中国科学: 信息科学 2024 年 第 54 卷 第 3 期: 582–602
SCIENTIA SINICA Informationis
论文

一种基于动态可寻址会话的服务器无感知计算

李子俊, 赵一龙, 陈全*, 过敏意*

上海交通大学电子信息与电气工程学院新兴并行计算中心, 上海 200240
* 通信作者. E-mail: chen-quan@cs.sjtu.edu.cn, guo-my@cs.sjtu.edu.cn

收稿日期: 2023–05–29; 修回日期: 2023–07–21; 接受日期: 2023–08–23; 网络出版日期: 2024–03–11

国家自然科学基金 (批准号: 62232011, 62022057) 和上海市国际科技合作项目 (批准号: 21510713600) 资助

摘要 服务器无感知计算作为云原生范式中快速发展的新兴技术, 因其按需付费、自动资源伸缩和底层环境屏蔽等特点而受到越来越多的开发人员欢迎. FaaS (函数即服务) 作为 Serverless 架构的主要实现方式, 以函数粒度对应用进行解耦和执行. 大多数云服务提供商也为应用开发人员提供了基于 Serverless 架构的应用搭建服务, 这些服务允许开发人员以函数的形式部署代码, 并根据实际的请求量进行自动扩缩容. 然而, 在部署有状态函数时, 由于 Serverless 架构的无状态特性, 管理其中的有状态数据变得复杂, 往往无法满足 Serverless 中函数对有状态数据的访问性能要求. 因此, 本文提出了一种基于有状态和动态可寻址会话机制的服务器无感知计算系统 XFaaS, 实现了低开销的有状态数据访问和更高的应用吞吐. 实验结果表明, 通过采用 XFaaS 系统部署有状态函数的方式, 可以降低有状态数据访问时延 3 个数量级, 并提高 2 倍以上的函数最大吞吐量.

关键词 服务器无感知计算, 函数即服务, 有状态函数, 粘滞会话, 容器

1 引言

传统的基础设施即服务 (infrastructure-as-a-service, IaaS) 部署模式需要长期运行的服务器来提供持续的服务交付. 然而, 这种传统部署模式导致数据中心资源利用率很低, 尤其对于具有昼夜模式负载变化的在线服务而言, 整体平均每天的资源利用率仅约为 10%. 无论应用是否正在提供服务, 这种独占式资源分配模式都要求持续占用资源. 这个问题促使了平台管理和按需服务模式的发展, 以满足对更高资源利用率和更低云计算成本的需求. 在服务器无感知计算 (Serverless computing) 的概念中, 应用被拆分为函数 (或函数级别的微服务), 遵循函数即服务 (function-as-a-service, FaaS) 模型, 也被称为 Lambda 范式 [1~3]. 其背后的思想是开发者只需关注应用程序代码的编写, 而无需关注后端服务器的运维、部署和扩展等方面. 由于 Serverless 架构基于实际处理的应用请求进行计费, 当请求负载增

引用格式: 李子俊, 赵一龙, 陈全, 等. 一种基于动态可寻址会话的服务器无感知计算. 中国科学: 信息科学, 2024, 54: 582–602, doi: 10.1360/SSI-2023-0155
Li Z J, Zhao Y L, Chen Q, et al. Serverless computing based on dynamic-addressable session (in Chinese). Sci Sin Inform, 2024, 54: 582–602, doi: 10.1360/SSI-2023-0155

加时, 会动态地进行水平资源扩展, 创建更多的容器来处理请求, 而不是像 IaaS 中预先分配资源的实例进行计费. 这使得它能够为应用开发者提供更便捷和低成本的服务构建和部署 [4~6].

然而, 现有 Serverless 架构并不是一种通用的计算模型, 通常只能处理一些与状态无关的应用, 如 HTTP 请求、图片处理和视频转码等. 对于一些有状态的应用, 它们需要持续地维护有状态数据在程序或实例内存中, 函数可以通过逻辑地址直接找到数据并以变量形式进行访问. 现有的 Serverless 框架并不支持上述有状态应用构建的需求, 其中最主要的限制是 Serverless 中的函数请求是不可寻址的. 函数请求的可寻址性指的是: 任何时刻的函数都可以通过一个固定且不可变的逻辑地址来访问特定的数据, 我们称之为 Serverless 的静态可寻址性. 然而, 在 Serverless 中实现静态可寻址性面临两个限制. 首先, 当负载增加时, 水平扩展会生成多个容器, 负载均衡策略会导致某个数据可能在任意容器中生成, 这使得用户函数无法确定某个数据的确切生成地址, 也无法建立固定且不可变的数据通路. 其次, 即使应用控制了请求负载以避免容器的水平扩展, 并且知道单个容器的服务地址, Serverless 的保活策略仍会在容器空闲时自动回收资源, 包括分配的临时服务地址和建立的数据通路. 这意味着每次函数冷启动时, 生成的容器都会分配新的服务地址, 使得用户函数无法保持与上一次请求相同的数据寻址地址. 上述的两个限制中, 后者是无法避免的. 因为 Serverless 中的用户函数对底层云基础设施是无感知的, 云供应商会根据各个节点的资源利用率等指标为冷启动函数选择最合适的物理分布.

现有 Serverless 平台提供的一些用于构建有状态应用的解决方案主要分为两类: 其中一种方案是利用有状态 Serverless 工作流 (stateful Serverless workflow) 来构建有状态应用 (如 AWS step functions[1)] 和 Azure durable functions[2)]). 然而, 这种方案的数据大小限制为 256 kB, 与 Lambda 对函数输入的大小限制相同, 无法满足许多实际应用中对函数间数据传输的需求. 非规划地提高这部分配额上限不仅会增加云商的内存成本, 而且会使得 step functions 调用 TCP 连接时向宿主机引入更多的 CPU 和网络资源开销. 在目前主流的 Serverless 平台所给出的最佳实践中, 图片检测[3)4)]、视频转码[5)]、文本处理[6)]等应用工作流中函数间的传输数据量由于均超过了上述 256 kB 的限制, 从而采用了另一种方案, 即通过远程存储访问关键状态数据. 如果一个函数请求需要将输出数据传递到后续的自身请求或其他函数请求中, 就需要将有状态数据存储在后端存储中 (例如, 远程数据库), 然后后续请求重新下载和加载到内存中 [7~10]. 然而, 由于网络传输开销的影响, 服务的请求端到端时延较高. 上述方案的限制本质上在于无法解决 Serverless 中函数的不可寻址性. 因此, 数据的存取存在中间件管理开销或多副本的高同步开销, 无法提供可靠的点对点数据通路, 从而无法满足实际应用的需求. 然而, 如果我们能够在负载增加时, 保证容器资源的自动伸缩不影响函数数据的生成分布, 就可以实现在应用容器存活期间, 函数可以持续地通过一个固定且不可变的地址对某个数据进行访问. 我们将其称为 Serverless 的动态可寻址性.

本文旨在基于动态可寻址性的特点上, 设计一种面向有状态函数高效执行的服务器无感知计算系统, 以解决原生 Serverless 架构对无状态应用构建的限制. 为达成上述目标, 所利用到的一个关键机制

1) AWS step functions: assemble functions into business-critical applications. https://aws.amazon.com/step-functions/.

2) Azure durable functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/.

3) Google cloud functions — detect and blur offensive images. https://cloud.google.com/functions/docs/tutorials/imagemagick.

4) Google cloud functions — optical character recognition (OCR). https://cloud.google.com/functions/docs/tutorials/ocr.

5) Alibaba function compute — process audio and video files using ffmpeg. https://www.alibabacloud.com/help/doc-detail/146712.htm?spm=a2c63.l28256.b99.313.5c293c94dPLJV1.

6) Serverless reference architecture for real-time file processing. https://github.com/aws-samples/lambda-refarch-fileprocessing.

是粘滞会话 (sticky Session [11]). 通过在 Serverless 中引入粘滞会话机制, 可以确保来自同一 Session 的函数请求始终被路由到同一个容器中执行, 并且可以通过该容器的地址唯一检索生成的各种数据和状态. 然而, 要实现上述目标, 需要克服 3 个关键挑战: (1) Serverless 场景下大多为小规格容器, CPU 时间片和内存资源分配粒度更细 (如 0.1 核和 128 MB). 若允许 Serverless 容器内请求并发, 则请求的响应时延会因为争抢 CPU 导致严重波动, 或者容器内存被超额申请导致 OOM (out-of-memory). 因此 Serverless 容器通常不允许内部请求以多线程方式并发执行 [12~14]7), 当函数面临高并发请求时, 需要设计容器资源的垂直扩展和请求隔离机制, 以确保每个 Session 内的所有请求都能被正确执行. (2) Session 内缺乏对有状态数据进行多个请求的读写操作的机制, 需要在运行时层为其提供有状态数据的访问接口和原子化事务抽象, 保证数据访问强一致性和弱一致性需求. 同时, 由于同时引入了 Session 内的垂直扩展和 Session 间的水平扩展, 数据操作的锁机制需要仔细设计以最小化同步开销. (3) 需要设计基于 Session 的 Serverless 调度机制, 以管理 Session 容器和内部有状态数据的生命周期. 这需要在容器编排框架中引入 Session 感知的容器和数据调度能力, 以实现高系统扩展性和高数据可用性.

我们因此提出了 XFaaS 系统, 旨在为 Serverless 系统提供一个面向有状态函数的系统架构, 以解决传统 Serverless 架构中难以处理有状态数据和复杂逻辑的问题, 并提高应用访问数据和执行效率. XFaaS 系统采用有状态和动态可寻址的 Session 机制来隔离函数请求之间的状态, 并允许对 Session 容器进行水平和垂直扩展, 以适应不同的负载需求. 在 XFaaS 系统中, 每个函数都拥有一个专用的函数调度器, 用于管理函数内部的所有 Session 容器, 并根据请求中的 Session 提示信息 (Session hint) 将请求调度到相应的 Session 容器中执行. 当高并发请求访问 Session 容器中的有状态数据时, Session 容器可以进行垂直扩展, 为每个请求提供独立的资源隔离环境. XFaaS 系统还提供了规范化的数据调用接口和原子化事务的抽象, 以确保多个请求对同一数据操作时的一致性. Session 代理进程 (Session proxy) 支持跨多个函数的原子化事务, 在同一 Session 中进行操作. 此外, 基于 Session hint 的函数调度策略在保护应用的隐私的前提下, 实现了 Session 感知的函数请求标识和调度. 本文的主要贡献总结如下.

• 在 Serverless 语义下为 Session 容器提供了垂直扩展机制. 利用 WebAssembly (简称 wasm) 作为安全隔离和资源分配的基本单位, 在容器内部允许请求并发执行.

