CWASI: 用于无服务器边缘-云连续体中的功能间通信的 WebAssembly 运行时 Shim

摘要

随着云计算的发展，无服务器计算（Serverless Computing）作为一种新兴的计算范式，为开发者提供了无需管理基础设施的便利，简化了编程和基础设施管理。特别是在边缘计算与云计算的连续体（Edge-Cloud Continuum）中，无服务器计算带来了显著的优势，如降低成本、提高开发效率和更好的可扩展性。目前，无服务器函数需要不断交换数据，而无服务器平台依赖于远程服务（如对象存储和键值存储）作为交换数据的常见方法。在 WebAssembly 中，函数利用 WASI 连接到网络并通过远程服务交换数据。即使两个函数位于同一台主机，两个函数的通信仍然需要利用网络通过远程服务进行通信，这会增加延迟并增加网络开销 [1]。为了缓解这个问题，拟复现论文提出了 CWASI，一种符合 WebAssembly OCI 标准的运行时的 Shim，它根据无服务器函数位置确定最佳的函数间数据交换方法。其通过实验证明其能有效降低无服务器函数通信时的时延并提高吞吐量。然而，我注意到该论文在进行部分实验时采用的是 HTTP 技术，而实际上采用 Unix Domain Sock 进行实验效果会更接近于实际情况，因此，我对这部分实验进行了部分重新实验。

关键词： Webassembly；容器技术；无服务器；函数通信；

1 引言

边缘-云连续体充分利用了无服务器计算模式的众多优势。无服务器计算通过消除开发者在基础设施管理方面的复杂性，简化了应用程序的开发流程。最近，WebAssembly（Wasm）与无服务器计算的结合，增强了无服务器功能的可移植性，提高了性能，并扩展了对多种编程语言的支持。此外，在由边缘-云连续体等资源受限设备构成的环境中，Wasm 凭借其接近本地执行的速度、安全性以及减少冷启动时间等特点，提供了显著的优势。在 Wasm 中，传统的大型容器镜像被小型的二进制文件所取代，这些文件在一个隔离的沙箱中执行。Wasm 通常创建一个没有主机访问权限的安全沙箱，从而进一步增强了安全性。尽管 Wasm 具备诸多优势，但其对主机访问的限制在实现无服务器功能间通信时带来了额外的挑战。

目前的数据交换方式包含两种方式。第一种是通过第三方服务进行数据交换。由于目前的无服务器函数的位置通常是不可知的，因此目前大部分无服务器函数是通过第三方服务进行数据交换，它们可以提供灵活性和扩展性。这些第三方服务通常是一些键值对的远程内存服务或者是远程存储服务。然而，通过第三方服务进行通信会增加通信的网络开销和内存开销。这在由边缘-云连续体等资源受限设备构成的环境中影响较大。而且如果是使用远程存储

服务，其还会增加存储成本以及限制于单一的云服务提供商。随着 Webassembly 的发展和 Webassembly 系统接口（WASI）的发布，Wasm 将可以进行系统调用来进行网络访问，这使得将 Wasm 与无服务器函数结合起来并与远程的第三方服务进行通信成为可能。然而，对于处在同一个主机中的函数，利用远程的第三方服务无疑会增加各种开销和成本。第二种方式是函数间通信的方式。这种方式能避免远程的第三方服务，从而减少网络开销。目前常见的函数间通信方式为消息队列和共享内存。在消息队列中，两个函数之间的通信往往是通过发布/订阅模式来进行通信的。在共享内存方式中，两个需要通信的函数通过共享同一块内存区域进行通信，通过共享内存通信能极大提高通信效率，然而通过共享内存进行通信会打破不同函数所处容器的隔离性，此外，所有的函数都必须同时启动，因为共享内存地址空间必须在进程启动之前进行分配，这无疑增加了系统内存的开销。在 Wasm 中，利用 Wasm 函数启动时的静态链接能实现基于共享内存的通信。静态链接在 Wasm 虚拟机（VM）中创建一个共享内存区域，使其在模块之间可访问。一旦宿主运行时显式链接模块，Wasm 模块也可以重用。然而，静态链接在当前基于 Wasm 的边缘-云基础设施中仍然不可用，因为这需要通过宿主函数显式地静态链接模块 [1]。

拟复现的论文通过引入一个 Webassembly 容器运行时 Shim（CWASI）来促进基于 Wasm 的函数间通信。CWASI 中提出了一种新的无服务器的函数间通信模型，该模型能够充分利用函数所处的位置来进行不同方式的通信，从而减少它们对第三方服务的依赖并提高函数通信的效率。

