**基于微服务架构的系统的无侵入监控**

Fa´bio Pina, Jaime Correia, Ricardo Filipe, Filipe Araujo and Jorge Cardoso

CISUC, Dept. of Informatics Engineering University of Coimbra

Coimbra, Portugal

fpina@student.dei.uc.pt, jaimec@dei.uc.pt, rafilipe@dei.uc.pt, filipius@uc.pt, jcardoso@dei.uc.pt

摘要

把大型软件系统拆分成较小的、相互交互的功能模块这样的设计方式正呈现上升的趋势。这样的架构风格被称为“微服务体系架构”，它以增加了复杂度和牺牲可观测性为代价，简化了开发、部署和管理过程。事实上，在大规模的系统中，由于单个模块影响多个模块产生的级联效应，很难准确判断是哪部分微服务模块导致的用户请求时延。单独分析某个组件是无法得到答案的，除非大范围的检测源代码，否则想找到瓶颈和跟踪问题的根本原因是很难做到的。为了解决这个问题，我们提出了一个更简单的方案：记录网关活动，以注册微服务模块之间的调用，因此可以在不改变源代码的前提下实现提取拓扑结构和性能指标。为了验证方案可行性，我们实现了提出的平台，监控着一个正在运行的微服务体系架构的应用。实验结果显示我们可以提取相关的性能信息，几乎可以忽略不计带来的负载，甚至在遗留的系统中也可以，不过这种情况下检测模块的实现是个高消费的任务。

**关键词**：黑箱监控；网关；微服务

**Nonintrusive Monitoring of Microservice-based Systems**

**ABSTRACT**

Breaking large software systems into smaller func- tionally interconnected components is a trend on the rise. This architectural style, known as “microservices”, simplifies develop- ment, deployment and management at the expense of complexity and observability. In fact, in large scale systems, it is particularly difficult to determine the set of microservices responsible for delaying a client’s request, when one module impacts several other microservices in a cascading effect. Components cannot be analyzed in isolation, and without instrumenting their source code extensively, it is difficult to find the bottlenecks and trace their root causes. To mitigate this problem, we propose a much simpler approach: log gateway activity, to register all calls to and between microservices, as well as their responses, thus enabling the extraction of topology and performance metrics, without changing source code. For validation, we implemented the proposed platform, with a microservices-based application that we observe under load. Our results show that we can extract relevant performance information with a negligible effort, even in legacy systems, where instrumenting modules may be a very expensive task.

**Key Words**:Black-box monitoring; Gateway; Microservices

**一、引言**

微服务架构在分布式系统中成为了一种趋势。这种新的模式是由于多种因素得以发展的。首先，标准的单体系统很难维护，部署，开发和扩展。因此，有必要将这些垂直系统分解成一些面向功能的模块，而这些分解出的模块在发展和管理方面可以单独处理。其次，微服务架构更适合在Docker[1]或其他容器中部署和操作。最后，开发产品的方法，例如敏捷开发或DevOps，单独工作的团队越小，越适合使用微服务架构。所以，微服务架构有着巨大的优势在开发，操作，可用性和可伸缩性方面，由此逐渐成为大规模系统的一个标准。

尽管有着上述的好处，但仍然存在着问题需要解决。其中之一就是监控。在传统的单体系统中，监控是局限于系统的，有着稳定的架构而没有弹性。在微服务系统中，管理员必须在成百上千台机器上找出异常的根本原因，这些机器上的服务具有很高的弹性，并且相互通信。复杂性的陡然性增加，给管理员造成了艰巨的任务。

有些监控工具，由于其在标准系统中经过考验的能力，也被微服务架构采用。这些工具，例如Nagios或Zabbix，通常监控多个基础结构指标，例如CPU或内存使用率，并且含有仪表盘来提供系统整体概况，当某些规则或阈值被违反时通知管理员。

其他的监控平台，例如New Relic[2]或者Dynatrace本质上是与编程语言和系统耦合来监控的，但是提供了框架总体的概观。其他有意思的方法有Kibana[4]或Grafana[5]。Kibana主要用来分析日志，而Grafana更多是用来分析和构造系统性能可视化，比如CPU或I/O利用率。

既然几个微服务之间会有交流，一个更强大的监控技术就包含检测所有模块，为一个特定的请求创造可跟踪性。跟踪正常情况下是传播一个相关的标识符，它可以由几个微服务间的传播来确定流图。换句话来说就是跟踪允许系统管理员确定应用的整个工作流程。有一些帮助实现跟踪的框架，如ZipKin[6]，Opentracing[7]和Dapper[8]。尽管跟踪有着一些好处，但它也带来了两个缺点。第一，所有的微服务都必须扩展跟踪功能，并且负责发送数据给中心节点。这个平台收集，处理，整合这些原始数据。因此开发人员不仅需要关注业务算法还必须关注微服务的监控和操作。第二，中心节点可能由于大量的记录而成为系统瓶颈。事实上，跟踪系统通常抛弃较老的数据或仅保存一部分数据。

