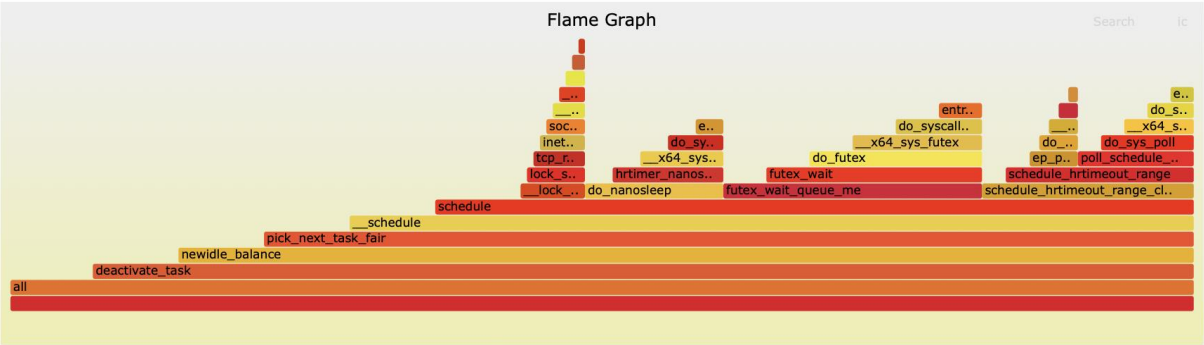

图 12.一个进程的非 CPU 时间的火焰图。火焰图可以反映出一个程序是如何度过它的时间的。每个条形图都是随机的暖色，并按字母顺序排列，最上面的条形图显示离开 CPU 时间的长短，下面是它的祖先。条形图的 Y 轴显示堆栈深度。在这幅图中，大部分离开 CPU 的时间是由函数 futex（同步）和 poll（网络）获得的。

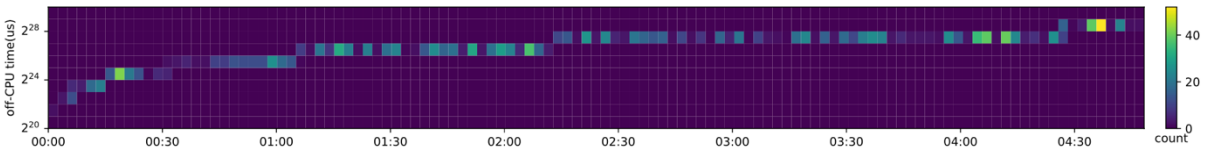

图 13.负载发生器开始运行时，"推荐服务 "在 4 分钟内的脱机时间热图（仅在睡眠状态下）。

TABLE VI
RESPONSE INFLUENCE FOR MICROSERVICE IN 30 MINUTES

	With Kmon Node	***Without Kmon Node***
Response-time(Avg)	361.97ms	348.97ms
Response-time(P95)	790.09ms	784.09ms

Kmon: An In-kernel Transparent Monitoring System for Microservice Systems with eBPF

VIII. INTRODUCTION

Attracted by the characteristics of high flexibility and fast de- livery, microservice is widely used in many modern companies now, such as Google and Microsoft. To monitor and manage microservice systems, system operators need to collect per- formance indicators that reflect system states and persistently store those indicators in a database.

S. ome conventional monitoring tools (e.g., cAdvisor1) can help system operators to collect and aggregate indicators. However, most of them only focus on basic metrics of resource usage(e.g., CPU utilization). Another type of monitoring tools like Istio2 can monitor more indicators like requests latency at L7 network layer, but they need to change the underlying infrastructure. To gain diversified metrics for existing systems without modification, we propose Kmon, an in-kernel transpar- ent monitoring system for microservice systems.

Kmon can capture conventional metrics more accurately. Furthermore, it can collect fine-grained in-kernel performance indicators(e.g., the number of system calls). In-kernel indica- tors are useful because they can reflect hidden problems in deep (e.g., livelock shown in §II or limplock [14]). It is almost impossible to find out these deeper problems with current monitoring tools since they cannot obtain in-kernel events. Therefore, it is important to collect in-kernel indicators for microservice observability.

Although previous tools like strace3 can capture in-kernel indicators, they sacrifice the performance of the system and need to aggregate manually for distributed systems. Other methods (e.g., OpenTelemetry4) need to change the source code of the user's program, which introduces additional complexity. Kmon chooses to use eBPF, a component of the Linux kernel to collect metrics of programs without instrumentation. It reduces the difficulty of using eBPF and enables eBPF to sense changes in application layer, especially for the changes of microservice. Users can simply use it by write configuration and deploy Kmon to each host. Then Kmon can automatically collect metrics and store them into databases. The

configuration specifies metrics users want to gain and server instances in microservice that need to be monitored, which does not require modifying source or provide internal logic information of programs.

