The Cold Start Problem: How container life cycle influences serverless functions

Ben Civjan Diego Chahuan

May 3, 2022

Contents

1 Introduction

Serverless computing has gained a huge amount of traction among developers and DevOps professionals in recent years because of its simplicity and cost efficiency in addition to all the other benefits of cloud computing. It coincides with and complements the paradigm shift from the monorepo to microservice, where developers are increasingly breaking up their applications into smaller parts for better scalability, reliability, and easier code maintainability.[Gar]

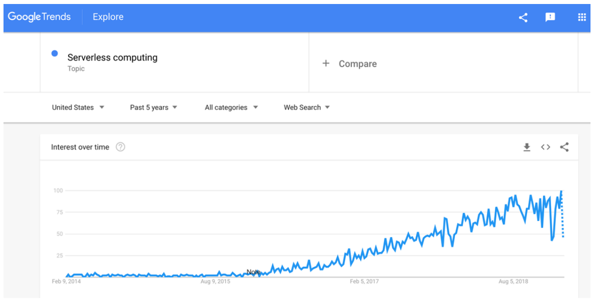

Figure 1: Google searches of 'Serverless computing' over time [Unka]

Despite the benefits that serverless computing offers, one of the biggest drawbacks of using this architecture is performance. This comes from something known as a *cold start*. This article will describe what this issue is, dive into what exactly causes it, why and how container lifecycle has an impact, and provide some solutions for solving the problem.

After reading this article, a developer with little to no experience with deploying a serverless architecture will understand the pros and cons of doing so, will understand how serverless works behind the scenes, and will be able to understand and address an important drawback of the popular design.

2 Background on Serverless

2.1 A Faster and Cheaper Way to Scale

The serverless or FaaS architecture is one of the latest models of cloud computing. Once companies began realizing that building and maintaining their own servers was becoming difficult to manage because it was tough to scale and hard to maintain good availability, the need for cloud computing was born. The movement started with virtual machines hosted on other companies servers, which allowed for a customer to have a fully managed, configured, and easily scalable server without any of the maintenance or headache that comes with owning and operating your own servers. Later with the popularization of microservices and the creation of lighter-weight virtual containers, keeping a machine running became unnecessary and overly expensive. The idea of minimizing the amount of compute time that your application must use evolved into the concept of serverless computing: Separating an application into several independent and stateless functions that only occupy memory and CPU space when run, saving resources in the cloud and offering a more affordable service to the customer. The first company to implement this concept in production was Amazon Web Services with the invention of Lambda Functions.

2.2 Functionality and Advantages

This architecture works by containerizing individual functions that are later built and executed on a specific event. This can result from an HTTP request, IoT device sending information, or any other networking event. The repeated starting and stopping of containers is what makes serverless more affordable and efficient.

If the server is receiving 0 requests, 0 containers are running and no resources are being wasted. This makes building software cheaper since the customer is only charged for what resources are actually being used. A second advantage it that this hides the complexity that maintaining your own server

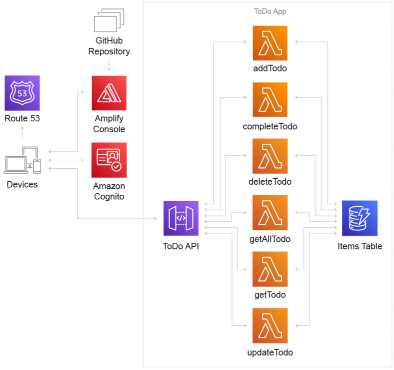

Figure 2: AWS example of serverless with lambda functions[Ser]

concurs and allows a businesses focus more resources on their product. It also provides the flexibility for teams to work in the programming language of their choice.

3 How Does a Container Work?

3.1 Namespaces and Cgroups

Containers first became possible with the implementation of *namespaces* and *cgroups*, which were both first available with the release of Linux 2.6.24 in 2008. Cgroups added onto an already developed feature, namespaces, to create even more granular control of processes. [Kal]

Namespaces limit the scope of what an executing process can see. For a concrete example, let us analyze the image below. The base namespace is PID namespace 1, which is created when Linux first boots. The user can create whatever processes they want in this namespace. Listing the processes in this namespace will display all currently executing PIDs (in this case PIDs 1 - 4).