• 在基于双向 (水平和垂直) 资源伸缩模型下为 Session 内访问有状态数据引入了对应的一致性保证. XFaaS 面向用户编程封装了统一访问接口, 并提供原子化事务抽象, 实现了低开销数据存取.

• 基于动态可寻址性构建了面向有状态函数的 Serverless 系统 XFaaS, 以及配套的基于 Session hint 的容器恢复、Bin-packing 调度、资源回收和 Session 间通讯机制.

XFaaS 通过双向资源扩展的支持, 实现了更加灵活的资源调度, 为用户提供了方便易用的有状态函数编程接口, 同时确保数据的一致性和高可用性. 实验结果表明, XFaaS 中基于 Session 的有状态函数部署能够将数据访问时延降低 3 个数量级. 在高并发场景下, 相较于现有的部署方式, 访问有状态数据的时延降低了近 2 倍, 并且应用的峰值吞吐量提高了 3 倍以上.

2 背景和动机

本节将评估现有 Serverless 部署有状态应用的性能, 并通过比较其与原生内存访问的性能差异, 探

7) AWS Lambda execution context. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html.

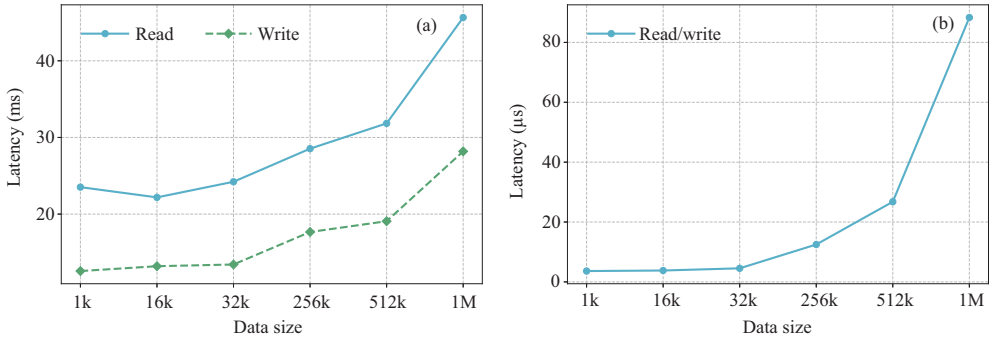

图 1 (网络版彩图) 传统基于 OSS 的有状态数据访问和原生基于本地内存的有状态数据访问速度对比

Figure 1 (Color online) Comparison between traditional OSS-based and native memory-based stateful data access. (a) OSS; (b) memory

讨使用 Session 机制和垂直并发来提高有状态应用的执行效率的潜力.

2.1 现有 Serverless 有状态应用读写性能

鉴于现有的 Serverless 平台无法原生支持有状态函数的部署, 采用远端数据库存储有状态数据已经成为一种常见的解决方案. 在这种基于中间件的架构中, Serverless 应用需要通过中间件获取有状态数据, 而传统的有状态函数则可以直接访问内存变量. 本小节旨在比较基于中间件的方案 (如阿里云 OSS) 和传统的有状态函数在读写操作性能方面的差异. 我们使用不同大小的数据作为输入, 记录读写操作的平均延迟, 以评估它们的性能表现. 实验环境配置遵照 4.1 小节中的配置.

如图 1 所示, 当直接从内存访问有状态数据时, 小型数据仅需约 3000 ns 完成一次读写操作. 然而, 通过网络访问存储在 OSS 中的有状态数据时, 读写时延分别增加到 10 ms 以上, 导致性能下降了 4 个数量级. 此外, 随着数据量增加至 1 MB 以上, 相对于内存访问, 通过 OSS 网络访问有状态数据的性能损失更加显著, 数据访问时延显著增加.

因此, 使用 OSS 等中间件方式访问有状态数据构建 Serverless 有状态应用并不理想. 不仅性能明显下降, 而且在数据量动态变化时, 数据访问时延也会出现明显波动 [15]. 如果能够以直接访存的方式实现有状态数据访问, 将能够避免上述问题, 并最大化提高有状态函数的运行效率. 这种原生支持的方式可以提供更高的性能和稳定性, 从而满足实时、高吞吐量的有状态应用的需求.

2.2 Serverless 中 Session 的并发请求处理性能

考虑到在部署 Serverless 有状态函数时, 使用远端数据库存储有状态数据会带来昂贵的时间成本, 我们希望 Serverless 系统能够原生地支持从本地内存访问有状态数据的模式. 为了实现这一模式, 我们需要确保访问相同有状态数据的请求始终被路由到同一个容器中, 以避免由于请求的不可寻址性而导致无法定位有状态数据的地址, 从而使生成的各种数据和状态能够通过该容器的唯一地址进行检索. 这种请求路由模式也被称为粘滞会话 (sticky Session).

为了模拟引入 Session 的请求执行过程, 我们预先将一份数据加载到容器的内存中. 当请求被路由到该容器时, 可以进行简单的内存数据拷贝操作来操作该数据. 通过生成连续的请求并使用负载均衡器将请求路由到上述容器中, 多个请求对同一份数据的所有操作都可以在该容器中进行, 以模拟本地内存访问有状态数据的情况. 为了评估单个 Session 在处理请求时的性能, 我们记录了在实验中有不同并发请求到达单个 Session 时的数据访问时延. 然而, 由于 Serverless 容器通常不允许容器内并发

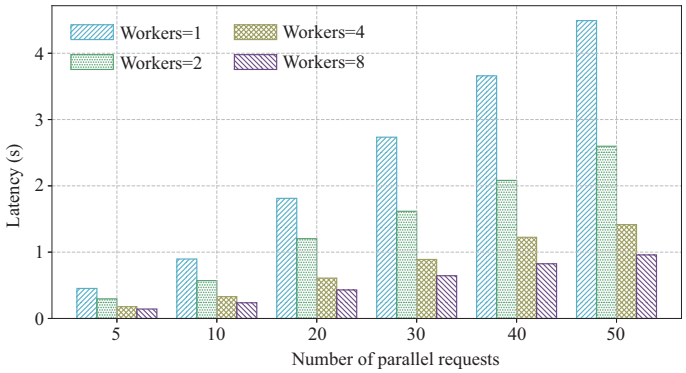

图 2　(网络版彩图) Session 内不同垂直并发度下 (worker 数量) 处理不同请求数量的时延对比

Figure 2　(Color online) Parallel requests processing efficiency when there are more workers in one single session

执行, 因此在单个 Session 中并发请求只能在容器中串行执行, 导致较高的端到端时延. 因此, 我们还模拟了在单个 Session 中支持多并发数据访问的性能 (即单个 Session 内可创建的工作进程数量大于 1 时), 以探究在 Session 机制中开启垂直并发度的重要性.

根据图 2 的结果显示, 当 Session 容器不允许垂直并发 (即 worker 工作进程数为 1) 时, 并发请求在单个 Session 中串行占用容器资源, 导致请求时延随着并发请求数量的增加呈线性增加趋势. 然而, 当 Session 内垂直并发度开启 (即工作进程数大于 1) 时, 随着 Session 内支持的并发处理能力增加, 请求时延显著降低. 这表明基于会话粘滞的有状态数据本地内存访问方案需要 Session 内支持垂直并发, 以有效支持高并发访问, 从而提供较低的访问时延.

因此, 我们得出的第 1 个见解是, 单个 Session 内的所有请求可以由同一个容器执行, 以实现对有状态数据的连续操作, 但缺乏并发请求在同一个容器内的垂直并发支持. 由于 Serverless 容器通常不允许其内部请求并发执行, 当函数高并发请求到达时, 请求会被串行执行, 导致数据访问时延增加和性能下降. 为了解决这个问题, 我们需要在单个 Session 内引入垂直并发的支持, 并实现请求之间的安全隔离和资源的按需扩展.

2.3　Session 内请求并发的数据一致性

由于有状态函数通常依赖于上下文数据, 这些数据可能会在多个不同的请求中被访问. 当多个并发请求同时对同一份数据进行写入操作时, 就会出现数据冲突的风险 [16,17]. 例如, 一个请求读取数据并基于该数据进行计算, 而在该计算完成之前, 另一个请求对数据进行了修改. 这种情况下, 计算结果可能会变得不正确, 或者基于过期的数据进行操作.

如果请求所属的函数需要强一致性保证, 即要求数据操作具有强一致性, 那么并发的读写操作会破坏数据的一致性. 为了解决这个问题, 常用的解决方案是引入互斥锁. 互斥锁可以确保在同一时间只有一个请求或一段临界区能够对数据进行写入或读取操作, 其他请求则需要等待锁释放后才能执行操作. 通过使用互斥锁, 可以保证数据操作的互斥性, 从而避免数据不一致的发生. 然而, 对于不需要强一致性保证的请求, 即只需要弱因果一致性保证的请求, 可以采用不同的策略. 在这种情况下, 对于同一份数据的读写操作不需要引入互斥锁机制, 只需要使用事务来保证单次请求内的一系列读写操作是原子化的. 通过将这些操作包装在事务中, 可以确保它们要么全部执行成功, 要么全部回滚, 从而保持有状态数据在单次请求中的多个读写操作是因果一致的 [18].

适用传统方式部署 Serverless 有状态函数时, 高并发请求会导致 Serverless 平台水平扩展多个容

器以并行处理请求, 这使得有状态数据被访问时跨越多个容器. 在这种情况下, 应用开发人员需要在函数逻辑中额外考虑跨容器间的数据锁管理, 然而这种方式不可避免地引入了更多的锁同步开销. 相比之下, 在基于 Session 的请求垂直并发模型中, 我们能够在单个 Session 中维护唯一的数据锁, 从而以更低的开销方式管理不同请求之间的锁同步流程, 并解决数据一致性问题.

因此, 我们的第 2 个见解是, 在单个 Session 内实现请求垂直并发时, 需要考虑数据访问一致性. 由于多个请求在同一个容器中并行, 彼此之间可能会互相干扰或覆盖对方的操作结果, 会导致数据竞争和不一致的情况发生. 我们需要在运行时为其封装有状态数据的统一访问接口和原子化事务抽象, 并且在不同数据一致性要求下 (强线性一致性和弱因果一致性) 实现对应的请求并发模型.