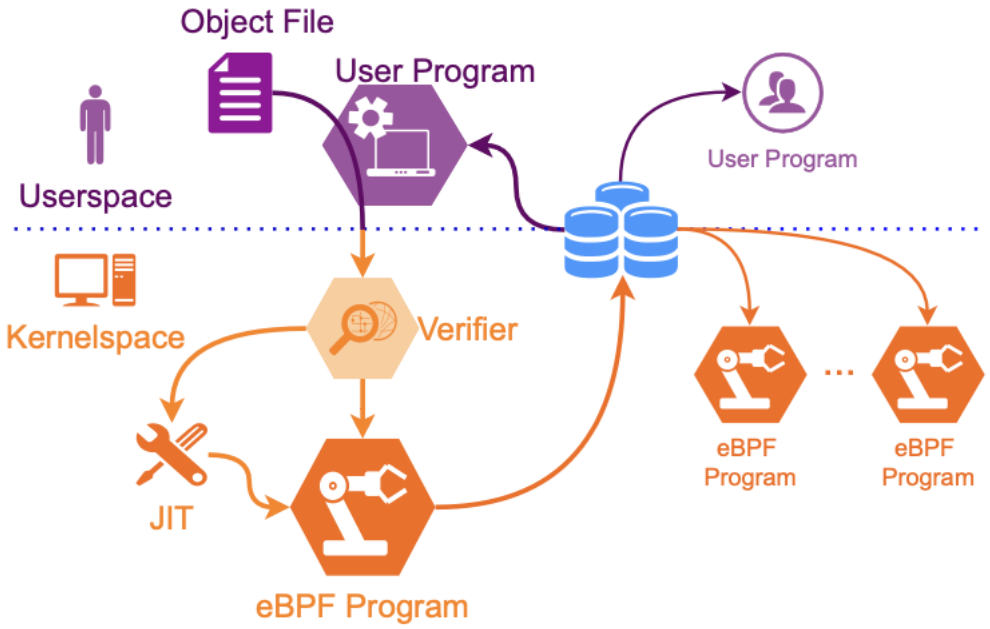

Fig. 1. The Basic Architecture of eBPF.

As part of the Linux kernel, eBPF provides a possibility to extend the Linux kernel. System operators can use it to capture in-kernel indicators at a low cost. However, write eBPF program directly is hard. Fig. 1 shows the basic architecture and procedure of eBPF. First, a developer should write eBPF code in C, then use llvm to compile it into an object file, which will be loaded into kernelspace; Second, the developer should write a program to load eBPF, this program usually running on userspace. Before the object file is loaded, the kernel runs a verifier to ensure it cannot damage the kernel. Then JIT transfers the object file from byte code to machine instructions. Finally, eBPF program and user program communicate with each other by eBPF maps, the user program can extract kernel runtime information from maps, or ask eBPF program to change the behavior of the kernel or itself by sending signals to maps.

Writing eBPF code is hard because it needs to understand the source code of the Linux kernel and observe the rules of the verifier. The development environment is also complex. In recent years, some eBPF tools (e.g., BCC5) have emerged to reduce the difficulty of developing eBPF programs. But it has two challenges to monitoring microservice with eBPF as follows.

• The eBPF program collects many non-numerical indica- tors (e.g., stack address shown in §III-C) that

cannot be utilized directly. Thus, we must translate those unreadable non-numerical data to human-readable indicators.

• The eBPF program captures the in-kernel indicators that are corresponding with the microservice containers' PID. Therefore, eBPF cannot sense the changes of microser- vices semantically. We must give eBPF the ability to adapt to microservice and other high-level changes(e.g., user configuration changes).

For these challenges, Chang et al. [2] used eBPF to col- lect indicators for microservice profiling, but they have not considered the changes of microservices. Shiraishi et al. [3] implemented dynamic sensors to microservices through eBPF, but they need to create or delete eBPF programs when adjusting monitor items, which causes extra overload. Viperprobe [1] is a microservices collection framework, which focuses only on numerical metrics and ignores the non-numerical indicators.