读完上述的解决方法，你可能会认为管理员操作所有的工具来监控系统。然而事实上操作者使用很多平台和框架，其中一些是从单体系统移植过来的。这些工具仅提供系统中发生的情况的洞察力，承担浏览几个仪表盘和提示信息定位问题所在的重任是管理员的责任。因此，微服务框架引入了一种新的范式开发分布式系统，定义了函数和边界，但仍使用类似于我们在旧体系结构中可以找到的监控技术。

我们这里提出的方法完全是“黑盒”方式，把监控功能从面向功能的微服务框架中解耦。这种方法能够做到既不侵入也不破坏源代码，同时也不需要对微服务进行调整。因此这对于已经投入生产环境的系统是个非常好的方法。为了完成这个目标，我们使用并更改了Netflix公司的Zuul[9]网关模块，来从微服务发出的请求中收集度量。基于例如返回时间，请求源和请求目的地等度量，我们整合了这些原始数据，然后输出一个简洁的相关信息，例如平均回复时间，拓扑结构和全部服务特性描述。

我们的结果显示尽管我们使用的是无侵入式的方法，我们依然可以获得相关且有效的信息给系统管理员。而且不需要检测服务或者向架构中添加代理，只需要微服务模块已经需要的组件。因此，展示的这个方案很有用，特别是对于采用了微服务架构的经常改动和有弹性的系统。

论文剩余部分按如下结构组织。第二章描述我们处理的问题和我们解决它所使用的方法。第三章描述实验设置。第四章展示和评估实验结果，这个方法的好处和它的局限。第五章展示相关的工作。最后，第六章做出总结并提出未来的工作方向。

**二、提出的方法**

在这篇论文中我们处理了监控微服务体系结构的问题。在垂直解决方案中，监控简单一些，因为应用不会随着时间变化很大。微服务是从新的开发模式中逐步形成的，例如敏捷或DevOps和新的部署技术，例如容器。系统监控却不随着这些变化而变化，依然基于单体系统的应用和技术。第五章我们讨论世界上主要的科技公司都是如何困在这个问题上的，他们被迫为他们的需求创建自定义的平台。确实，监控在高度动态变化的系统中是一个很复杂且困难的问题。

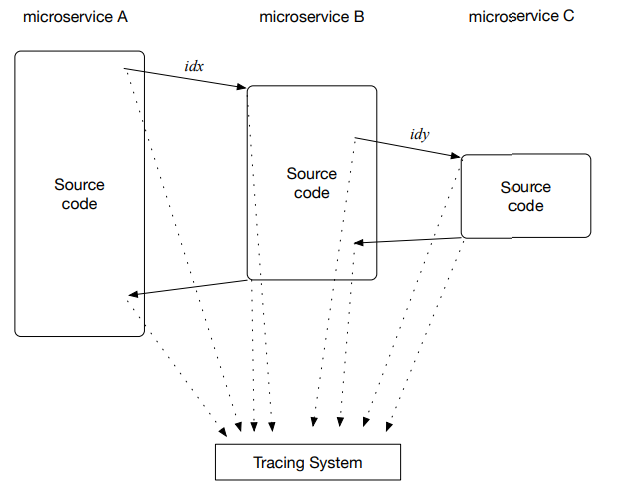


图1 微服务应用追踪图（省略了减少跟踪消息的优化）

我们从不同的角度分析监控问题。一个关于监控的典型方法会包含检测或在尽可能多的层上添加代理，从主机到中间件，一直到应用层上。图1展示了三个微服务的调用序列。A的一些方法调用B中的一个方法，而B中这个方法又调用了C中的一个方法。为了把这种相互依赖的调用关系传递给监控系统，信息必须携带一些标识符来体现关联关系，例如一个带有一些标识符(idx=idy)的HTTP头。不幸的是这需要修改应用源代码。虽然这种技术创建了几个监控点，但它们也是将监控与业务逻辑结合起来而引起额外故障的维护点。这违背了微服务方法，它遵循面向函数的细粒度模块的前提。为了消除这种规范化模块的需要，我们采用了一种非传统的方法。了解到微服务使用了网关来做服务发现和重定向请求，我们添加了收集监控信息的功能到网关中。这个想法是让网关收集信息，例如响应时间，调用方和被调用方的IP和端口，以及被调用函数的身份信息。这种方法有些优点，例如在不影响系统可伸缩性的情况下将监控系统和应用程序分离，因为网关和相关服务是水平可伸缩的。

在下一子段中，我们详细讨论使用的方法。首先，展示整体框架，以及我们如何将解决方法并入网飞的模块中。然后，检查我们在网关中能收集到的度量，并讨论我们可以使用标准工具构建的信息仪表盘。最后，展示我们如何实现并分布此工具。

（一）框架

在微服务框架中，用网关做服务发现是个常见的方法。因此，利用这个模块来观察系统是非常有吸引力的。我们凭借三个网飞的模块，Zuul，Ribbon和Eureka，分别负责网关，负载均衡以及可伸缩服务的注册。这些服务允许我们收集关键路径之外的指标，这是监控微服务系统的优势。在图2中展示了高层的架构。