2.4 基于 Session 的安全调度

在为 Session 引入了请求的垂直并发和安全隔离后, 我们考虑了 Session 中并发请求的数据一致性问题, 确保了单个 Session 空间内请求执行和数据操作的正确性. 然而, Serverless 平台的特点是应用开发人员只需要考虑构建函数粒度级别的应用工作流, Serverless 调度器会根据负载变化动态的调度容器资源, 不需要额外考虑请求和容器之间的对应关系. 因此, 现有 Serverless 系统没有向应用开发人员提供该类 Session 和请求的映射接口, 我们还面临一个基于 Session 的系统层调度机制缺失的挑战. 其中最关键的问题是如何在保护应用隐私性的前提下, 为请求添加 Session 标识, 并根据 Session 信息实现动态的请求路由.

使用传统方式添加 Session ID 时, Session 信息保存在 Serverless 平台的服务器端, 且这个信息交由服务器端解密. 但开发人员可能不希望请求所属 Session 的分类策略向 Serverless 调度器暴露, 因为应用根据 Session 分类可能构建出相应的用户画像, 而这些用户画像属于应用的敏感隐私信息. 同时, 应用开发人员也不能将此类信息明确标注在进入 API 网关的请求中 (例如 HTTP 请求的头部), 因为将会对应用的安全性构成威胁. 攻击者可以注入恶意输入并生成异常的 Session 调用, 破坏 Session 中的有状态数据, 导致应用工作流的执行顺序失控. 此外, 攻击者还可以向特定 Session 发送大量调用, 通过此方式攻击现有服务质量或为应用程序账户生成大量账单. 因此, 任何暴露的 Session 都可能扩大应用的攻击面, 从而危及整个应用程序的完整性.

因此, 我们的第 3 个见解是, 在 Serverless 平台中, 为了安全性考虑, 开发人员需要为请求添加和管理 Session 标识分类信息, 同时为了保证应用数据的隐私性, Serverless 平台应对请求和 Session 的标识策略不可知. 我们需要保证在为请求标识 Session 信息的映射函数不向 Serverless 平台暴露的前提下, 实现 Session 间请求的安全路由和数据访问.

3 XFaaS 框架设计

XFaaS 基于动态可寻址 Session 设计, 从而实现有状态数据的高效访问. 如图 3 所示. 其中每个函数有一个函数调度器来管理其内部的所有 Session 容器. 函数调度器可以根据请求中携带的 Session hint 信息将请求调度至对应的 Session 容器中执行.

每个 Session 拥有自己的 Session 空间 (Session namespace), 用于隔离函数内不同 Session 的运行环境, 其内部可以创建一个主 Session 容器 (primary Session container) 和多个从 Session 容器 (secondary Session container). 在主 Session 容器中, 有一个 Session proxy 守护进程和一个 Session state 数据结构, 而在水平扩展的从 Session 容器中则不包含 Session state. Session proxy 负责转发请求并调度 wasm 进程处理, 同时管理 Session 容器中的 Session state 数据结构.

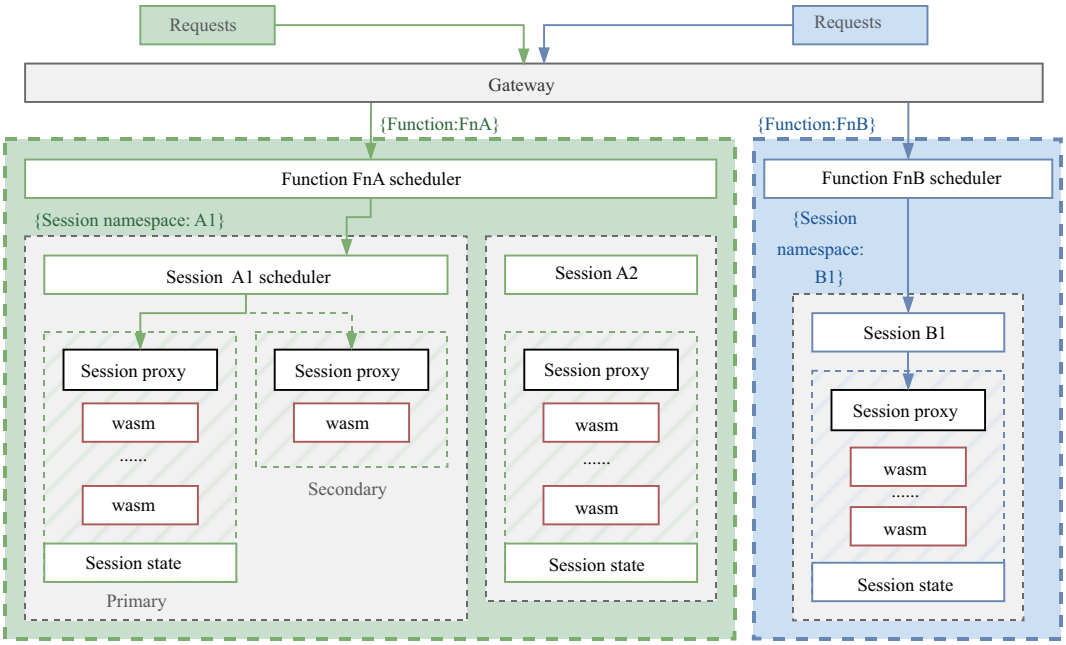

图 3　(网络版彩图) XFaaS 框架设计

Figure 3　(Color online) System design of XFaaS

当一个请求到达时, 网关会将其调度到相应的函数调度器中. 请求进入函数调度器后, 用户定义的 hook 函数可以提供请求和对应 Session 之间的映射关系, 并向该请求的事件触发器注入 Session hint 信息. 函数调度器根据 Session hint 将请求调度到相应的 Session 调度器中. Session 调度器采用 bin-pack 策略, 优先将请求调度到主 Session 容器中. Session proxy 会监听请求并选择空闲的 wasm 进程来处理请求. 如果 Session 容器中没有足够的 wasm 进程来处理请求, Session proxy 将创建一个新的 wasm 进程并将其加入到 Session 容器的 wasm 调度池中. 在主 Session 容器中, 所有的 wasm 进程都可以通过 Proxy 访问 Session state 数据结构中的有状态数据.

如果主 Session 容器中的请求达到了垂直并发度的上限, Session proxy 会向 Session 调度器请求水平扩展一个新的 Session 容器, 称为从 Session 容器, 并将请求重新调度到新创建的从 Session 容器中进行处理. 当从 Session 容器中的 wasm 进程需要访问 Session state 时, 从 Session proxy 会通过 RPC (remote procedure call) 方式调用主 Session 容器中的数据访问方法, 并等待调用返回后再将数据发送给对应的 wasm 进程. 一旦 wasm 进程完成请求的执行, 结果将依次返回给 Session proxy 和请求的发起方.

3.1　运行时隔离和 Session 抽象

在面对 Session 内请求并发的情况时, XFaaS 面临的第 1 个挑战是如何保证 Session 内并发请求的运行时隔离. 为了应对这个挑战, 需要在两个方面进行保证. 首先, 必须确保在 Session 内执行的请求在 Host 主机端具有安全性, 以防止恶意请求对主机系统造成损害. 为此, 需要在 Session 内部实施适当的安全措施, 并提供按需的隔离级别保证. 其次, 需要确保 Session 内请求的资源隔离, 以避免在高并发请求场景下由于资源竞争而导致性能下降. 这意味着每个请求在 Session 内部都应该获得独立的资源分配, 以避免彼此之间的干扰.

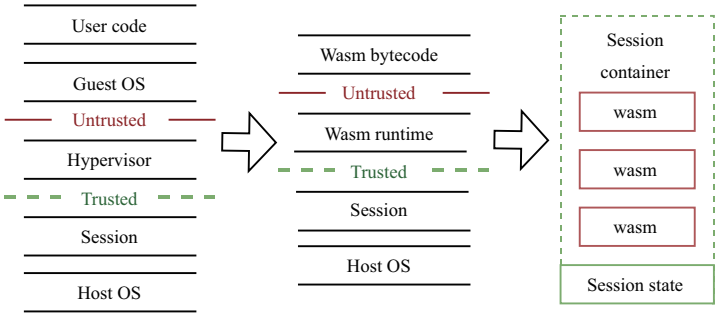

图 4 (网络版彩图) XFaaS 的安全模型

Figure 4 (Color online) Security model for XFaaS

为了安全隔离问题, XFaaS 引入了一个安全模型, 如图 4 所示. 为了确保在同一 Session 内的请求具有强隔离性和安全性, 我们将每个请求置于单独的 Guest OS 中, 并限制其对 Host OS 上的系统调用. 通过这种方式, Guest OS 能够有效保护请求和有状态数据在 Session 内的安全性, 预防恶意请求对主机系统造成损害. 对于不同的 Session 之间, 并不存在强隔离性的需求, 因此它们可以共享 Host OS 上的资源. 通过 Guest OS 的隔离性, 每个 Session 内保存的有状态数据都能够得到可信的管理, 确保数据的完整性和安全性.

然而, 为每个请求分配一个沙箱式的 Guest OS 实现过于重量级, 这导致了 Session 内并发请求所能使用的内存资源被大量的 kernel 文件占用, 使得实际上用户执行请求所使用的资源远远小于其申请的量. 此外, 来自请求的系统调用在与 Host 内核进行交互时, 由于用户态和内核态之间的切换, 带来了显著的性能损耗, 导致请求的运行性能明显弱于原生方式. 因此, 需要对 Session 的抽象进行优化, 以满足性能和资源利用的需求. 具体来说, 需要满足以下要求:

• 在同一 Session 内的请求之间引入一个能够保证强隔离性和安全性的轻量级运行时环境, 以避免过多地占用内存资源.

• 不同 Session 之间仅需要进程级的隔离, 通过隔离地址空间的方式, 确保有状态数据不会跨 Session 进行申请和使用.

XFaaS 采用 wasm 作为隔离用户请求的运行时环境, 并使用容器作为不同 Session 之间的有状态数据隔离. 通过上述抽象优化, 我们能够在不牺牲性能和资源利用率的情况下, 构建一个更轻量级的执行环境, 并使得系统能够更好地满足性能和资源的需求. 用户的代码首先被编译成一种受虚拟机保护的可移植虚拟指令集 (.wasm), 该指令集可以通过 WASI (wasm system interface) 在不同指令集架构的机器上以接近原生性能运行. 在 XFaaS 中, 每个请求都在 wasm 运行时级别进行隔离运行, 每个请求都拥有自己的 wasm 运行时实例. 当函数面临突发负载时, Session 容器内可能没有足够的空闲 wasm 进程来处理请求. 为应对这种情况, Session proxy 会触发垂直扩展操作, 增加 Session 容器的资源. Cgroup 是 Linux 内核中的一种管理进程资源使用的机制, 它可以限制进程使用的 CPU、内存等资源. XFaaS 利用 cgroup 机制来管理和分配每个 wasm 进程在 Session 容器中可使用的 CPU 和内存资源. 同一 Session 内的不同 wasm 进程之间可以实现请求间的资源隔离, 避免在高并发请求场景中由于资源竞争而导致性能下降.