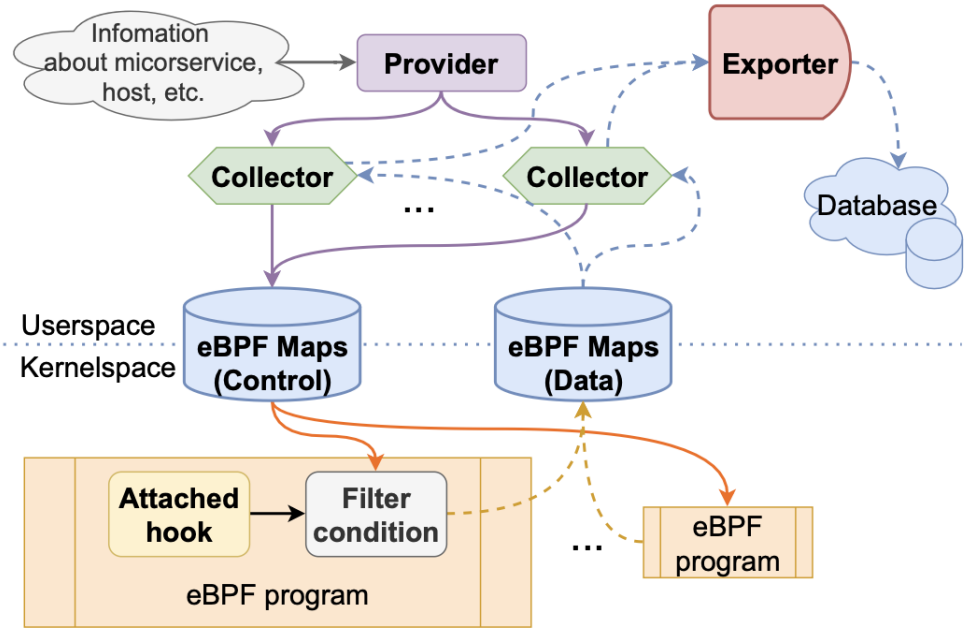

Fig. 2. The Architecture of a Kmon

Kmon use visualize tools like flamegraph and heatmap to increase readability for visualization high-density and non- numerical indicators. Kmon consists of Provider, Collector, Exporter, and eBPF. Provider collects the high-level informa- tion and sends them to Collector for further indicators trans- formation. Collector controls the behavior of eBPF programs through the eBPF map, transforms and aggregates indicators from eBPF programs. Exporter forms the indicators and stores them into databases. Experiments show that Kmon can collect fine-grained indicators at a low cost.

The contributions of this paper can be summarized as follows:

• It not only enables eBPF to sense the changes of mi- croservice, but also enables it to sense other high-level information like changes of user-defined configuration.

• It is an attempt at storing, handling and visualizing non-numeric in-kernel metrics for the distribution system diagnosis.

• It unifies multiple in-kernel metrics in one system with the same data format, reduces the workload of metrics collecting and translating.

IX. MOTIVATION

Firstly, capturing all types of indicators with a unified data structure can reduce the cost of indicator aggregation. There are various types of monitoring tools now. There is no doubt they work well in some specific fields. However, it is difficult to obtain a holistic view of a system with only one monitor tool. Moreover, the output format of each tool is quite different. Therefore, it is hard to combine all indicators in one view.

Secondly, a monitoring system that collects fine-grained indicators can help people find out more problems. Here is an example of livelock: if a process gets stuck in a livelock, it still consumes CPU resources while does nothing useful for its tasks. It cannot reflect a livelock problem only with CPU usage. Livelock occurs between processes that will increase the number of context switches. If we can obtain the number of context switching or the execution stack of each context switch, we are more likely to recognize such a problem.

Thirdly, a system that does not need instrumentation can reduce the cost of development and deployment. Some metrics in microservice, such as the latency of TCP messages, are hard to collect. They can be captured via code instrument, or by deploying software that results in an additional performance loss of users' programs like Istio. Kmon is designed without any code instrument. Thus it has little impact on the new applications and runtime applications.

X. APPROACH

A. System Architecture Overview

Kmon monitors microservices at three levels, which share the same architecture when implemented. That is because all indicators can be similarly collected by eBPF. Before describing methods of monitoring

at each level, the shared architecture is introduced first. Fig. 2 illustrates the architecture of Kmon.

- Provider: It collects high-level information which are sent to Collectors for updating monitoring strategy.
- Collector(s): They load eBPF program and communicate with them by eBPF maps. Each Collectors are responsible for a type of indicator.
- eBPF Maps: eBPF maps are the key/value-based storage structure in the kernel. In Kmon, they are used to send control messages from Collector to eBPF programs (called control maps), and collect metrics from eBPF program to Collector (called data maps).
- eBPF Program(s): They run and hook some specific events and functions in kernel to collects and store data into data maps.
- Exporter: It receives the metrics from Collectors. Then it aggregates those metrics and sends them to the database.

B. Provider

Provider (Fig. 3) is responsible for collecting high-level information. The initial eBPF is used at the kernel level, so it is hard to capture high-level information, such as microservice information and user-defined configuration. However, as Kmon is designed for microservice, it is necessary to provide this information for eBPF. Without PID provided to identify the program in a container, eBPF does not know which program it needs to monitor; With notice of configuration changing by a user, Kmon node can change its monitoring strategy without restarting the node or the eBPF program..