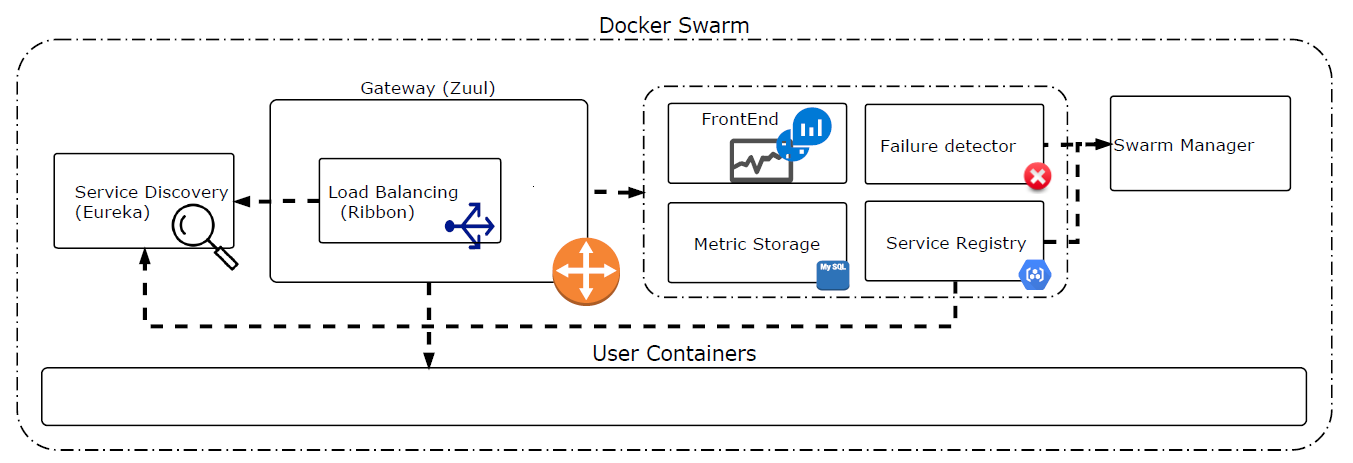


图2 系统组件

我们的方法有四个组件，它们与微服务最佳实践相一致，例如服务发现或容器化。首先，“度量存储”模块收集自定义的Zuul应用程序收集的度量。这些度量包括响应时间，两服务间的请求或直接来自客户端的请求的响应时间，请求源及目标的IP和端口号，还有目标微服务被调用的函数名。这个模块扮演了“前端”模块的后端，这里的“前端”是指展示相关信息例如响应时间，拓扑结构和服务特性的前端。

其他的两个模块与“服务注册”和“错误检测”有关。在本文中，我们主要研究docker容器[1]，考虑到其受欢迎程度。通过分析容器化，容器的数据描述以及容器管理器的工作方式，可以完全自动化注册和故障检测的过程。这给我们提供了很大的好处，因为我们不需要再做容器检测。在这种情况下，当容器中产生变化时，比如创建，销毁或状态改变，将通过于容器管理器关联的代理通知我们的模块。考虑到我们同样观测HTTP结果，实现一个用于错误检测的模块也是可能的。此模块将来自HTTP结果的信息与容器状态结合起来，为我们的自主维护和恢复系统添加功能。当实例失败时，可以删除或重新启动此实例，而不需要管理员管理或微服务检测。还需要指出的是，监控中涉及的组件是水平可伸缩的，因此不会损害应用程序的性能或可用性。由于我们删除了图1所示系统所需的工具，所以负责提取和处理度量的流程在关键路径之外，因此不会产生任何类型的开销。

（二）收集的度量

在表1中我们展示了“度量存储”模块收集的度量数据。对于一个请求，无论是来自另一个微服务还是客户端的，我们都保存请求有关源和目标的度量数据。

除了具有平均值和四份位数如方框图之类的标准图外，这些原始数据同样能创建有关系统的高级信息。例如，可以动态地提取拓扑结构，描绘不同微服务间交互的级别。除此之外，我们也可以计算每个微服务的响应时间和负载，推断每个模块最大承载能力和服务质量，以确保正确的尺寸标注。

我们采用Grafana[5]，一个很灵活可自动化的用于分析和监控的开放平台来做我们的监控系统的前端。为了显示一些更复杂的图，我们使用R语言生成的图形作为补充，这是学术领域的一个通用标准，用于模拟和分析，并结合Grafana上的输出。

表1 收集的度量

|  |  |
| --- | --- |
| 度量 | 类型 |
| 开始时间 | Long |
| 结束时间 | Long |
| 时长 | Long |
| 源IP | String |
| 源端口 | Integer |
| 目标服务 | String |
| 目标实例 | String |
| 目标IP | String |
| 目标函数 | String |

（三）实现

为了验证我们的方法，我们为docker集群容器管理平台做了一个完全无侵入式的实现。源码和部署指南都作为开放资源在Github[10]上可访问到。另外，它同样包含在第三章中我们用来做实验验证的微服务应用。在安装有Docker和集群管理器的系统中此工具很容易部署。由于监控工具需要一个覆盖网络[11]，系统必须创建和配置这个网络，以确保我们的方法正确运行。