2 相关工作

2.1 无服务器平台概述

图 1表明了 CWASI 是如何适应现有的边缘-云堆栈。图 1(a) 展示了一个标准的基于容器的无服务器平台，在该平台中，每一个函数被封装到一个容器中，容器中包含自己的依赖库和语言运行时，这提供了很强的隔离性，然而，这也使得每个容器镜像的体积过大，而且在启动时会有较高的启动时延和内存开销。

图 1(b) 展示了基于进程的无服务器平台，在该平台中，一个容器包含无服务器管理进程和无服务器函数进程，每一个无服务器函数进程包含了一个无服务器函数，所有的进程共享所处容器的依赖库和语言运行时。这个平台相较于基于容器的无服务器平台减少了开销，但是由于容器的特性，其仍然会有较高的启动时延和内存开销。

图 1(c) 展示了基于 Wasm VM 的无服务器平台，每一个函数被封装到一个 Wasm VM 中，通过 WASI 能为 Wasm 提供主机访问能力，通过导入和导出 [2] 实现 Wasm 模块的交互。Wasm 实现了接近本机的速度，并显著降低了无服务器函数冷启动问题 [3, 4]。然而，由于 Wasm 的沙盒限制，其函数间的通信受到一定的阻碍。

图 1(d) 展示了 CWASI 对应的无服务器平台，其不仅提供了 Wasm 的隔离性，还能优化功能间的通信。其方法是根据函数的位置为每个无服务器函数选择最佳的函数间通信方法 [1]。

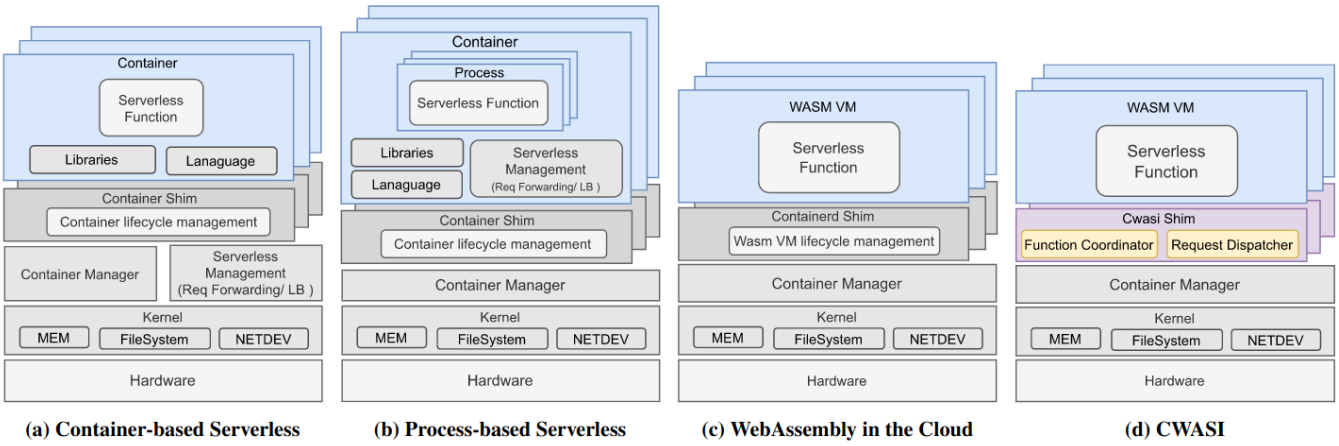

图 1. Edge-cloud 无服务器的平台概述 [1]

2.2 沙盒模型

当前的边缘-云平台利用沙箱的概念为无服务器函数提供隔离。在基于容器的无服务器平台中，这种隔离是通过 Linux 命名空间和控制组 (cgroups) [5] 实现的。像 SOCK [6] 和 Faasm [7] 这样的解决方案利用额外的沙箱来分别隔离函数、工作器和 faaslet，因此，在同一沙箱中的函数可以安全地共享资源。SOCK 采用进程间通信 (IPC)，而 Faasm 则使用分布式共享内存 [8]，这样共置的函数可以因地理位置接近而享受低延迟的函数间通信。尽管额外的沙箱减少了冷启动和函数间延迟等问题，但它们也增加了复杂性和资源开销。SAND [9] 提供了基于工作流的不同隔离方法，容器用于隔离不同的工作流，而同一容器中的进程则用于隔离同一工作流中的函数。SAND 为不同的工作流强制实施更为严格的隔离，并利用同一工作流中的容器共享来缓解冷启动问题。Sledge [10] 利用 Wasm 沙箱机制提供隔离，使得不可信的模块可以在宿主机上安全执行。但是由于 Sledge 没有使用标准的容器隔离，所以它不符合 OCI 标准。对于 CWASI，除了采用 Wasm 沙箱外，还遵循 OCI 规范提供隔离性和安全性。