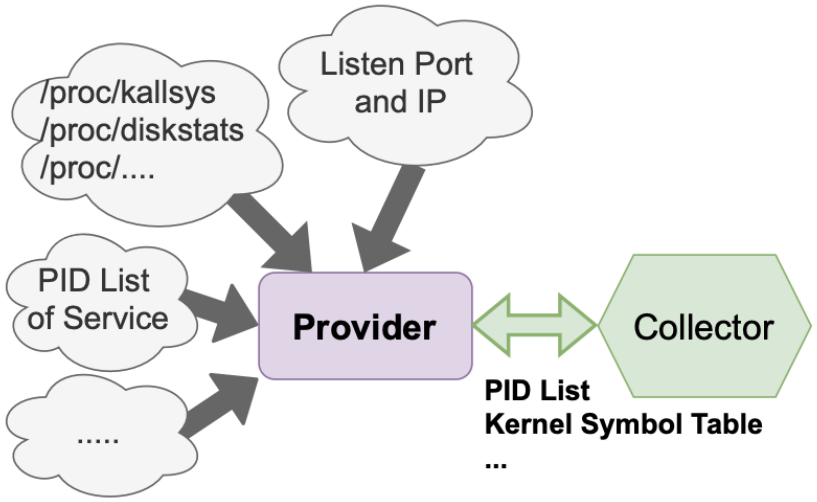

Fig. 3. The high-level information probed by Provider for data parsing

Provider also needs to provide necessary information for Collector to parse data. For example, eBPF can collect kernel stack in hook trigger, and stores addresses in form of unsigned long integer, which is unreadable for humans, and hard to analyze for algorithms without a kernel symbol table. So if an indicator contains kernel stacks, Provider should collect the kernel symbol table for Collector to convert addresses to symbol names.

C. Collector

The Collector (Fig. 4) interacts with eBPF programs. It loads the eBPF program into kernel and communicates with eBPF programs via eBPF maps. Communications contain two parts: to control behaviors of eBPF programs and to receive data collected by eBPF programs.

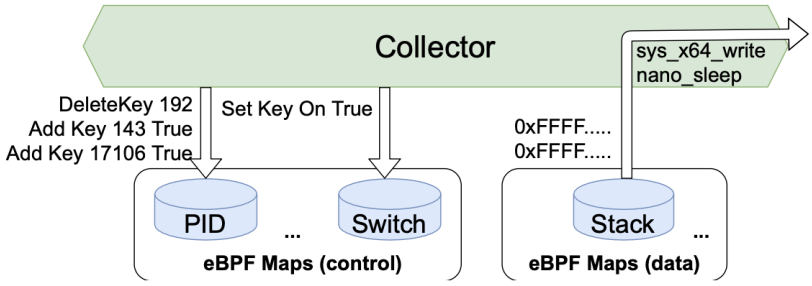

Fig. 4. The primary workflow of Exporter

For example, if services are deleted or created, Collectors should immediately notify eBPF programs of chang- ing their monitoring PID list. If a user changes configurations that are relative to some Collectors, these Collectors need to reflect these changes to control maps. For the latter, Collectors receive data from

data maps. This raw data (bitmap, bytes data, address, etc.) are usually human unreadable, so Collectors should convert them to an appropriate form. These indicators are sent to Exporter and stored in databases

D. eBPF Maps and eBPF Program

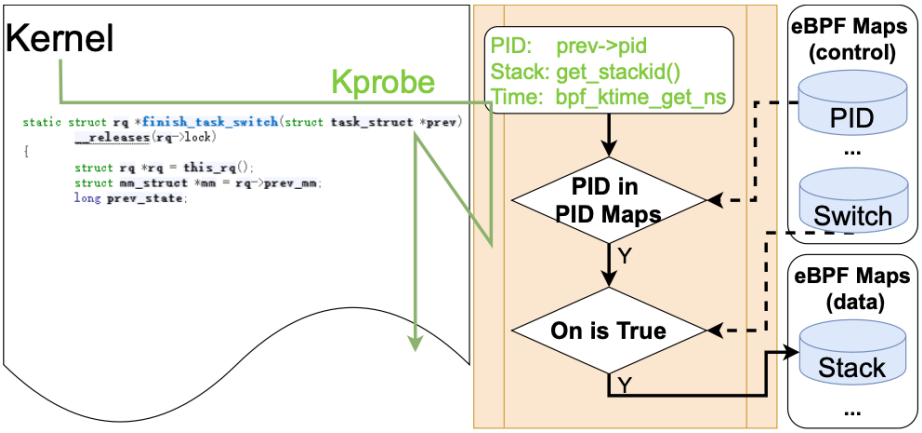

Fig. 5. The control maps and data maps in Kmon

To make Kmon a flexible system, eBPF programs(Fig. 5) need to adjust their behaviors when microservice or configu- ration changes.

One way for eBPF programs to adjust their behavior is to recompile and reload them, which are used in BCC. It needs to install the heavy compiling environment (e.g., LLVM, Clang, etc.) in production environment. As microservice changes frequently, recompiling of eBPF program for each change causes much overload.

Kmon adjusts in another way, namely storing the control message into eBPF maps. eBPF programs check maps for choosing executive branch. The usage of these eBPF maps is different from eBPF maps that store collected data. Therefore, we name the former control maps and data maps respectively. In this way, eBPF can be precompiled before deployed in the system, avoiding the heavy compiling environment and the recompilation overload in run-time.

E. Exporter

Exporter (Fig. 6) sends indicators collected by Collector to databases. Exporter supports different databases by transfer data forms to meet the requirements of databases.