同时, XFaaS 使用容器作为不同 Session 之间的有状态数据隔离机制. 在同一个 Session 容器中, 由 wasm 进程产生的有状态数据通过 socket 连接进行传输, 并保存在对应的 Session state 中. 每个 Session 都有自己的 Proxy 代理, 能够独立地管理自身的有状态数据读写, 不受其他 Session 的影响. 当

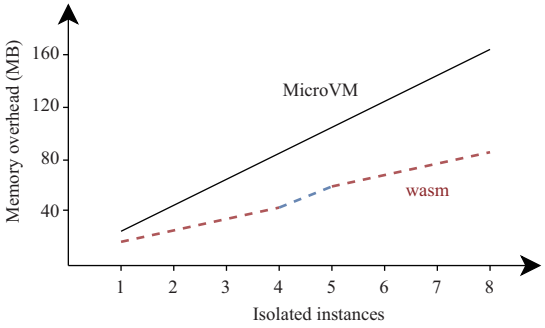

图 5 (网络版彩图) 使用 wasm 运行时与基于轻量级虚拟机隔离请求的内存开销对比. 其中 wasm 的内存开销曲线中蓝色虚线表示 MicroVM 的内存开销, 红色虚线表示 MicroVM 中启动 wasm 的内存开销

Figure 5 (Color online) Memory overhead of wasm runtime and MicroVM when isolating different requests. The blue dashed line represents the memory cost of MicroVM, while the red dashed line indicates the memory cost of initiating wasm in MicroVM

同一个函数创建多个不同的 Session 时, 每个 Session 仅维护自身的 Session state, 其有状态数据只与当前 Session 绑定, 并无需与其他 Session 进行数据同步. 即使在同一个函数创建的不同 Session 中, 内部的 wasm 进程在未获得目标 Session 的授权之前, 仍由 Host 操作系统管理进程的访问权限, 以控制对有状态数据的访问, 确保数据的隐私性和安全性.

图 5 展示了基于 wasm 运行时隔离和基于轻量级虚拟机 (MicroVM) 的内存开销对比. 基于强隔离性需求, MicroVM 的隔离方式选择了学术界 SOTA 的 RunD 安全容器运行时, 使其比采用 Qemu 或者 FireCracker 等 VMM 隔离有更低的内存开销, 平均单实例仅为 20 MB 内存开销. 但 RunD 和 FireCrakcer 等基于 MicroVM 的隔离方案依然不允许单实例内请求并发, 因此内存开销随着创建的实例数量增加而线性增加. XFaaS 中采用了 wasm 运行时隔离的方案, 相比于无虚拟化的理想场景, 额外内存占用仅为 8 MB, 对比现有低开销 MicroVM 的单实例隔离方式内存开销降低了约 60%.

3.2 Session 水平扩展和数据一致性

XFaaS 提供了一种基于 Session 的安全隔离和垂直扩展机制, 在同一 Session 内, wasm 进程可以通过唯一且固定的 Session state 访问有状态数据, 而不同 Session 之间的有状态数据访问则依赖于各自的 Session state. 当 Session 内的垂直并发度过高时, 会导致 Session proxy 成为该 Session 容器处理请求的性能瓶颈, 并且会增加资源调度单元的粒度, 延长云平台拉起 Serverless 容器的时间并增加资源开销, 同时也容易导致物理节点间的负载不均衡.

为了解决这些问题, XFaaS 为每个 Session 容器默认设置了一个最高并发度, 以平衡性能和负载均衡. 当单个 Session 容器内的 wasm 并发度达到上限时, XFaaS 会触发水平扩展, 在当前 Session 空间下水平扩展出新的 Session 容器. 在这种情况下, 最初创建的 Session 容器被称为该 Session 空间的主 Session 容器, 而后续水平扩展出的 Session 容器则称为从 Session 容器.

图 5 中同样探究了触发从 Session 容器创建的内存开销. 为了方便观察, 我们将 Session 中的垂直并发上限设为 4. 蓝色虚线部分即为触发了从 Session 容器扩展时的内存开销. 由于前 4 个创建的 wasm 隔离实例在主 Session 中, 在第 5 个创建的 wasm 隔离实例需要伴随着从 Session 的创建, 此过程伴随着大约 10 MB 左右的内存开销. 因此, 即使触发了从 Session 的扩展, 其内存开销同样低于基于 MicroVM 的方案.

表 1 XFaaS 向用户提供的系统接口

Table 1 System interfaces provided by XFaaS to users

User interface	Interface description
`Read_input(int argIndex, unsigned char* buffer, int bufferLen)`	Read input data received by a proxy
`Set_output(unsigned char* buffer, int bufferLen)`	Save execution results of a function
`Read_state(char* key, unsigned char* buffer, int bufferLen, int mode)`	Read stateful data of a Session
`Write_state(char* key, unsigned char* buffer, int bufferLen, int mode)`	Modify stateful data of a Session

为了避免在同一 Session 空间中的主从 Session 之间引入有状态数据的拷贝和同步开销, 从 Session 容器在设计上与主 Session 容器有所区别. 从 Session 容器内部不设置 Session state. 从 Session proxy 除了接受 Session 调度器调度的请求并将其并发发送给内部的 wasm 进程执行外, 不直接与主 Session state 进行交互. 当从 Session 容器中的 wasm 进程需要访问 Session state 时, 从 Session proxy 会通过 RPC 方式调用主 Session 容器中的数据访问方法, 并在等待调用返回后将数据发送给对应的 wasm 进程. 在同一 Session 空间内创建的 wasm 进程中, 对有状态数据的访问均使用了表 1 中所示的规范化数据调用接口, 这样可以确保在 XFaaS 中不引入数据副本的多实例, 从而避免了有状态数据存取的额外同步开销.

XFaaS 通过引入 Session 容器、Session state 和 wasm 进程的组合实现了具有状态的 Serverless 函数. 由于 Session state 是有状态的, 当多个进程并发读写同一份数据时, 可能出现数据不一致的问题. 例如, 如果在一个 Session 空间内, 两个 wasm 进程同时对 $x = 0$ 执行读后写操作 $(x = x + 1)$, 我们期望得到基于强一致性的结果 $(x = 1, x = 2)$, 而不是弱一致性的结果 $(x = 1, x = 1)$.

为了解决这个问题, XFaaS 引入了原子化事务抽象的概念, 以确保 Session 状态的线性一致性. 具体而言, 在定义函数时, 可以使用原子化事务抽象来指定哪些操作应该是原子的, 原子化事务的范围由数据访问接口中的 mode 参数指定. 对强线性一致性而言, 面向有状态数据设计, 对任一有状态数据的操作可以抽象为 `read` 和 `write`, 且需要使用 mode = 1 的读写参数, 即 `read_state(..., mode = 1)` 和 `write_state(..., mode = 1)`, 以定义原子化事务的起点和终点. 原子事务将任何请求中一对匹配的 `read` 和 `write` 操作之间视为临界区 (critical section). 对弱一致性而言, 不需要请求中一对匹配的 `read` 和 `write` 操作, 单独的读/写操作可直接提交. 无论是定义的原子事务或者是单独读/写操作, 来自主 Session 中的 wasm 进程的数据访问均提交至主 Session proxy 处理. 由于从 Session 容器内部不设置 Session state, 因此请求在水平扩展出的从 Session 容器访问有状态数据需要通过从 Session proxy 向主 Session proxy 发起 RPC 操作.

主 Session 中采取了中心锁设计来管理所有读/写操作, 并提供了乐观锁和悲观锁以供用户指定处理应用中读多写少和读少写多的场景. 若有状态数据的访问是基于一对 `read` 和 `write` 定义的原子事务, 则 Session proxy 在操作 Session state 时直接把该 Session 中的中心锁加锁, 直到操作完成后才会释放锁, 临界区上锁期间其他原子事务或者单独的读/写操作无法访问数据. 若有状态数据的访问仅基于未声明原子事务的 `read` 和 `write` 操作, 则 Session proxy 仅提供因果弱一致性保证, 来自不同请求的读写操作进行排序并且按序执行, 在读/写过程中对状态加锁, 避免中间状态可见.

通过上述原子事务定义的函数, 可以确保在同一 Session 空间内的数据访问按照临界区串行执行. 如果没有定义原子化事务抽象, 则默认 wasm 进程可以并行执行, 并提供因果一致性保证.

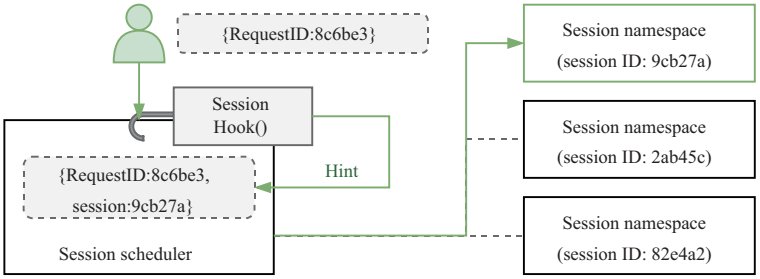

图 6　(网络版彩图) XFaaS 基于 Session hint 的请求调度机制

Figure 6　(Color online) Hint-based scheduling mechanism for sessions in XFaaS

3.3　基于 Session hint 的安全函数调度

为了保护应用的隐私和安全, XFaaS 采用了基于 Session 提示 (Session hint) 的请求调度策略, 如图 6 所示. 这种策略的核心思想是将 Session 分类决策和 Session 与请求的映射关系解耦, 不将请求的 Session 标识交由 Serverless 平台执行. 在此情况下, Session 调度器提供了一个可编程的 Session hook 函数, 开发人员可以对 Session 和请求之间的映射关系 (即 Session hint 信息) 进行添加、删除、索引和修改操作, 而 Session 调度器对具体映射逻辑不可知.

在用户事件到达 Session 调度器之前, hook 函数会将该次请求所对应的 Session hint 加密后注入到事件触发器中, 例如 HTTP 触发器的标头中. Session 调度器会根据当前事件触发器中包含的 Session hint 进行请求调度, 以实现对 Session 容器的资源调度和生命周期管理.