之后，唯一的参数化需要就是覆盖网络的名字。其余参数可以用默认值定义，而不会使功能失效。为了使用我们的监控解决方案，人们可以下载仓库，在配置文件中定义覆盖网络，运行将自动生成和部署Docker撰写清单文件的安装脚本。在第二章A小节中描述的服务注册组件，会订阅docker事件API，在容器创建，销毁和状态发生变化时会收到提醒。因此网关上的服务注册将自动完成，不需要服务本身的协作。这是可能的，因为每个容器已经携带了相关的元数据，比如名字和服务端口。

这个监控解决方案包括一个用Grafana套件开发的用户自定义前端模块。除此之外，我们开发并包含了一个用R语言编写的自定义插件，以生成更复杂的可视化效果，如弦图。我们的原始数据存储模块使用influxDB和MySQL数据库。在表2中，我们展示了与该工具关联（和部署）的整体容器。一旦安装到Docker集群容器管理器上，其上部署的所有其他应用都会自动的使用我们的网关进行服务发现和监控，只要它们位于同一个覆盖网络上。

表2 工具容器

|  |  |
| --- | --- |
| 容器 | 描述 |
| Eureka | 服务发现 |
| Zuul | 网关 |
| Service Registry | 与Eureka联合管理容器生命周期 |
| InfluxDB | 时间序列可扩展数据库 |
| MySql | 数据库 |
| Grafana | 前端 |
| Chord Plugin | 生成可视化弦线图 |

图3显示了在实验阶段提取的示例仪表板。在这个例子中，我们展示第四章中描述的图表，如方框图和弦图[12]。

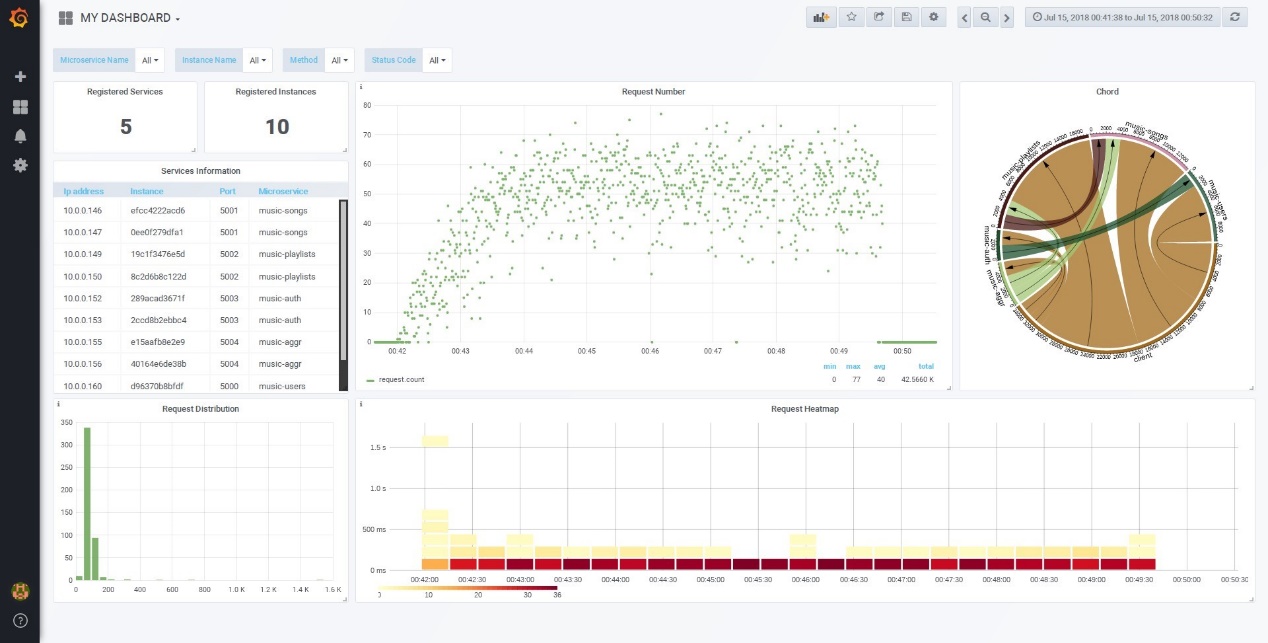


图3 用户自定义前端样例

**三、实验设置**

在这一章中，我们展示使用的实验设置以及对Netflix模块、我们的微服务和应用做出的改动。首先，关于测试应用使用的基础设施模块，我们使用稳定的模块。为了负载平衡，我们使用Ribbon。这个模块给我们提供了一些好处，比如可用负载均衡算法，REST风格接口的使用，但是最重要地是与Netflix剩余支持模块的现成集成。因此，与发现和注册模块（Eureka）进行了集成，允许更灵活地实例化和实现我们的方法。我们同样使用了Netflix的模块Zuul配合Ribbon和Eureka，它内部使用了Ribbon。Zuul通过查询到Eureka获取服务地址，然后将请求路由到正确的服务。由于request请求都必须经过Zuul，因此这个模块允许我们对微服务之间的流量有一个清晰的了解，并将监控信息收集到一个中心点。