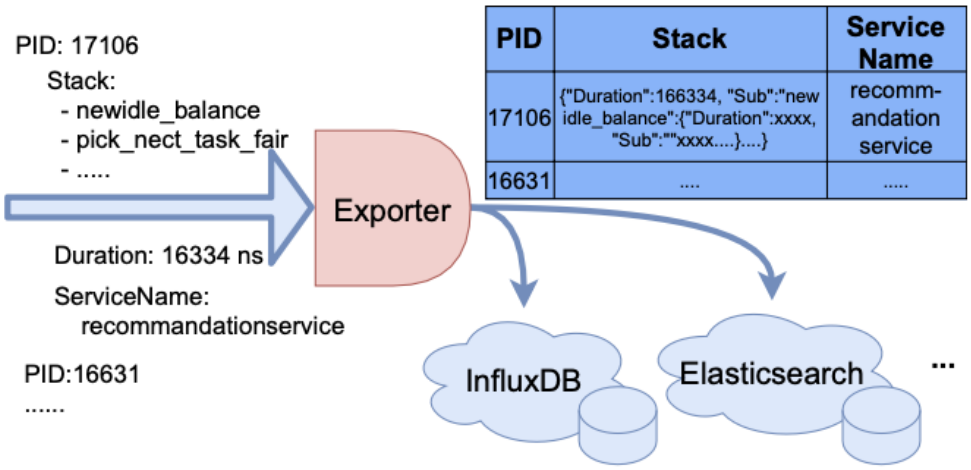

Fig. 6. The primary workflow of Exporter

XI. IMPLEMENTATION OF IN-KERNEL MONITORING

Kmon can collect various types of indicators via its archi- tecture. There are too many types to decide which should be focused on when monitoring, so a hierarchy of these indicators for locating the problem is helpful.

Kmon classifies indicators into three categories(Fig. 7). The first is TCP request level. TCP requests is common in microser- vice and many abnormal detecting algorithms for microservice depend on it. In this level, Kmon captures indicators of each TCP connection such as quaternion and latency.

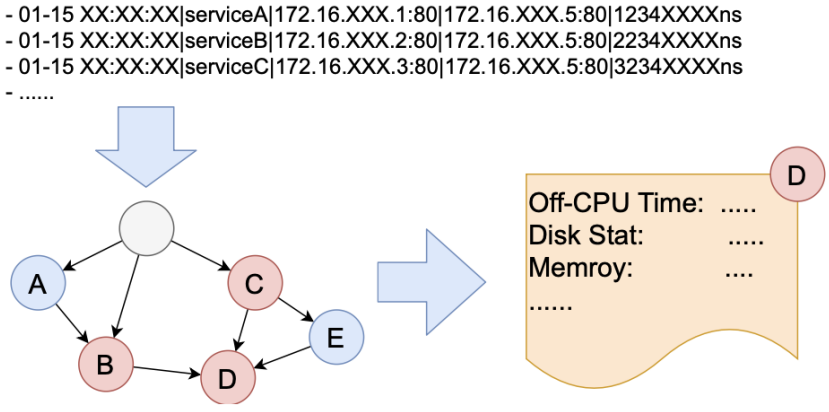

Fig. 7. Three levels of performance indicators.

The second level is the topology level. Dynamic service graph are formed by TCP request data. Nodes in the graph represent service instances, and the link between nodes reflects flows of recent network

requests. It aggregates information of each host from the whole microservice system. Data at this level is intuitive for humans and algorithms to identify the relation between services. According to the latency information and topology, it is convenient for system operators to estimate if there is an abnormal node in the graph.

The third level contains other fine-grained indicators for fur- ther investigation. It includes metrics relative to CPU, memory, block I/O, etc. The source code of eBPF programs for Kmon refer to BCC examples, with some changes to match Kmon architecture.

A. TCP Request monitoring

Here is an example of collecting indicators of TCP request. Tbl. I shows what should be provided by Provider and Tbl. II shows what kernel function should eBPF program to moni- tor. The request latency is calculated by subtracting the first received message timestamp from the last sending message timestamp(Fig. 8), so eBPF should hook relative function call to capture the time.

TABLE I
HIGH-LEVEL INFORMATION

PID	Identify receiver of requests.
Service name	Convert PID to service name for readability.
IP listen list	Identify server side for persistent connection.

TABLE II
HOOKED KERNEL FUNCTIONS

tcp_sendmsg ***tcp_cleanup_rbuf***	Capture the time when TCP message sent and received, for latency timing.
security_socket_accept ***tcp_close***	Capture the time when TCP connection accept and close, for identify server side for short connections.

Kmon only collects requests sent to servers, but it is difficult

to distinguish the direction after a socket being accepted. Kmon uses the hook about “accept” and “close” operation to decide short connection direction, which assumes that the server-side executes the “accept” system call.

However, it is not suitable for a persistent connection. If a connection was made before Kmon has been

deployed, Kmon cannot sense it. Hence Provider should collect the listen port infomation. Kmon assumes that if the local host is on the server-side, its IP and port should be in the local host's IP listening list.