Session 容器恢复和冷启动. 当一个新的 Session 空间在函数中被注册时, 它需要在后台创建一个对应的 Session 调度器来管理其状态并执行相关的函数请求. 当 Session 调度器在某个物理节点上发现一个对应的 Session hint 的请求调度至当前 Session 空间时, 却发现该节点没有匹配的 Session 空间 ID 时, 就需要进行空间创建以及 Session 容器的冷启动和恢复. Session 调度器会在其他物理节点中查找是否存在匹配的 Session 空间检查点快照. 如果存在, Session 调度器会将检查点快照拉取至当前物理节点, 并使用其中的状态数据恢复 Session 空间内的容器和 Session state 中的有状态数据, 以便继续处理该 Session 的后续请求. 如果没有找到匹配的检查点快照, 则说明该 Session 空间在当前函数中首次注册. 此时, Session 调度器会创建该 Session 空间并冷启动一个空的 Session 容器, 并将其与对应的 Session 空间 ID 绑定.

Session 容器的 Bin-packing 调度. 在 3.2 小节中提到了 Session 容器中默认设置了最高垂直并行度上限, 当超过该上限时, 需要在该 Session 空间内进行水平扩展. 由于从 Session 容器中并没有 Session state, 因此从 Session 容器中所有创建的 wasm 进程对有状态数据的访问都需要通过从 Session proxy 的 RPC 调用实现. 这会导致从 Session 容器中的 wasm 进程获取有状态数据的效率要低于主 Session 容器. Session 调度器也会优先将请求调度至主 Session 容器中, 以最大程度地提高系统利用率. 只有当主 Session 容器已达到最高垂直并发度时, Session 调度器才会将请求调度至空闲从 Session 容器中. 采用 Bin-packing [19] 的调度策略可以提高 Session 空间内有状态数据的访问效率, 减少 RPC 调用的次数. 同时, 该策略还能够避免由于频繁地创建和销毁从 Session 容器而导致的额外开销, 可以有效地提高系统的吞吐量和性能.

除了 Session 容器内对请求采用 Bin-packing 调度, Session 空间内同样对 Session 容器采用 Bin-packing 调度. 当 Session 空间内的主 Session 容器到达垂直并发上限时, 会优先在主 Session 容器所在的物理节点创建从 Session 容器, 以避免 Session 空间内分布于不同物理节点的 Session 容器发起开销

较高的跨节点远程 RPC 调用. 对比 FaaSFlow [20], Nightcore [13] 等先前工作以函数作为粒度, 将所有容器实例以 Bin-packing 方式部署在同一个物理节点的方式, 以函数的 Session 作为调度粒度时能够更容易的将对同一数据访问的 Session 容器实现单节点共存. 而针对不同函数和函数下的不同 Session 空间, 函数调度器则选择 Spread 调度策略将新创建的 Session 空间调度至一个资源利用率最低的物理节点上以实现负载均衡.

Session 容器回收. 保活策略 (keep-alive) 是 Serverless 的一个核心策略. 如果容器在 keep-alive 时间内没有被再次调用处理请求, 平台会将其回收并释放资源. 然而, 对于 XFaaS 这样的有状态 Serverless 平台而言, 保证 Session 容器中的 Session state 数据在资源回收时不会丢失是至关重要的. XFaaS 在回收容器时, 如果发现 Session 空间内容器的空闲时间超过设定的阈值, 平台会先制作一个当前时刻的检查点快照, 将该 Session 空间的检查点快照和内部的 Session state 数据持久化至磁盘中, 然后才会回收该容器并释放资源. 这样制作的检查点快照将用于后续的 Session 容器恢复流程. 通过这种方式, 即使 Session 容器被回收, 内部的 Session state 数据仍然可以持续保留, 并且在下次唤醒时重新恢复, 确保数据的完整性和持久性.

Session 间直接数据通讯. 在具有拓扑依赖的 Serverless 应用中, 例如 Serverless 工作流 (Serverless workflow), 不同函数节点之间可能存在数据依赖关系 [2, 21, 22]. XFaaS 采用了 pull 模式在 Session 之间建立直接的数据通路. 在容器被创建或者回收时, Session 调度器会按照 Session 启动时的 Session 地址发布机制, 更新订阅者函数中的环境变量, 并重新建立数据通路, 以确保动态可寻址性. 通过 pub/sub 机制和动态可寻址性的设计, XFaaS 能够确保在拓扑依赖的 Serverless 应用中 Session 之间的数据通路正确建立. 这一设计保证了 XFaaS 能够将 Session 容器中的有状态数据正确恢复, 保证有状态 Serverless 应用执行的可靠性和一致性.

4 实验

本节将对比 XFaaS 与现有的有状态函数平台的实现进行性能比较, 包括数据访问性能、函数执行端到端时延、系统整体吞吐的指标、系统处理突增负载处理的稳定性.

4.1 实验配置

我们选取基于远端数据库和本地内存作为 XFaaS 中 Session 机制的比较对象. 在基于远端数据库的实现中, 依然使用 wasm 进程进行请求处理, 但统一调用 OSS 进行有状态数据的访问 (数据图中标记为 OSS). 请求中访问数据采用同步管道方式, 防止数据访问竞争导致的一致性问题. 在基于本地内存实现中, 访问有状态数据的方式遵顼学术界中最新的 Serverless 系统 FaaSFlow [20] 和 SONIC [23] 的方式, 使用本地内存或者内存数据库 Redis 访问有状态数据 (数据图中标记为 Redis).

我们使用阿里云 ECS 服务器部署了 XFaaS 系统并进行对比实验. 遵循 3.3 小节中对 Session 的调度机制的描述, XFaaS 在实验中对函数均采用 Bin-packing 调度方式, 即单个函数创建的所有实例 (XFaaS 中对应 Session 容器) 只会分布在单个物理节点上, 因此对每个 benchmark 进行的性能测试仅需单节点验证. 我们将上述 3 个系统分别部署在一台 ECS 服务器, 其中配备了 3.5 GHz 的 Intel Xeon (Ice Lake) Platinum 处理器, 包含 8 核心和 16 G 内存, 200 GB 大小 3000 IOPS 的固态硬盘用于存储数据. 所选取的基准测试程序同样包含两种类型的一致性需求和读写需求, 即弱一致性只读应用 Read_Only (RO), 弱一致性读后写应用 Read_Modify_Write (RMW), 强一致性读后写应用 Read_Modify_Write_Linear (RMW_Linear). 由于只读应用只需覆盖弱一致性需求, 所以不需要评估强

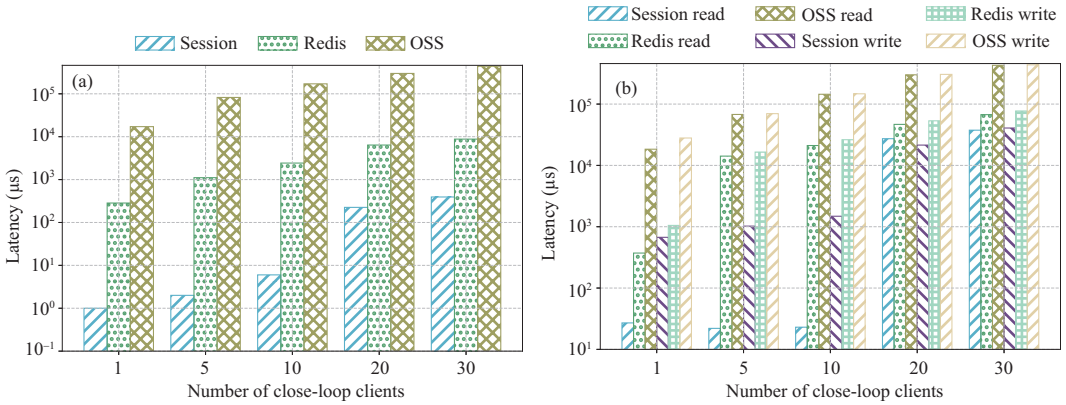

图 7 (网络版彩图) (a) 只读和 (b) 弱一致性读后写应用在不同访问数据模式下面对不同闭环负载的访问数据时延对比

Figure 7 (Color online) Comparison of latency of (a) RO and (b) RMW under different access data modes for different closed-loop loads

一致性条件下的只读应用 (RO_Linear).

Session 中运行函数请求的 wasm 进程遵循实际生产场景中的 CPU/内存分配规格, 即每个 wasm 进程使用 cgroup 限制 0.1 核和 128 MB 大小的内存规格, 而 Session 内的整体的 CPU 和内存使用限制取决于其内部所有 wasm 进程资源限制的总和.

4.2 数据访问时延

在本小节实验中, 由于不同系统平台处理请求的效率有所不同, 为了测试在特定并发度下它们对有状态数据的并发访问性能, 我们使用闭环测试[8)]以固定系统内对有状态数据的操作并发度. 在本闭环测试中, 每个客户端在接收到上一次调用的结果后立刻发送下一次调用, 形成一个调用闭环, 对应系统中一个闭环执行的 wasm 进程. 由于每个 Session 容器可以创建 10 个 wasm 进程, 因此每 10 个闭环客户端对应创建了一个 Session 容器, 内部包含 10 个 wasm 进程. 图 7 和 8 展示了 3 种应用通过 OSS, Redis 和 Session 访问有状态数据的平均时延.

如图 7 所示, 对于弱一致性的只读应用, 对比基于 OSS 和 Redis 的方式, 基于 Session 的有状态数据读取能够分别达到 3 个数量级和 2 个数量级的时延提升. 由于主 Session proxy, Session state 和 wasm 进程封装在同一个容器中, 其有状态数据的访问可直接建立于共享内存方式. 因此当客户端数量较低, 未触发从 Session 容器扩展时, Session 方式比基于 socket 的 Redis 获取中心锁和数据的方式更快, 约有 12 倍的性能提升. 随着闭环客户端数量增加, 当闭环测试的客户端数量达到 10 以上时, 从 Session 容器创建, 此时 Session 方式对有状态数据的访问增加了从 Session proxy 的 RPC 调用链, 平均时延有所增加. 在高并发场景下时, 从 Session 的 RPC 调用开销与 Redis 方式获取数据锁的开销持平, 但 Redis 的单线程模型导致数据访问只能串行执行, 因此排队延迟更长. 而 Session proxy 允许多线程执行内存拷贝, 对比最快的本地 Redis 带来了近 2 倍的时延降低.

对于图 8 中的强一致性读后写应用, 基于 Session 的机制和 Redis 有相似的读性能, 但在高并发写操作上有平均 5 倍左右的性能提升. 这是由于 XFaaS 中的原子化事务抽象使得读锁和写锁在一开始被同步获取, 因此写操作过程中无需再次获取锁. 对于强一致性读后写应用, 由于需要为临界区加锁,

8) Synchronous and asynchronous invocations in OpenFaaS. https://docs.openfaas.com/reference/async/.