我们实验设置的另一个组件是允许我们测试监控方法的应用程序。我们实现的应用程序与音乐相关，有五个功能定义明确的微服务。该应用程序允许其客户机管理用户、播放列表和歌曲。在表3中，我们明确了与每个微服务相关的整体端点，各自的调用方法和简短的描述。

表3 微服务和可用的函数

|  |  |  |  |
| --- | --- | --- | --- |
| **微服务** | **功能** | **请求类型** | **描述** |
| Authentication MS | / | GET | 系统健康检查 |
| /login | POST | 验证用户凭据并创建令牌 |
| Users MS | / | GET | 系统健康检查 |
| /login | POST | 验证用户凭据 |
| /users | GET | 获取用户信息 |
| /users | POST | 创建用户 |
| /users/*{*id*}* | DELETE | 删除用户 |
| /users/*{*id*}* | PUT | 更新用户 |
| Playlists MS | / | GET | 系统健康检查 |
| /playlists | GET | 获取与用户关联的播放列表 |
| /playlists | POST | 创建播放列表 |
| /playlists/*{*id*}* | GET | 获取播放列表 |
| /playlists/*{*id*}* | PUT | 更新播放列表 |
| /playlists/songs/*{*id*}* | DELETE | 从播放列表中删除某个音乐 |
| /playlists/songs/*{*id*}* | GET | 获取与播放列表关联的音乐信息 |
| /playlists/songs/*{*id*}* | POST | 向列表中添加音乐 |
| Songs MS | / | GET | 系统健康检查 |
| /songs | GET | 获取音乐信息 |
| /songs | POST | 创建音乐信息 |
| /songs/convert/*{*id*}* | GET | 将音乐从MP3扩展转换为wav |
| /songs/criteria | GET | 根据某些条件获取音乐列表 |
| /songs/*{*id*}* | DELETE | 删除音乐 |
| /songs/*{*id*}* | PUT | 更新音乐 |
| Aggregator MS | / | GET | 系统健康检查 |
| /playlists/songs/*{*id*}* | GET | 获取与播放列表关联的所有音乐信息 |

由于我们想要收集关于请求的原始数据，但又不修改微服务代码，所以我们修改Zuul的源代码来注册关于每个请求源和目标的信息。我们保存如下的信息：发出请求的微服务，开始时间，结束时间，请求源的IP和端口，处理请求的微服务实例，以及调用的方法。有了这些信息，我们能够提取出系统相关信息，比如拓扑结构，平均响应时间，并按微服务和函数进行细化。如上所述，我们并不需要修改应用的任何源码（即我们只改变基础架构）。然后，对原始数据进行预处理，并将其重定向到MySQL数据库，该数据库是我们“系统度量”模块的一部分。

该软件安装在运行Ubuntu 16.04的虚拟机上。虚拟机有8个vcore，22G内存。除Zuul参数敏感头外，所有组件均采用标准参数化安装。这样的配置允许我们能够在所有的微服务间传递身份验证令牌，而不需要操纵网关。

为了模拟系统负载，我们使用Apache JMeter[13]。我们配置这个负载工具10个线程，启动周期为120秒。每个线程运行10分钟，循环如下：1）创建用户；2）身份验证；3）获取用户；4）更新用户；5）添加歌曲；6）获取歌曲；7）更新歌曲；8）转换歌曲；9）添加演奏者；10）获取表演列表；11）更新表演列表；12）向列表中添加音乐；13）从列表中获取音乐；14）从获取列表中所有音乐；15）从列表中删除音乐；16）删除列表；17）删除歌曲；18）删除用户。

在表4中，我们展示了本实验中使用的开源组件以及其各自的版本。

表4 使用的软件

|  |  |  |
| --- | --- | --- |
| 组件 | 作用 | 版本 |
| Zuul | 网关 | 1*.*4*.*4 |
| Eureka | 服务发现 | 1*.*4*.*4 |
| Ribbon | 负载均衡器 | 1*.*4*.*4 |
| MariaDB | 微服务使用的数据库 | 10*.*3*.*7 |
| MySQL | 前端使用的数据库 | 8.0 |
| JMeter | 负载测试工具 | 4.0 |

我们这个实验的最终目标很简单：了解我们的“黑盒”式非入侵监控系统的局限，优点和劣势。图4总结了整个系统，包括应用，基础设施和监控工具以及负载生成器。

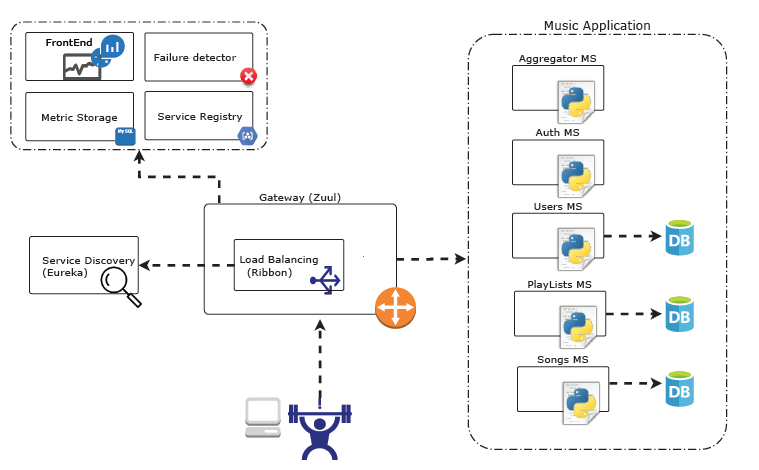


图4 系统架构

**四、结果**

在本章，我们展示了从API网关收集监控数据的结果。此技术允许我们从微服务交互中提取原始数据，从而为管理员创建一组带有相关信息的度量和表格，而不需要在主机层面使用工具或代理。在本文中，我们提出了5个可视化组合，给我们一个清晰的系统大观。