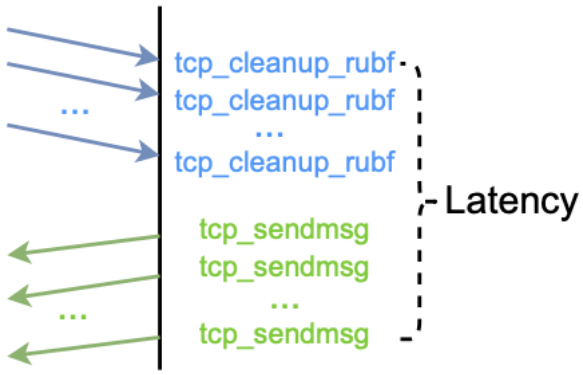

Fig. 8. TCP latency calculation.

B. Service Topology Monitoring

Service topology can be constructed by the data from TCP requests. It searches all requests that occur in a period, draws node and edge according to its source, and destination. It is a dynamic topology as different topologies are constructed in different periods, which is suitable for microservice for its frequent changes. The data from TCP requests to construct topology is sufficient, so no more Collectors are needed.

C. Fine-grained Performance Indicators

Other indicators share the same architecture of Kmon, but with different eBPF programs and Collectors. Tbl. III and Tbl. IV show the information about some implemented indi- cators that the Provider and the eBPF program collect.

Network: we choose tcp drop as an example. It can capture each packet dropped by kernel, with a stack for each dropping. It can debug high-rate of drops. There are many indicators about network like tcp connect and tcp accept. We introduce tcp drop because others are similar and we have only imple- mented tcp drop now.

CPU: Off-CPU time6 can capture what and when a process is blocked. It is useful to analyze details of the process's or kernel's behaviors7. Because capturing this indicator has high cost as tracing scheduler is called frequently8, Kmon just stores data for those who let processes go to "sleep" state instead of each scheduling (While it is still high-cost.). This indicator needs to store stacks of each schedule, which need high memory. Inspired by Flame Graph, Kmon translate and stores stack in trees. Each function call is a

node of the tree with its total durations. Collector constructs trees for each interval (for example, 5s) to trace changes of a process.

Block I/O: Many metrics can be captured from three hooks in Fig. III, such as I/O type and throughput. These metrics are mainly gain from pointer of ”request” struct, which is the parameter of hooked function. It can also capture I/O latency by subtracting timestamps of “blk account io start” from times- tamps of “blk account io done” It is useful to optimize I/O performance, for example, placing service instances that always write at the same time to different machines.

TABLE III
HIGH-LEVEL INFORMATION

Configuration	For changing Kmon’s behaviour.
PID	Identify and filter program of monitoring.
Service name	Convert PID to service name for readability.
Kernel symbol table	Convert stack address to symbol name.
Disk name	Convert major and minor number of disk to name.

TABLE IV
HOOKED KERNEL FUNCTION

Type	***Hook***	***descirbtion***
Network	tcp_drop	Capture TCP packets or segments that were dropped by the kernel.
Block I/O	blk_account_io_start blk_mq_start_request blk_account_io_done	Capture indicator about block I/O, such as read-write type, throughput, latancy, I/O.
CPU	activate_task deactivate_task	Capture the on and off CPU counts, time and its stack for specific program.

We also implement a generic system call hook to count the number of specific kernel functions called(e.g. tcp connect, tcp drop, write, and read) specified by user. Some kernel functions like system calls can reflect the type of program, for example, the number of execution “read” or “write” may be high for an I/O frequent service.

XII. EXPERIMENT SETTING

Our experiment uses four Linux virtual machines with Linux kernel v5.4 to make up a Kubernetes

cluster. we use “Hipster shop”9, a cloud-native microservices application demo from google as a benchmark. It compromises 10-tier microservice on which users can browse items, add them to the cart, and purchase them. We change its load generator to k6, which can get more metrics after load having been generated. The load of k6 is set to be 100 users in all experiments.

Kmon node is deployed to each host via Kubernetes with a new ClusterRole. ClusterRole is necessary to get permission for Kmon node in a container to execute eBPF and collect information of Kubernetes (e.g., namespace, pod name, service name, nodes, etc.). The resource usage of service and Kmon node is recorded by a metric server.

A. Performance metrics

To estimate the resource usage of an indicator of a Kmon node, the first experiment runs Kmon node program manually without the help of Kubernetes(K8s), and with the Exporter disabled. It avoids the network overload of storage, so we can focus on the indicator monitoring part.

The host which runs the Kmon node contains 3 of 10 services (“paymentservice”, “emailservice”, and “frontend”). In the first experiment, Tools “top“ in Linux is used to measure the resource usage of Kmon nodes.