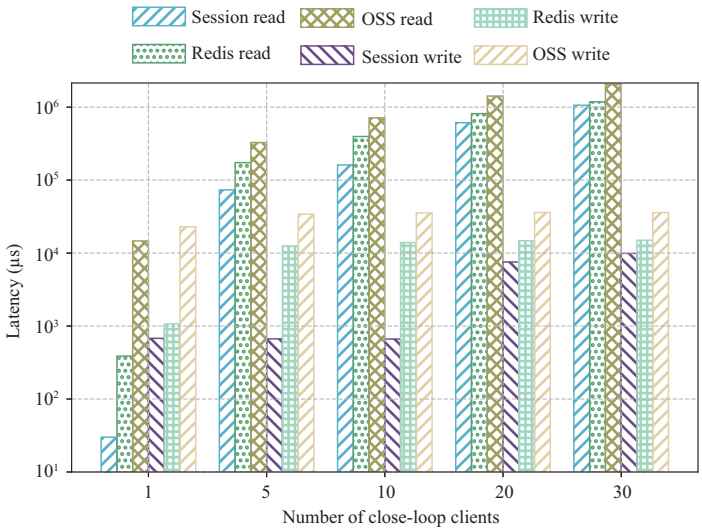

图 8 (网络版彩图) 强一致性读后写应用在不同访问数据模式下面对不同闭环负载的访问数据时延对比

Figure 8 (Color online) Comparison of latency of RMW_Linear under different access data modes for different closed-loop loads

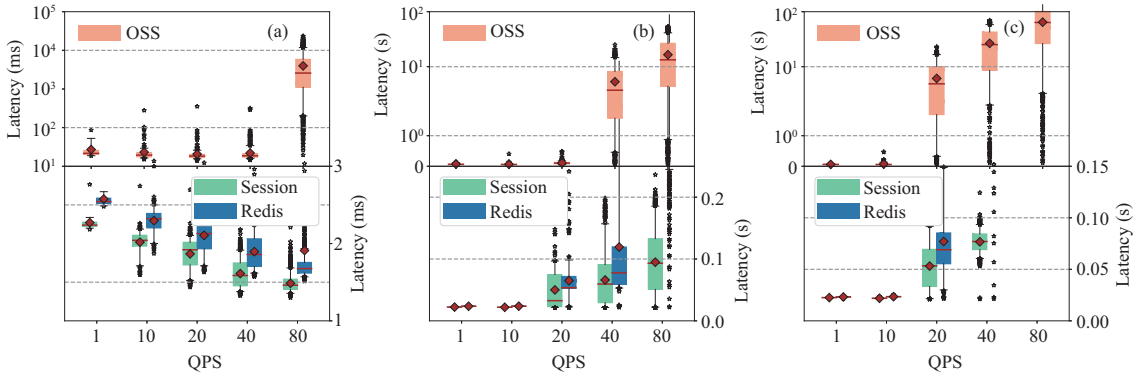

图 9 (网络版彩图) 3 种典型应用在不同访问数据模式下面对不同开环负载的端到端时延对比

Figure 9 (Color online) End-to-end latency comparison of benchmarks with different access data modes for different open-loops. (a) RO; (b) RMW; (c) RMW_Linear

保证数据的修改为串行. 基于 Session 的方式和 Redis 的方式的性能差异在于获取锁和释放锁的效率, 代表了在同一 Session 内维护单一的一把数据锁比多容器之间维护多把同步锁的机制更有效. 随着进程 loop 数量更多, 对 CPU 和锁等共享资源的竞争会愈发明显, 导致时延增加.

4.3 端到端时延

在本小节实验中, 我们使用开环测试以覆盖系统进行资源扩展的开销和排队开销. 系统中固定了工作进程的数量为 80, 即 OSS 和 Redis 方式中, 水平扩展出的容器上限数量为 80, 而在 XFaaS 系统中, 可以水平扩容出 8 个 Session 容器, 每个容器中可以垂直拓展出 10 个 wasm 进程.

我们统计了所有请求的时延并做出如图 9 所示的箱线图, 3 个子图分别代表了 3 种不同类型的应用, 其中箱体的上下边代表了请求时延分布的四分位数, 红色横线代表中位数, 红色点代表所有请求

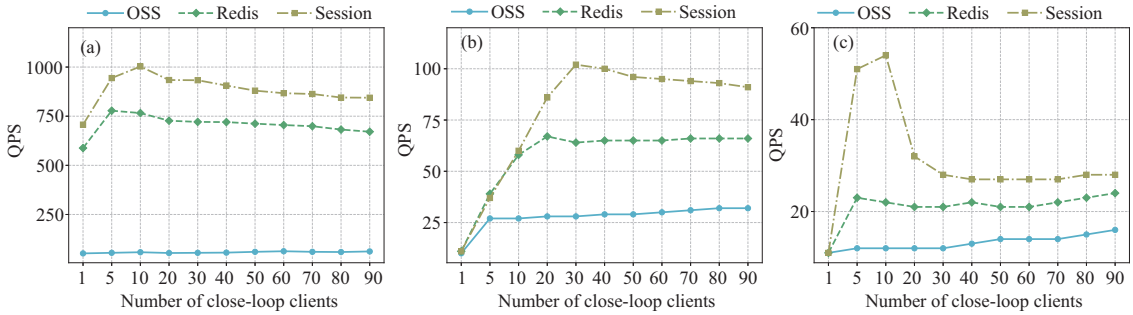

图 10 (网络版彩图) 3 种典型应用在不同闭环负载下的吞吐 (请求数量每秒) 对比

Figure 10 (Color online) Comparison of throughput of typical applications under different closed-loop loads. (a) RO; (b) RMW; (c) RMW_Linear

的平均值, 数据异常值作为离群点分布在箱体上下. 可以观察到在基于 Session 机制的 XFaaS 中, 3 种类型的应用都获得了最低的时延响应. 例如, 对于只读应用 RO, 相较于使用 OSS 和 Redis 的方式, 基于 Session 的平均时延降低了 9.4 倍和 1.19 倍, 在高并发场景下分别达到了 3 个数量级和 1.1 倍的时延降低. 除了平均值, 中位数和 p99 尾时延等其他指标也都低于 OSS 和 Redis 方式. 随着每秒请求数 QPS 的增加, 只读应用更频繁地触发 CPU 缓存命中, 基于 Session 的请求响应延迟仍然保持在更低的水平. 在两种读后写 (RMW) 应用中, Redis 方式的高并发请求的性能显著下降, 时延增加近 15 倍, 而基于 Session 的机制能有效保证高并发场景下请求的稳定执行, 并将性能降低控制在 3 倍以内. 这是由于在资源量一定的情况下, 未得到处理的请求会在系统中排队. Session 方式由于能够更快地获取有状态数据, 单请求处理时延更短, 因而系统中的排队时延更低. 而 Redis 方式因为有更多请求在系统中排队, 时延的累加效应导致系统严重拉长请求处理的尾时延. 这表明基于 Session 的机制在请求高并发场景下能够在保持较低时延的同时维持稳定的性能水平.

4.4 系统峰值吞吐

在本小节实验中, 我们希望评估在相同资源占用下, 不同系统所能提供的峰值吞吐. 因此我们使用闭环测试以保证所有系统具有相同的工作进程, 通过增加闭环数量观察系统吞吐变化. 每个闭环客户端对应系统中 1 个闭环执行的 wasm 进程, 每个 Session 容器可以创建 10 个 wasm 进程, 因此实验中创建的每 10 个闭环客户端对应 1 个 Session 容器.

图 10 展示了在不同闭环测试进程数量下的系统级吞吐. 在闭环测试中, Session 方式均提高了系统支持的吞吐上限. 而基于 Redis 和 OSS 的方式在闭环测试的客户端数量达到 10 或者 20 时就已经达到系统吞吐上限. 这说明了 Session 方式可以降低并发请求访问有状态数据的资源开销, 显著提高整体系统的吞吐量. 相比于 Redis 的方式, Session 方式在 3 种应用中分别提升了系统最大吞吐量 1.3 倍、1.52 倍和 2.3 倍. 对于只读应用和弱一致性读写应用的高并发场景, Redis 和 Session 方式获取数据锁的开销类似, 但吞吐上限存在差异的主要原因是: Redis 的单线程模型导致数据访问只能串行执行, 因此系统级的应用吞吐受限于 Redis 的吞吐上限. 而 Session 方式中的 Session proxy 可以并行处理来自 wasm 进程的有状态数据内存拷贝请求, 吞吐瓶颈不存在于数据访问处, 而受限于系统的物理机资源. 系统峰值吞吐的影响除了来自于上述原因之外, 还受数据一致性管理中的锁同步开销影响. Redis 方式对比 Session 方式处理并发请求时会创建更多的容器, 大量容器间的锁同步占据大量时间片, 导致容器分配的 CPU 率先成为资源瓶颈限制吞吐.

当闭环测试中的客户端数量增加时, 系统吞吐量都会因为 CPU 瓶颈而达到饱和状态, 进一步增

表 2 不同冷启动方式的时延对比

Table 2 Comparison of latency of different cold startup methods

Startup in XFaaS	Startup latency (ms)	Startup description
–	262	Traditional Serverless containers cold startup scale out
Fork	1.98	Wasm instance scales within a Session container
CopySession	263	Session container scales within a Session namespace
ColdSession	268	Create session namespace for the first time

加负载将导致系统吞吐量下降. 然而, 当使用 OSS 时, 系统吞吐量受限于网络传输 IO 效率, 无法进一步通过增加负载提升系统吞吐量. 这意味着 XFaaS 使用 Session 机制能够比使用本地 Redis 更有效提高系统吞吐, 缓解多容器进程间锁同步带来的有状态数据访问开销. 同时对比基于 OSS 的有状态数据访问, Session 机制能够有效避免因为网络 IO 而导致的吞吐瓶颈.

4.5 冷启动开销

在 XFaaS 中, 函数请求会在得不到空闲 wasm 进程处理时触发冷启动. XFaaS 根据当前 Session 的状态会选择不同的冷启动方案, 以最小化冷启动开销. 我们在本小节中评估 XFaaS 中所有出现的冷启动方式并统计对应启动时延. 在本小节实验中, 我们使用开环测试并调整请求间的发送间隔, 以保证每次请求都正确触发对应的冷启动方式. 表 2 展示了 Session 机制内的不同冷启动方式时延.