关于前端应用，我们将可视化划分为3个不同的图表。首先我们需要知道微服务在响应时间上有着很高的不同性。为了获得这些数据（参见图5），我们选择了方框图。这种图表允许我们在一个视图中比较信息。图5基于从MySQL数据库提取的数据创建。值得一提的是，虽然我们呈现的是微服务的响应时间分布，但是用户可以向下钻取，并在每个微服务内按目标函数查看相同分布下的可视化结果。

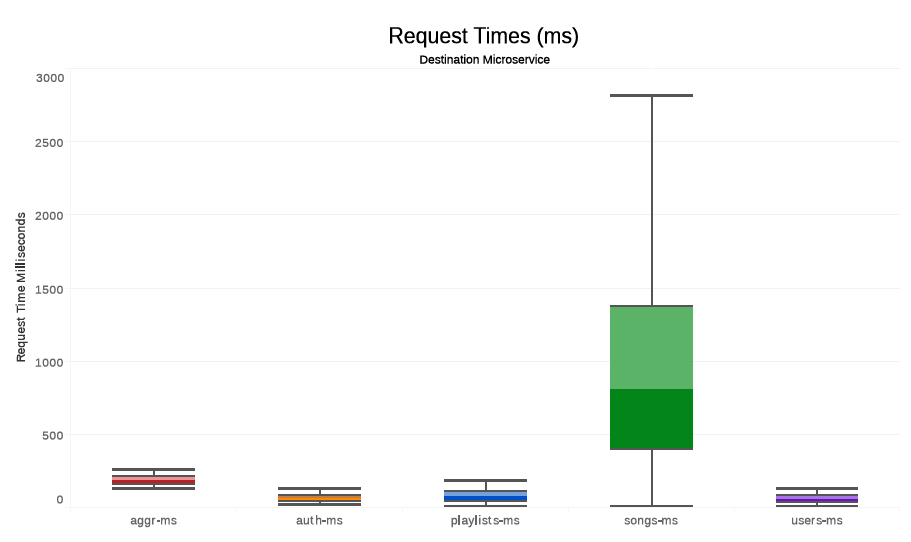


图5 按微服务划分的响应时间方框图

关于微服务之间的依赖性，我们采用一个图来展示。这种表示使我们能够表达模块之间的拓扑结构和依赖关系。在图6中，我们可以很轻松的看出不同微服务间的关系，和客户端的直接访问。

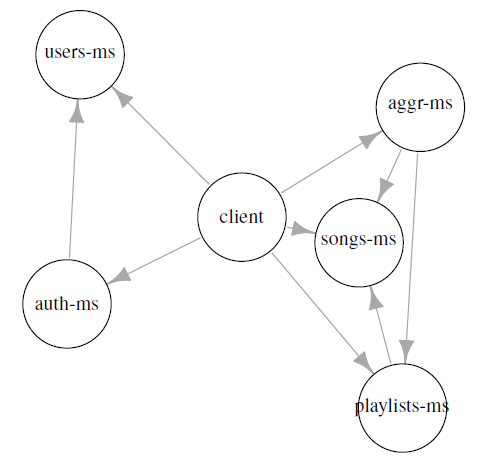


图6 应用组成图

尽管每个微服务（和功能）的响应时间分布在方框图中，微服务之间的依赖关系展示在图形可视化中，我们仍然缺少一个关键方面来理解系统的健康状况：微服务和函数在系统中的重要性。为了实现这个目标，我们借助于Gu等人的工作[12]，绘制了弦图。这种图允许我们能够看到不同实体间更复杂的关系。图节点排布成一个圆，其相互作用的重要性与连接弧的宽度成正比。我们使用箭头表现哪端接受调用，并使用颜色来简化理解。例如，在图7a中，我们可以看到请求的数量，在图7b中看到等待时间。在一个非常大的系统中，包含所有内容的弦图很难读懂。因此，为了提高图表表示力，管理员可以选择显示哪儿微服务，如图3所示。例如，在图7c中，我们看到延迟，但没有客户端发布的请求，因为这些请求会对图形产生巨大的影响，并使其他交互操作不那么可见。