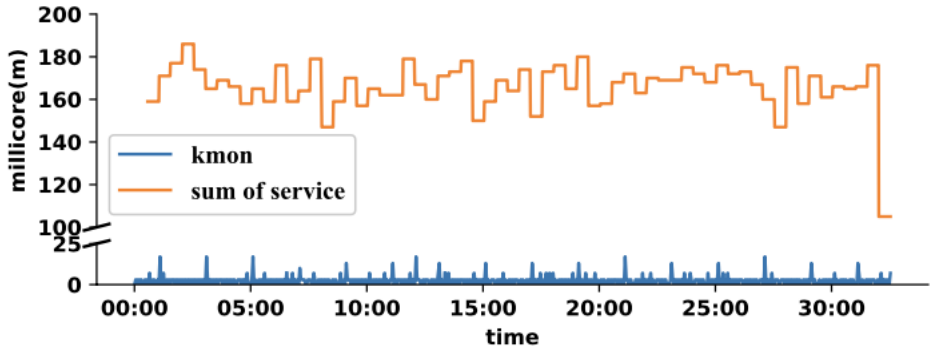

Fig. 9. CPU usage for micrioserive and Kmon node in single host.

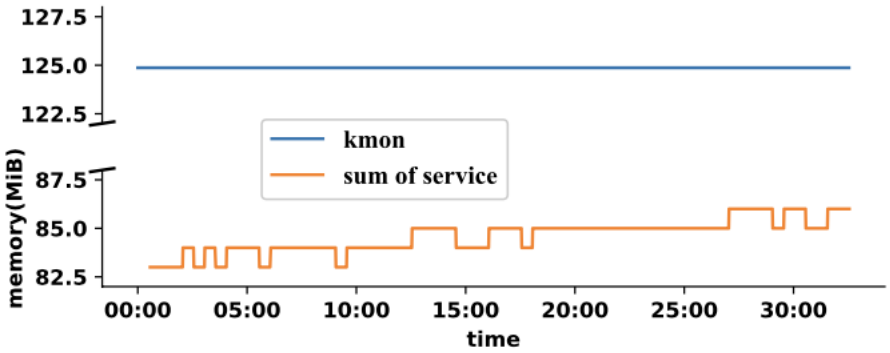

Fig. 10. Memory usage for micrioserive and Kmon node in a single host.

Fig. 9 and Fig. 10 show the CPU and memory usage of the Kmon node. The CPU usage (exclude eBPF program) is negligible while memory usage is relatively high. It is possible to have more optimization for memory usage. We will do this in the future.

After measuring the resource usage of one Kmon node, it is necessary to estimate Kmon's performance in the whole microservice. In this part, Kmon is deployed in each host, with 4 kinds of indicators collected, which is Network (TCP request latency, drop message), Block I/O state (throughput, count, type, stack), Function counter (system call "write" and "read"), and Off-CPU time (stack, duration). Tbl. V shows some of the results of exported TCP request data, which contains timestamp, latency source, and destination.

The average resource usage for Kmon node in 30 minutes is 9.55% in CPU usage) and 616.11MiB in Memory usage. Tbl. VI shows the influence in response time.:

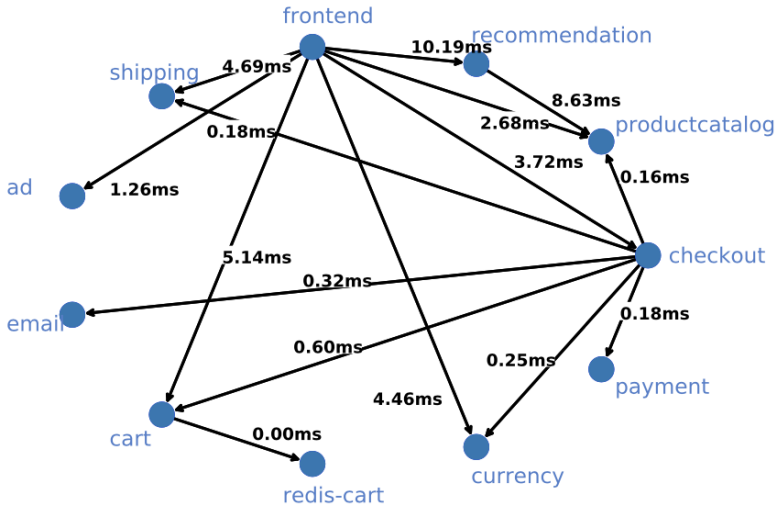

Fig. 11. Topology constructed by indicators collected in 1 minutes.

Fig. 11 shows the topology visualization of requests by network data collected from this part of the experiment. For a more fine-grained indicator in a pod, like “recommendation service”, Fig. 13 shows the state at the beginning of K6 starts to generate load. The program in the pod sleeps longer and more frequently. Combine with Fig. 12, which shows the duration when the program is off-CPU in each type of stack. Sleep (do nano sleep), synchronize (futex wait queue me) and epoll (poll schedule ...) may cause its increasing.

XIII. RELATED WORK

A. Intrusive Monitor Framework

The intrusive monitor frameworks need to change source code or binary file of the user’s programs, which introduces addtional overhead. Pythia [5] focuses on where, what, and when to instrumentation in a distributed application in an au- tomatic way. Seer [6] and MicroRank [9] instrument trace API to microservice applications for request information to match patterns of services. The work [10] increases observability by inserting API hooks into source code for distributed system diagnosis. Compared with the previous works, users can get the detailed monitor information with the help of Komn without any instrumentation.