如表 2 所示, 传统 Serverless 系统仅通过水平扩展来处理突增负载, 平均开销高达 262 ms. 此种冷启动对应 XFaaS 系统中的 ColdSession 方式: 当函数在系统中创建 Session 空间时, 需要检查节点中是否存在该 Session 容器的 checkpoint 文件, 若存在, 恢复时需要重新载入进 Session 容器内存的 Session state 结构中. 实验中测试的有状态数据结构大小为 1 MB, 完整的数据载入需要额外花费 5 ms 左右的时间片. 实验测定一轮完整的 ColdSession 创建需要花费的平均开销在 268 ms. 而当 XFaaS 通过 Fork 方式创建 wasm 进程的平均开销在 1.98 ms. 因此, 当后台已经存在主 Session 容器时, XFaaS 通过 Fork 方式扩展的开销为传统方式的 0.75%, 扩展效率远优于基于水平资源扩展的传统 Serverless 系统. 若主 Session 容器内达到了垂直并发度上限, 将会触发 CopySession 在该 Session 空间内创建从 Session 容器, 平均开销为 263 ms, 与传统 Serverless 水平扩展容器的时间持平.

4.6 突增负载响应

考虑到 Serverless 模式更适用于处理工作负载突增的场景, 因此我们在该实验中评估了 XFaaS 在 Session 中引入垂直并发度的机制有效性, 体现在是否能够以更低的响应时延处理突增负载场景. 在实验中, 我们选取了两个负载较高的应用 RMW 和 RMW_Linear 作为测试对象, 使用开环测试维持 QPS 为 1, 并在一定时间后 (5 s 后) 突然增加测试负载测 (从每分钟 1 个请求增加到每分钟 100 个请求). 我们改变 XFaaS 中 Session 的垂直并发度限制 (即可创建的 wasm 进程数量), 统计了在各个垂直并发度下请求的完成时延. 图 11 中展示了这些请求完成的时延累积分布图.

如图 11 所示, 当 worker 数量为 1 时, XFaaS 退化为传统的 Serverless 执行模型: 每个容器只能串行执行用户请求, 处理并发请求需要通过水平扩展和同步锁来保证数据一致性. 当 worker 数量大于 1 时, XFaaS 采用 Session 机制, 每个容器可以在 Session 容器内创建 worker 实现垂直并发.

对于弱一致性读后写应用 (RMW), 随着 worker 数量增加, 并发请求会优先在 Session 内访问内存中的有状态数据, 因此延迟更低且分布更稳定. 而对于强一致性读后写应用 (RMW_Linear), 随着

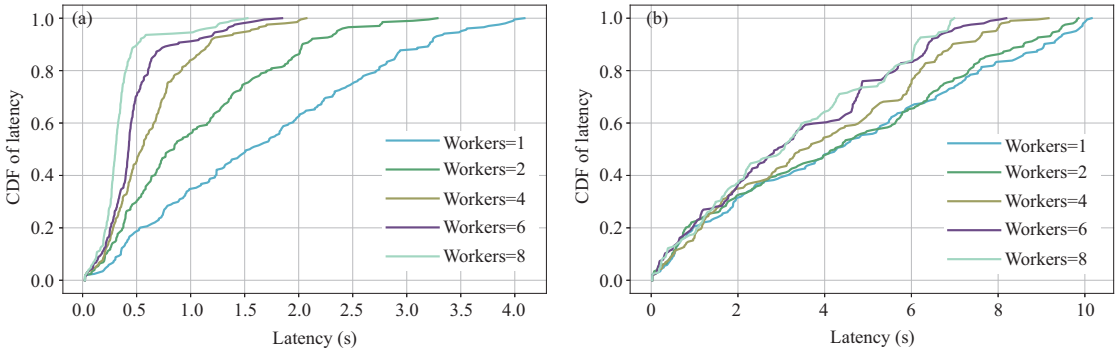

图 11 (网络版彩图) 两种典型应用在不同垂直并发度下面对负载突增的端到端时延累积分布曲线

Figure 11 (Color online) Cumulative end-to-end delay distribution for typical applications with different vertical concurrency for bursts. (a) RMW; (b) RMW_Linear

worker 数量增加, 由于临界区加锁的原因, 并发请求在系统中串行执行, 因此增加 worker 数量并不能显著降低突增请求的端到端延迟. 然而, XFaaS 通过 Session 内的 Fork 方式能够更快地扩展所需的 worker 进程, 响应请求从而提高性能.

5 Session 机制的讨论

为了解决原生 Serverless 架构中函数请求的不可寻址性, 本文基于 Session 机制构建了面向有状态函数应用构建的 XFaaS 系统. 在 XFaaS 中, 来自同一 Session 的函数请求始终被路由到同一个容器中执行, 并且可以通过该容器的地址唯一检索生成的各种数据和状态. 虽然实验验证了该机制能够有效提高有状态数据的访问速度、应用执行效率和系统最大吞吐, 但也存在一些局限性.

首先, 请求数据与 Session 地址绑定, 固定且不可变的 Session 地址可能对数据的安全性构成风险. 如果恶意用户或攻击者获得了固定地址, 并能够持续访问数据, 就可能面临数据泄露或未经授权的篡改风险. Serverless 云平台需要采取适当的安全措施来确保只有经过授权的用户或服务能够访问 Session 中的数据. 其次, Session 命名空间创建后, Session 内的数据便与当前物理节点绑定. 如果对该 Session 内数据存在依赖关系的其他函数位于其他物理节点, 而依赖的函数必须通过该 Session 地址访问数据, 可能会导致网络延迟和性能下降. 对于需要频繁访问远程数据的函数, Serverless 云平台需要设计新的应用负载均衡策略来最小化数据访问的开销. 最后, 由于函数的扩展变为更精细的 Session 容器粒度, Serverless 云平台需要引入自适应的资源调控策略以避免单一 Session 命名空间征用过多资源, 或者过度配置 Session 容器的资源从而降低节点的资源利用率.

除了对 Serverless 云平台提出的新挑战, 使用 Session 机制也需要应用开发人员仔细地设计当前业务的 Session hint 分类策略. 用户是一种常见的 Session 分类策略, 但对于存在在海量用户的应用 (如 AWS 和 Alibaba 的典型业务电商, 或者活跃用户数量达到千万级的应用), 可能会导致创建过多 Session 使得数据持久化、Session 恢复时的数据载入频繁触发, 从而引入过多资源开销. 本文举例了以下 3 种 Session 分类的最佳实践以避免上述问题.

• **用户组.** 一种常见的 Session 分类方法是根据用户标识符 (例如用户 ID) 将用户分类. 当用户登录时, 应用程序会创建一个新的 Session, 并将该 Session 与该用户 ID 相关联. 当应用存在海量用户时, 不适用于为单独用户创建独占的 Session, 我们可将拥有相似行为或者特征的用户组关联为一个 Session. 例如, 一个电商网站可能会根据用户购买历史或兴趣爱好建立不同的用户画像类别. 这通常

需要依赖于用户数据的分析和挖掘, 以便于识别和分类用户.

• **地域.** 基于地域的 Session 分类替代了基于地理位置的负载均衡和 CDN 加速服务. 与 CDN 类似, 通过在不同地域 (例如国家、城市等) 部署同一个函数应用的不同 Session 节点, 可以将数据和计算资源尽可能地靠近用户. 基于地域的 Session 分类需要开发人员对各地域的负载情况做预先调研, 以防止过密的 Session 分类导致的访问带宽瓶颈, 或过稀的 Session 分类导致的低资源利用率.

• **SLA 感知.** 对于一些数据敏感性应用, 其执行时间往往与其输入的数据大小、参数相关, 系统需要考虑各应用中输入数据大小的分布和对应的 Session 配置. 通过确定特定规模下的负载情况和 Session 的资源对应情况, 从而实现服务等级协议 (service level agreement, SLA) 感知的 Session 请求调度. 应用可以根据不同的 SLA 要求, 为不同的请求分配至不同的 Session 中执行. 例如, 标准用户可以在一些较低配置的 CPU 集群中执行请求, 相较之下有更高的延迟. 而高级用户的请求交由更优配置的 CPU 和内存型号的物理机上的 Session, 享受更低的响应时延和更高的带宽.

除了上述分类策略, Session 机制还可根据应用特征自定义其他可行的分类策略, 同时将多个 Session 分类策略结合使用使得系统能更细粒度地优化 Session 的资源占用和性能影响.

6 相关工作

轻量级 Serverless 运行时和冷启动. FireCracker [12] 是首先针对 Serverless 语义下提出低开销 VMM 实现, 使用 Linux 内核的 KVM 来提供支持现代 Linux 主机以及 Linux 和 OSv 客户机的微型虚拟机 (MicroVM). RunD [24] 通过读写分离的 rootfs、精简的 guest 内核和 host 端轻量级 cgroup 实现了 Serverless 场景下的轻量级 MircoVM, 在为函数提供硬件隔离的基础之上显著降低了高并发启动的 CPU 开销和高密度部署时的内存开销. 基于对进程初始化过程中启动的文件和加载的内存, Catalyzer [25] 和 Replayable execution [26] 基于只有少量文件和加载的内存在进程初始化过程中被使用这一观察, 按需地恢复 (on-demand restore) 用户级别的内存状态和系统状态. FAASM [27] 则直接将同一工作流中的函数放在同一进程地址空间中, 使用线程来驱动函数实例的执行, 并通过 wasm 机制来确保不同函数间的内存安全. XFaaS 中同样使用了 wasm 机制, 但隔离粒度为更细的请求级别, 并使用 Fork 机制加速了请求实例的冷启动流程.

有状态会话和服务. 有状态服务和会话是一个经典问题, 现有主流实现包括了基于数据库、Session 和 Cookie 的有状态应用构建方案. 使用关系型数据库来管理 Serverless 中会话状态可以提供强大的事务支持和数据一致性, 但在可扩展性和并发访问方面存在限制 [16, 28, 29]. 利用内存数据库 Reids 或者 Memcached[9] 等可以将会话状态和数据存储在内存中, 提供更高的访问速度和并发访问能力, 但需要考虑数据持久化和容错机制. 基于服务器端 Session 的有状态会话方案会在服务器在接收到客户端请求时为每个会话分配一个 token, 并在后续请求中使用它来识别会话[10][11]. XFaaS 基于 Session 构建了应用于 Serverless 场景的有状态方案, 并提供了 Session hint 定义以用于识别会话. 这种方案由于会占用较多服务器资源, XFaaS 采用了 checkpoint 持久化技术以减少内存占用. 另一种方式是使用 Cookie, 将会话状态和数据以 Cookie 的形式存储在客户端, 并随每个请求自动发送给服务器. 服务器

9) Memcached — a distributed memory object caching system. https://www.memcached.org/.

10) Authenticate with a backend server. https://developers.google.com/identity/sign-in/web/backend-auth.

11) AWS Security Token Service. The credentials consist of an access key ID, a secret access key, and a security token. https://docs.aws.amazon.com/STS/latest/APIReference/API_GetSessionToken.html.

无需存储会话数据, 只需验证 Cookie 即可[12)13)]. 这种解决方案简单易用, 但需要解决潜在的 Cookie 被篡改或伪造的安全风险 [30, 31].