考虑图片7a，管理员知道微服务playlist-ms是28000个请求的来源或目标。这些请求中，有大约3000个请求是从playlist-ms到songs-ms，约21000个请求直接由客户端提出，约3000个源自aggr-ms微服务。这样，我们就可以看到playlist-ms微服务在整个系统中的相关度。另外，同样的方法也可以用来分析延时。结合依赖关系图和弦图的方框图给我们很好的提供了系统性能，模块重要性以及微服务或函数的响应时间分布。

|  |  |  |
| --- | --- | --- |
|  |  |  |
| a）频率 | b）延时 | c）不含客户端的延时 |

图7 弦线图

系统管理员与监控系统的交互可以这样进行：管理员首先查看图5，方框图，它提供了哪些服务具有更长响应时间的清晰视图。很容易观测到服务songs-ms在所有的模块中具有最长的响应时间。拥有第二响应时长的模块是aggr-ms。然而管理员可能查看songs-ms微服务在整个系统中的重要性。尽管它较其他微服务具有更长的响应时间，管理员依然应该查看剩下的图片。在图6中，他可以看到songs-ms直接从客户端、aggr-ms以及playlists-ms接受调用，因此songs-ms微服务有着很大的系统依赖。此外，我们注意到songs-ms的延时主要取决于谁在调用它，客户端发起的调用有着更高的延时。这将表明，要么它们在调用不同的函数，要么存在一些异常。然后，管理员（使用我们的应用程序）可以向下钻取并查看粒度调用数据以进行进一步分析。

管理员最后需要的信息是微服务间的请求数量以及延时。尽管songs-ms有着很长的响应时间，就依赖关系来说是个关键模块，我们依然需要了解通过songs-ms的请求数量是否与系统处理的请求总数相关。我们可以看图片7a和7c来获取这个信息。我们可以看到songs-ms是请求的一个重要终点，特别是来自客户端和aggr-ms的。事实上看图7c，我们可以看到aggr-ms的依赖（songs-ms和playlists-ms）延时较低，所以它们不是瓶颈。

考虑到图7a显示aggr-ms服务提出的调用大约是它被调用的两倍，因此可以得出这样的结论：延迟是多个请求造成的，可能是串行的，也有可能是并行性较低的。因此，看完这些视图的管理员有两种解决方案：按函数深挖songs-ms的方框图，检查是否有拖后腿的函数，并/或者提高aggr-ms调用依赖的方式。重要的是要记住，所有这些信息都是可以实现的，基础设施中没有工具或代理。

当拿我们的方法和现在为微服务开发的监控工具比较时，就能看到一些好处还有一些缺点。其中一个缺点是与跟踪相关的。我们没有跟踪提供的精细粒度来理解某个请求的工作流。因此我们可能错过一些关于微服务间因果关系的信息。然而，如果我们有广泛的请求分布，我们仍然可以估计工作流。另一方面，我们的模块入侵性要小得多，因为它没有在系统中开发工具或部署代理的开销。此外，我们的解决方案可以以非常灵活的方式在遗留系统中实现，这可能是基于跟踪的解决方案无法实现的。

**五、相关的工作**

由于我们的工作与不同的研究领域间接相关，因此是一个非常活跃的话题，所以我们把相关工作分成行业和学术两种解决方案。此外，我们还介绍了一些工作，尽管与我们的工作没有直接关系，但它们是互补的。

首先，关于行业方法，我们有来自Netflix的软件。Netflix有几个模块用于监控和检测。Vector[14],是一个框架，允许创建带有度量的仪表盘，如CPU或网络利用率。该模块强制在每个主机或应用程序上存在一个名为PerformanceCoplot（PCP）的代理进行监视。另一个系统是Atlas[15]。这个平台专注于大数据和时间序列。其目的是应用预测方法，了解度量的演变和实时分析。虽然功能强大，但这个平台需要对微服务进行检测。另一个很相似的方法是Prometheus[16]，一个同样需要检测微服务的开源监控解决方案。

基于检测或代理的应用程序性能监视（APM）工具允许在违反某个阈值时创建仪表板和向管理员定义通知。例如，Dynatrace[3]和其他[2]、[4]、[5]、[17]具有一些与微服务基础设施相关的功能。然而这些工具主要集中在显示信息上，与我们相比，它们更不关心侵入性。Spotify使用与Netflix类似的方法。他们需要一个定制的监控基础设施来创建仪表盘和时间序列。再次地，每台机器需要运行一个代理，将信息发送到一个中心节点[18]。此外，还有一些开源项目，如iovisor[19]，用于检测数千个虚拟机中的性能问题。

视角转移到学术上的贡献，在文献[20]中，作者对如何构建和监控微服务平台给出了一些指导意见，以可以使用工具或代理从主机和应用程序收集信息为假设前提。在[21]中，作者呈现了一个监控仪表板，但还是基于代理和服务检测。

在[22]中，作者用了一种特殊的方法，其中每个微服务都负责其自身的弹性和可伸缩性。模块保存有关CPU利用率和响应时间的信息。尽管这有一些好处，但作者并没有关注组件交互时可能发生的“多米诺效应”或连锁反应。

在[23]中，作者提出了一种基于容器创建“监控作为一种服务”的方法，其中代理与微服务容器相关联。在这个体系结构中，代理和容器之间存在一对一的关系，这可能会导致一些开销和可伸缩性问题。此外，监控与容器关联，与模块之间存在的工作流无关。在[24]中，更改每个容器的访问点以监视网络。因此，它有一种“中间人”的方法，不考虑应用。