B. Non-intrusive Monitor Framework

Contemporary in-kernel monitoring tools such as ftrace10 and sysdig [4] can gain fine-grained indicators on a single host, but they are hard to collect and aggregate indicators from all nodes in microservice environment. Microscope [11] captures the network connection information of microsercice systems to monitor the changes of service dependency graphs. Microscaler [12], [13] uses the Service Mesh to monitor metrics of microservice systems. However, Microscope and Microscaler cannot get the system-lever metrics.

Chang et.al., [2] use the Bayesian model to analyze data collecting by eBPF collected by valtrace. The study in [7] uses the random forest model to analyses network-related metrics from eBPF in virtual machines. Both studies are focus on algorithms rather than fitting dynamic environment. Compared with the previous works, Kmon is more suitable for a dynamical microservice environment and can gain fine-grained indicators efficiently.

XIV. CONCLUTION

To create a transparent monitoring system with fine-grained indicators, we introduce Kmon, an eBPF based system. It transparently captures various types of indicators and organizes them in three levels, which is convenient for system operators and algorithms. Experiments show it has low CPU usage and little influence on service response time.

In the future, we aim to find a better way to capture TCP connection information on fewer assumptions. The current assumption of TCP requests is not suitable for some specific type of service like Message Queue. The memory usage of Kmon also needs to be reduced. Two directions are considered. One is to use library libbpf instead of libbcc, which uses less memory in run-time with portability, The other is to find a better representation of indicators to compress their size.

TABLE V
RECORD OF TCP MESSAGE SENDING

time	*service*	*latency (ns)*	*myIP:port*	*peerIP:port*
Jan 15, 2021 @ 21:56:13.206	emailservice	375170	172.20.1.132:8080	172.20.2.134:51614
Jan 15, 2021 @ 21:56:13.200	shippingservice	676425	172.20.1.129:50051	172.20.2.134:50330
Jan 15, 2021 @ 21:56:13.199	cartservice	594035	172.20.1.132:7070	172.20.2.134:53604

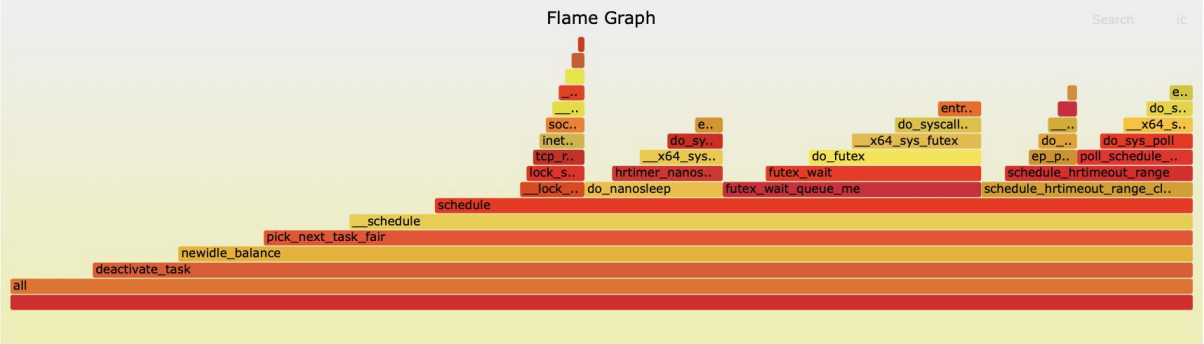

Fig. 12. The Flame graph of off-CPU time of one process. A flame graph can reflect how a program spends its time. Each bar is at random warm colors and ordered alphabetically, the top one shows the duration of off-CPU time, and beneath is its ancestry. The Y-axis of the bar shows stack depth. In this picture, most of the off-CPU time is obtained from function futex (synchronize) and poll (network).

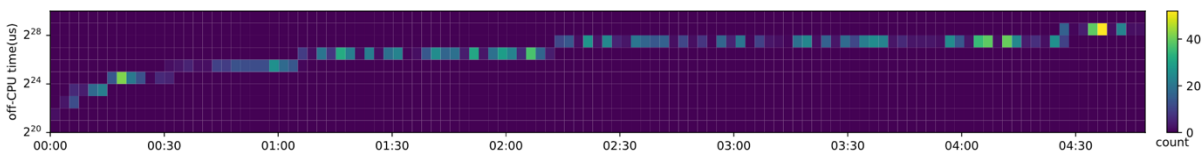

Fig. 13. Heatmap of off-cpu time (only with sleep state) of service “recommendation service” in 4 minutes at the start of load generator running.We can easily notice the increase of off-CPU time.

TABLE VI
RESPONSE INFLUENCE FOR MICROSERVICE IN 30 MINUTES

	With Kmon Node	***Without Kmon Node***
Response-time(Avg)	361.97ms	348.97ms
Response-time(P95)	790.09ms	784.09ms