函数间和工作流数据交换. Serverless 场景下有状态和无状态应用中的数据交换也开始受到研究人员关注. Bledi [32] 从 Olive [33] 扩展而来, 它采用了改进的有状态函数 (stateful serverless function, SSF) 实例, 并将 SSF 的状态和函数信息保存在内置的数据库中, 提供容错的工作流执行. Boki [17] 将共享的 log 索引和 Serverless 应用存放在相同的节点上, 一个应用中的所有对状态的操作可以保存在共享 log 中, 基于此保证有状态应用在访问数据的一致性. SNF [34] 将函数的计算单元和状态单元解耦, 在计算单元之间主动地复制临时状态, 并在工作流被调用时, 将函数的内部状态缓存在本地内存中, 以此提升 Serverless 函数访问有状态数据的效率. SAND [35] 和 Nightcore [13] 还实现了本地的消息队列, 避免调用的请求通过全局的 Gateway 进行裁决和任务下发的额外开销. SONIC [23] 通过在工作流中的函数之间引入了应用感知的数据传递机制, 通过在本地内存 – 直接传输 – 远端存储 3 种数据交换方式之间自适应选择性能最优的传递方案, 提高了函数在分支处理情况下的传输效率. 上述工作均考虑了函数间数据传输对应用服务性能的影响, 通过引入函数实例的数据缓存或者基于函数分布的调度机制来提高数据本地性. 对比上述以函数为粒度的数据本地性优化, XFaaS 则提供了更为细粒度的数据本地性管理, 针对单个函数或者实例中不同数据需求的调度特征, 将同一函数中的不同 Session 作为调度单元. 基于 Session 抽象机制, XFaaS 为其提供了动态双向扩展 (Session 容器内垂直扩展 wasm 进程, Session 空间内水平扩展 Session 容器) 以高效处理并发请求.

7 结语

为了解决现有 Serverless 服务器无感知计算中由于静态不可寻址特点, 使用后端存储来部署有状态函数带来的高开销问题, 本文提出了一种基于动态可寻址性构建的 XFaaS 系统. 该系统通过引入基于 wasm 运行时隔离的 Session 抽象模型、Session 内不同数据一致性要求下的请求并发机制, 以及基于 Session hit 的调度策略, 确保访问相同有状态数据的函数请求始终被路由至相同的 Session, 通过提高数据局部性来提高有状态函数对数据的访问效率. 实验结果表明, XFaaS 基于 Session 的有状态函数部署能够显著降低数据访问时延, 降低了 3 个数量级. 在高并发场景中, 相比本地 Redis 方式, 访问有状态数据的时延降低近 2 倍, 并且能够实现 3 倍以上的应用峰值吞吐提升.

参考文献

1 Li Z, Guo L, Cheng J, et al. The Serverless computing survey: a technical primer for design architecture. ACM Comput Surv, 2022, 54: 1–34

2 Malawski M, Gajek A, Zima A, et al. Serverless execution of scientific workflows: experiments with HyperFlow, AWS Lambda and Google cloud functions. Future Generation Comput Syst, 2020, 110: 502–514

3 Jonas E, Schleier-Smith J, Sreekanti V, et al. Cloud programming simplified: a berkeley view on Serverless computing. 2019. ArXiv:1902.03383

4 Shahrad M, Balkind J, Wentzlaff D. Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, Columbus, 2019. 1063–1075

5 Pu Q F, Venkataraman S, Stoica I. Shuffling, fast and slow: scalable analytics on Serverless infrastructure. In: Proceedings of th 16th USENIX Symposium on Networked Systems Design and Implementation, Boston, 2019. 193–206

12) Sticky sessions with application-based cookies in AWS. https://docs.aws.amazon.com/prescriptive-guidance/latest/load-balancer-stickiness/app-cookies-stickiness.html.

13) Cookies and user identification in Google. https://developers.google.com/tag-platform/devguides/cookies.

6 Wang L, Li M Y, Zhang Y Q, et al. Peeking behind the curtains of Serverless platforms. In: Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'18), 2018. 133–145
7 Fouladi S, Romero F, Iter D, et al. 2019. From laptop to Lambda: outsourcing everyday jobs to thousands of transient functional containers. In: Proceedings of USENIX Annual Technical Conference, 2019. 475–488
8 Boucher S, Kalia A, Andersen D G, et al. Putting the "Micro" back in microservice. In: Proceedings of USENIX Annual Technical Conference, 2018. 645–650
9 Müller I, Marroquín R, Alonso G. Lambada: interactive data analytics on cold data using Serverless cloud infrastructure. In: Proceedings of the International Conference on Management of Data, 2020. 115–130
10 Hellerstein J M, Faleiro J M, Gonzalez J, et al. Serverless computing: one step forward, two steps back. In: Proceedings of the 9th Biennial Conference on Innovative Data Systems Research, 2019
11 Stecca M, Bazzucco L, Maresca M. Sticky session support in auto scaling IaaS systems. In: Proceedings of IEEE World Congress on Services, Washington, 2011. 232–239
12 Agache A, Brooker M, Iordache A, et al. Firecracker: lightweight virtualization for Serverless applications. In: Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation, Santa Clara, 2020
13 Jia Z P, Witchel E. Nightcore: efficient and scalable Serverless computing for latency-sensitive, interactive microservices. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021. 152–166
14 Dukic V, Bruno R, Singla A, et al. Photons: lambdas on a diet. In: Proceedings of ACM Symposium on Cloud Computing, 2020. 45–59
15 Klimovic A, Wang Y W, Kozyrakis C, et al. Understanding ephemeral storage for Serverless analytics. In: Proceedings of USENIX Annual Technical Conference, Boston, 2018. 789–794
16 Sreekanti V, Wu C, Lin X C, et al. Cloudburst: stateful functions-as-a-service. Proc VLDB Endow, 2020, 13: 2438–2452
17 Jia Z P, Witchel E. Boki: stateful Serverless computing with shared logs. In: Proceedings of the 28th Symposium on Operating Systems Principles, 2021. 691–707
18 Daniel B P, Sutra P, Sánchez-Artigas M, et al. Stateful Serverless computing with crucial. ACM Trans Softw Eng Methodol, 2022, 31: 1–38
19 HoseinyFarahabady M R, Zomaya A Y, Tari Z. A model predictive controller for managing QoS enforcements and microarchitecture-level interferences in a Lambda platform. IEEE Trans Parallel Distrib Syst, 2018, 29: 1442–1455
20 Li Z J, Liu Y S, Guo L S, et al. FaaSFlow: enable efficient workflow execution for function-as-a-service. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, 2022. 782–796
21 Ao L X, Izhikevich L, Voelker G M, et al. Sprocket: a Serverless video processing framework. In: Proceedings of the ACM Symposium on Cloud Computing, Carlsbad, 2018. 263–274
22 Witte P A, Louboutin M, Modzelewski H, et al. An event-driven approach to Serverless seismic imaging in the cloud. IEEE Trans Parallel Distrib Syst, 2020, 31: 2032–2049
23 Mahgoub A, Shankar K, Mitra S, et al. SONIC: application-aware data passing for chained Serverless applications. In: Proceedings of USENIX Annual Technical Conference, 2021. 285–301
24 Li Z J, Cheng J G, Chen Q, et al. RunD: a lightweight secure container runtime for high-density deployment and high-concurrency startup in Serverless computing. In: Proceedings of USENIX Annual Technical Conference, Carlsbad, 2022. 53–68
25 Du D, Yu T Y, Xia Y B, et al. Catalyzer: sub-millisecond startup for Serverless computing with initialization-less booting. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020. 467–481
26 Wang K T A, Ho R, Wu P. Replayable execution optimized for page sharing for a managed runtime environment. In: Proceedings of the 14th EuroSys Conference, New York, 2019. 1–16
27 Shillaker S, Pietzuch P. Faasm: lightweight isolation for efficient stateful Serverless computing. In: Proceedings of USENIX Annual Technical Conference, 2020. 419–433
28 Klimovic A, Wang Y W, Stuedi P, et al. Pocket: elastic ephemeral storage for Serverless analytics. In: Proceedings

of the 13th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, 2018. 427–444
29 Schleier-Smith J, Sreekanti V, Khandelwal A, et al. What Serverless computing is and should become. Commun ACM, 2021, 64: 76–84
30 Fielding R T, Taylor R N. Principled design of the modern Web architecture. ACM Trans Int Technol, 2002, 2: 115–150
31 Bugliesi M, Calzavara S, Focardi R, et al. CookiExt: patching the browser against session hijacking attacks. J Comput Secur, 2015, 23: 509–537
32 Zhang H R, Cardoza A, Chen P B. Fault-tolerant and transactional stateful Serverless workflows. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 1187–1204
33 Setty S T V, Su C Z, Lorch J R. Realizing the fault-tolerance promise of cloud storage using locks with intent. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, Savannah, 2016. 501–516
34 Singhvi A, Khalid J, Akella A, et al. SNF: Serverless network functions. In: Proceedings of ACM Symposium on Cloud Computing, 2020. 296–310
35 Akkus I E, Chen R C, Rimac I, et al. SAND: towards high-performance Serverless computing. In: Proceedings of USENIX Annual Technical Conference, Boston, 2018. 923–935

Serverless computing based on dynamic-addressable session

Zijun LI, Yilong ZHAO, Quan CHEN* & Minyi GUO*

Emerging Parallel Computing Center (EPCC), Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240*, China*
* Corresponding author. E-mail: chen-quan@cs.sjtu.edu.cn, guo-my@cs.sjtu.edu.cn

Abstract Serverless computing, as an emerging technology within the cloud-native paradigm, has gained increasing popularity among developers due to its features such as on-demand payment, automatic resource scaling, and offloaded management of underlying infrastructure. Function-as-a-service (FaaS) serves as the primary implementation approach in Serverless architecture, enabling the decoupling and execution of applications at the function level. Most cloud vendors also provide application development services based on Serverless architecture, allowing developers to deploy fine-grained functions and automatically scale based on actual workload. However, the management of stateful data within the Serverless architecture becomes complex when deploying stateful functions. Existing solutions, such as remote or shared storage, usually fail to meet the performance requirements for accessing stateful data in Serverless functions. This paper proposes a Serverless computing system based on a stateful and dynamic-addressable session called XFaaS, which achieves low-cost data access and higher throughput for stateful functions. Experimental results demonstrate that deploying stateful functions through the XFaaS system can reduce the latency of accessing stateful data by three orders of magnitude and increase the maximum throughput of functions by over 2X.

Keywords Serverless computing, function-as-a-service, stateful function, sticky Session, container