3 本文方法

3.1 CWASI 模型概述

CWASI 提出了一种新的无服务器函数间高效通信的模型。图 2 展示了 CWASI 的三种通信模式，分别为 *Function Embedding*、*Local Buffer* 和 *Networked Buffer*。*Function Embedding* 通过将可信的无服务器功能组合到同一个沙箱中，让它们可以通过共享内存的方式进行通信。*Local Buffer* 通过在主机中创建一个缓冲区，让处于同一个主机上的无服务器功能能够通过这个本地缓冲区进行通信。*Networked Buffer* 则是利用网络进行通信。

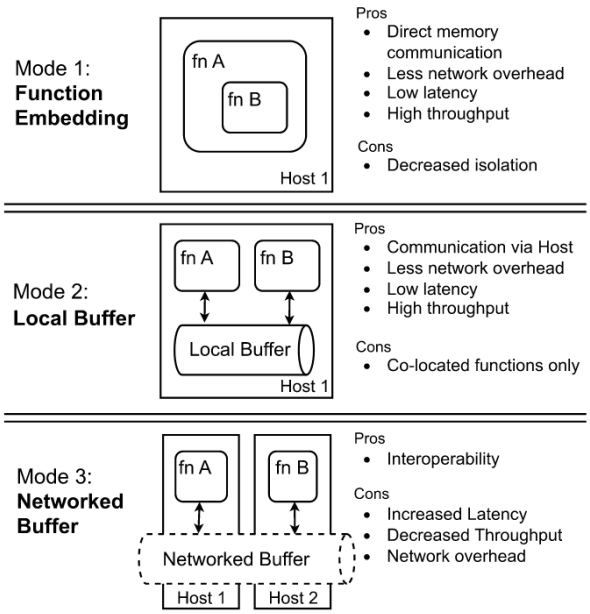

图 2. CWASI 功能间通信模型 [1]

3.2 CWASI Shim 架构概述

图 3展示了 CWASI 包含的核心组件 *Function Coordinator* 和 *Request Dispatcher*。在函数启动阶段，对于调用者模块，*Function Coordinator* 对 *Function Embedding* 通信模式进行函数嵌入发现，对于被调用者模块，*Function Coordinator* 对 *Local Buffer* 通信模式启动本地缓冲区，对 *Networked Buffer* 通信模式启动网络缓冲区。在函数运行阶段，*Request Dispatcher* 通过内部函数通信模式选择，即 *IFC Selection* 来选择来选择合适的通信模型进行通信，之后调用者模块发送通信内容，被调用者模块监听来自调用者模块的消息。

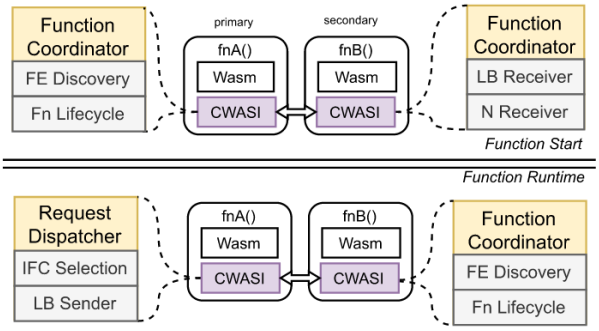

图 3. CWASI Shim 架构总览 [1]

4 复现细节

4.1 与已有开源代码对比

本次复现使用了作者在原论文中提到的开源源代码，我在源代码的基础上对实验部分进行了修改，让 Fan-out 实验不再是通过 HTTP 本地调用进行实验，而是通过 Unix Domain Socket 进行实验，这将会让实验效果更加接近真实的效果，因为通过 HTTP 本地调用会有内

核协议栈和网卡的开销，而通过 Unix Domain Socket 则没有这些开销。由于 Wasm 并不支持多线程，为了实现效果，利用了 tokio-wasi 来实现 Wasm 的异步请求。此外，由于 Wasm 是封装到 docker 容器中的，要实现 Fan-out 工作流，需要同时开启多个 docker 容器，会占据大量内存，因此，修改后的实验最多只使用了 10 次 Fan-out 请求。