我们的方法在至少两个方面与这些过去的方法完全不同。前面提到的一些工具旨在创建仪表盘和通知，前提是系统中有可用的代理或跟踪。这种工具并不考虑减少系统入侵性，而仅仅关注提供可视化界面和数据展示。其他方法对系统进行检测，创建与容器、微服务或网络耦合的框架，但它们不提供有关应用程序工作流的信息。

跟踪，这类最常用的方法之一，提供了理解特定请求流的能力。然而，也存在一些缺点：开发人员必须将每个微服务可检测化而不仅仅是写业务代码，还有监控的需要。不幸的是，这将源代码与截然不同的目标合并在一起。

与以往的工作不同，我们的方法是由简单性驱动的。它提供了监视系统的能力，而不需要代理，检测或开发开销。然而，在数据可视化方面取得的成果令人印象深刻，使系统管理员能够以最小的努力掌握系统的关键方面。

**六、结论与未来工作**

对管理员来说，监控和操作分布式系统是一项困难的任务。而由于微服务技术的弹性和动态性，这项任务变得比以往任何时候都更加复杂。大多数监控解决方案都是为较旧的体系结构设计的，因此缺乏对新范式的任何考虑。

在本文中，我们提出了监控的新方法，而不需要改变系统任何一部分或部署探针。我们的目标是分析“黑盒”方法的局限性，只使用已经部署在微服务体系结构中的一些基础结构模块。我们使用Netflix模块定制网关，从微服务调用中收集原始数据。结果表明，我们的解决方案只需要最少的配置工作就可以集成到系统中，并向管理员生成相关信息。

至于今后的工作，我们希望在几个方向进一步研究。首先，我们希望在自动化分析方面改进我们的开源工具，例如关键路径枚举和异常检测，以向管理员提供有关高层次的更多信息。最后我们认为生成能够预测系统和微服务能力的模型将非常有助于管理员确定维度和保证SLA（Service-Level Agreement：服务等级协议）。

**致谢**

这项工作部分是在PTDC/EEI-ESS/1189/2014-非程序员数据科学项目下进行的，由Compete 2020、Portugal 2020-POCI、UE-Feder和FTC提供支持。

我们还想感谢INCD-Infraestrutura Nacional de Computac¸ ˜ao Distribu´ıda,提供对其计算资源的访问。

**参考文献**

1. Docker. https://www.docker.com/what-docker. Retrieved June, 2018.
2. New Relic. https://newrelic.com. Retrieved May, 2018.
3. Dynatrace. https://www.dynatrace.com/platform/. Retrieved May, 2018.
4. Kibana. https://www.elastic.co/products/kibana/. Retrieved May, 2018.
5. Grafana. https://grafana.com/. Retrieved May, 2018.
6. Zipkin. http://zipkin.io/. Retrieved June, 2018.
7. Opentracing. http://opentracing.io/. Retrieved May, 2018.
8. Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephen- son, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Techni- cal report, Google, Inc, 2010.
9. Zuul. https://github.com/Netflix/zuul. Retrieved May, 2018.
10. Github – monitoring ms. https://github.com/fabiopina/monitoring ms. Retrieved June, 2018.
11. Docker overlay network. https://docs.docker.com/network/overlay/. Re- trieved June, 2018.
12. Zuguang Gu, Lei Gu, Roland Eils, Matthias Schlesner, and Benedikt Brors. circlize implements and enhances circular visualization in r. Bioinformatics, 30(19):2811–2812, 2014.
13. Papers — Apache JMeterTM . http://jmeter.apache.org/. Retrieved: May, 2018.
14. Vector. https://github.com/Netflix/vector. Retrieved May, 2018.
15. Atlas. https://github.com/Netflix/atlas. Retrieved May, 2018.
16. Prometheus. https://prometheus.io/. Retrieved June, 2018.
17. Appdynamics. https://www.appdynamics.com. Retrieved May, 2018.
18. Spotify. https://labs.spotify.com/2015/11/17/monitoring-at-spotify- introducing-heroic/. Retrieved June, 2018.
19. Iovisor. https://www.iovisor.org/. Retrieved May, 2018.
20. S. Haselbo¨ck and R. Weinreich. Decision guidance models for microser- vice monitoring. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pages 54–61, April 2017.
21. B. Mayer and R. Weinreich. A dashboard for microservice monitoring and management. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pages 66–69, April 2017.
22. Giovanni Toffetti, Sandro Brunner, Martin Blo¨chlinger, Florian Dudouet, and Andrew Edmonds. An architecture for self-managing microservices. In Proceedings of the 1st International Workshop on Automated Incident Management in Cloud, AIMC ’15, pages 19–24, New York, NY, USA, 2015. ACM.
23. Augusto Ciuffoletti. Automated deployment of a microservice-based monitoring infrastructure. Procedia Computer Science, 68:163 – 172, 2015. 1st International Conference on Cloud Forward: From Distributed to Complete Computing.
24. F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu. Conmon: An automated container based network performance monitoring system. In 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pages 54–62, May 2017.