4.2 实验环境设置

我的论文复现实验是在一台配置为 i7 16GB RAM 且装有 Ubuntu 24.04 LTS 的机器上进行的，该机器能够执行所有无服务器的功能，所使用的 WasmEdge 版本为 v0.11.2。我使用 Containerd 和它的客户端 cli 工具来执行无服务器的功能。此外，为了防止网络状况波动对实验结果产生偏差，我重复了十次实验，并收集了其平均结果。

4.3 调用分析

在一次的调用工作中，我先启动被调用者模块，让其创建一个本地的 Unix Doamin Socket，并监听该 sock。之后启动调用者模块，调用者模块会发送数据到被调用者模块。如图 4所示，右边的被调用者模块创建了一个 sock 并监听，而左边的调用者模块传输 10M 文件数据后右边的被调用者模块能成功接受到 10M 的文件数据。

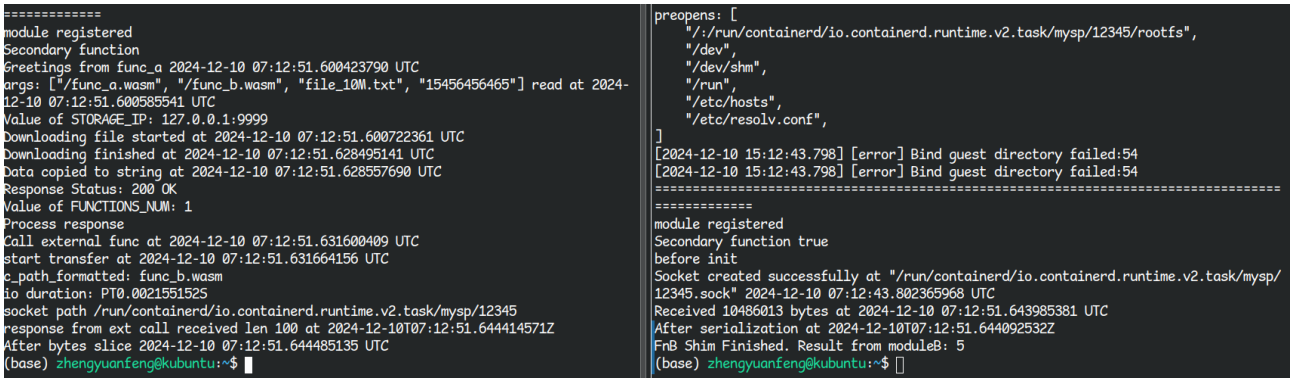

图 4. 调用示意图

4.4 改进方向

论文中 Fan-out 测试中采用 HTTP 客户端和服务端来测试所提出的内部函数通信性能。实际上的内部函数通信是通过 Unix Domain Socket(UDS) 进行通信的。在测试过程中，虽然通过 http 访问 localhost 没有网络开销，但是需要走内核的 TCP/IP 协议栈并且会有网卡的开销，而通过 UDS 进行测试就不用走内核协议栈，整体表现会更加接近实际的效果。

5 实验结果分析

5.1 Sequential 工作流

Sequential 工作流的实验结果如图 5 所示，随着输入文件的大小的增加，其时延逐步上升，吞吐量逐步减少。图 6展示了其具体的实验结果，可以看出，和原论文的实验结果相比，

我复现出来的结果略差于原论文的结果，我认为可能是实验部分的设置与原论文并不完全一样从而导致出现了偏差，此外，网络环境也可能进一步导致变差加大。

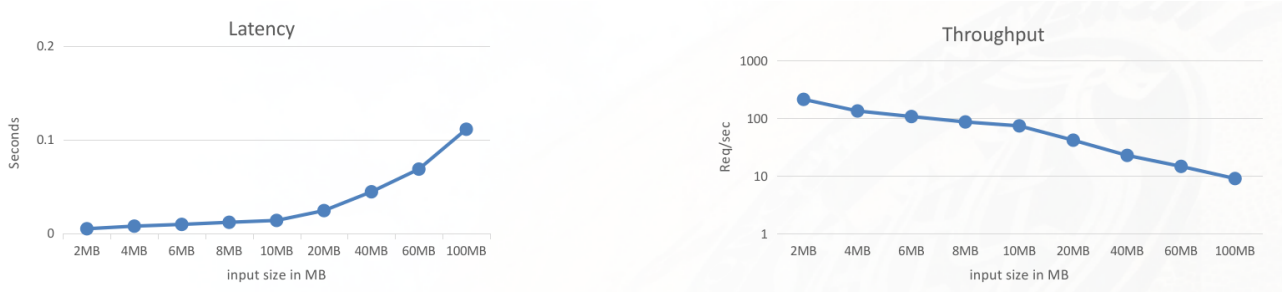

图 5. Sequential 工作流实验结果

	Latency sec		Throughput Req/sec	
	2MB	100MB	2MB	100MB
复现结果	0.00472	0.111	211.83	8.99
CWASI	0.0051	0.1436	266.4535	6.9569

图 6. Sequential 工作流具体结果

5.2 Fan-out 和 Fan-in 工作流

在 Fan-out 和 Fan-in 工作流中，可以看出我复现出的结果和原论文中的结果相差较大，我认为可能的原因是目前 Wasm 不支持多线程操作，只能利用异步编程来实现多线程的效果，而本文中利用 hyper-wasi 和 tokio-wasi 来实现 Wasm 异步处理，由于本人对 hyper-wasi 和 tokio-wasi 异步编程的了解不充分，在实验过程中可能并没有真正地利用异步编程来实现一个多线程的效果，这导致了我的实验结果和原论文的实验结果相差较大，因此，我打算之后继续学习相关知识并改进来达到原论文的实验结果。

固定输入大小2MB	**Latency sec**		**Throughput Req/sec**	
	10 Exec	100 Exec	10 Exec	100 Exec
复现结果	0.0167	0.0166	59.80	60.26
CWASI	0.0045	0.0066	203.7327	211.2039

图 7. Fan-out 工作流具体结果

固定输入大小2MB	**Latency sec**		**Throughput Req/sec**	
	10 Exec	100 Exec	10 Exec	100 Exec
复现结果	0.00139	0.00129	720.72	654.78
CWASI	0.0029	0.0032	298.9536	314.2019

图 8. Fan-in 工作流具体结果

5.3 改进后的 Fan-out 工作流

对实验方式进行改进后的 Fan-out 工作流实验结果如图 9所示，整体的实验效果会比我复现的 Fan-out 结果较好，这是因为这里同时开了多个容器进程来处理多个请求，并不是通过 HTTP 客户端和 HTTP 服务器以及异步编程来模拟并行请求处理。对于改进后的 Fan-out 工作流是否能更接近实际效果，由于这里测试并不充分，因此还需进一步的实验来进行验证。

固定输入大小 2MB	Fanout 请求数				
	2	4	6	8	10
Latency Seconds	0.0059	0.0075	0.0059	0.0059	0.0064
Throughput Req/sec	169.69	132.96	169.95	168.74	156.42

图 9. 改进后的 Fan-out 工作流具体结果

6 总结与展望

6.1 总结

目前所复现的工作部分实验不能达到预期效果，这是因为我对如何使用异步编程从而实现类似多线程的方式不够了解，为此，我将会学习相关的知识，并进一步对代码进行修改，从而达到预期结果。此外，改进部分的实验最多只支持 10 次 Fan-out 请求，与原论文的实验最高次数相差较大。为了提高实验的可靠性和有效性，我将着手修改实验代码，以支持更高数量的 Fan-out 请求。这将有助于更全面地评估系统的性能，并与原论文的结果进行更有效的对比。

6.2 展望

目前 CWASI 只支持 WasmEdge 这一个 Wasm 运行时，如何将 CWASI 扩展到其他 Wasm 运行时是一个未来的工作。为了使 CWASI 能够支持多个 Wasm 运行时，我将研究不同 Wasm 运行时的架构和接口，并制定相应的扩展方案。这能提高 CWASI 的灵活性，也将为其在更广泛的应用场景中提供支持。

参考文献

[1] Cynthia Marcelino and Stefan Nastic. Cwasi: A webassembly runtime shim for inter-function communication in the serverless edge-cloud continuum. In *2023 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 158–170. IEEE, 2023.

[2] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Weakening webassembly. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[3] Vojdan Kjorveziroski, Sonja Filiposka, and Anastas Mishev. Evaluating webassembly for orchestrated deployment of serverless functions. In *2022 30th Telecommunications Forum (TELFOR)*, pages 1–4. IEEE, 2022.

[4] Ju Long, Hung-Ying Tai, Shen-Ta Hsieh, and Michael Juntao Yuan. A lightweight design for serverless function as a service. *IEEE Software*, 38(1):75–80, 2020.

[5] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615. IEEE, 2019.

[6] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 57–70, 2018.

[7] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[8] Simon Shillaker, Carlos Segarra, Eleftheria Mappoura, Mayeul Fournial, Lluis Vilanova, and Peter Pietzuch. Faabric: fine-grained distribution of scientific workloads in the cloud. *arXiv preprint arXiv:2302.11358*, 2023.

[9] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[10] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st international middleware conference*, pages 265–279, 2020.