However with containers it is desired to have complete isolation between instances, meaning each container executes as though they are the parent process on the machine. In the below example, a child namespace has been created under `PID2` and `PID3`. If we enter PID namespace 2 and list all processes, we would only see `PID1`. [Kal]

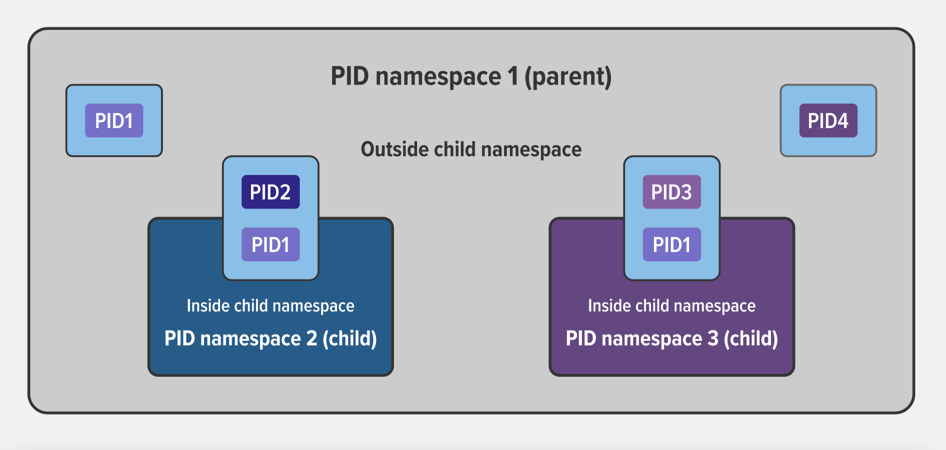

Figure 3: Example of Namespaces[Kal]

Cgroups provide granular control over the resources that a group of processes has access to. In the case of Docker, a popular container software, cgroups are used when a user wants to set a limit on the memory, CPU, network bandwidth, etc. of a specific container.

3.2 Creating and Managing a Container

This section will discuss the main stages required to initialize and manage a container as well as the costs associated with each, as described by [Unkb].

3.2.1 Pulling an image

If an image does not exist from a previous run on the machine or have made changes to your Dockerfile the server will need to (re)download your image, a ’blueprint’ for the container, from the registry (an external server dedicated to storing images). This process is known as ’pulling’ an image. This is shown as the second section in Figure 4. Pulling an image is extremely expensive, and will be described in more detail in Section 4.

3.2.2 Initializing a container

Once the image exists on the machine the server must create the container from the image, a process known as ’initialization’. This is the second most expensive stage and is shown as the third section in Figure 4.[Unkb]

3.2.3 Stopping a container

Stopping a container kills the running process and releases the memory that was held. However it keeps the state of the container, so any modified files will still exist.

3.2.4 Removing a container

Removing a container removes the state of the container, so it will need to be reinitialized from the image.

3.2.5 Starting a container

Starting a stopped container reallocates memory to the image so it can continue to run. This is much less expensive than completely reinitializing the container from the image.

4 The Cold Start Problem

The cold start problem occurs “when a request comes in and no idle container can be found for the execution of the target function, then a new container needs to provisioned. In that case, the request incurs an extra latency - the cold start latency”[BKB20].

4.1 The Influence of Containers on the Cold Start Problem

The amount of time taken to execute a function with a serverless architecture is heavily dependent on the containerization software used.

The steps can be seen in Figure 4, where a container needs to be scheduled to start, pulled from the registry, and initialized all before being able to execute code.

As previously discussed, pulling an image and initializing a container is extremely expensive. Pulling an image can take hundreds of milliseconds and initializing a container can take a similar amount of time. This creates a delay that is non-existent in traditional servers.

The usage of containers also limits the amount of concurrency that the server can achieve, since “maximum throughput of creating empty Docker containers is about 200-230 containers per second with the machine” [QIN+20]. Concurrency vulnerability issues may come into play in big scale applications or IoT devices where events that trigger lambda functions are constantly happening.

Resuming a stopped container is 100x or more faster than creating a new container, so it is generally preferred over a complete reinitialization.[QIN+20] However, stopping doesn’t release memory so only a relatively small amount of containers can exist at once in a machine’s memory. This may impact performance when RAM memory is near capacity because it will need to do frequents swaps with the

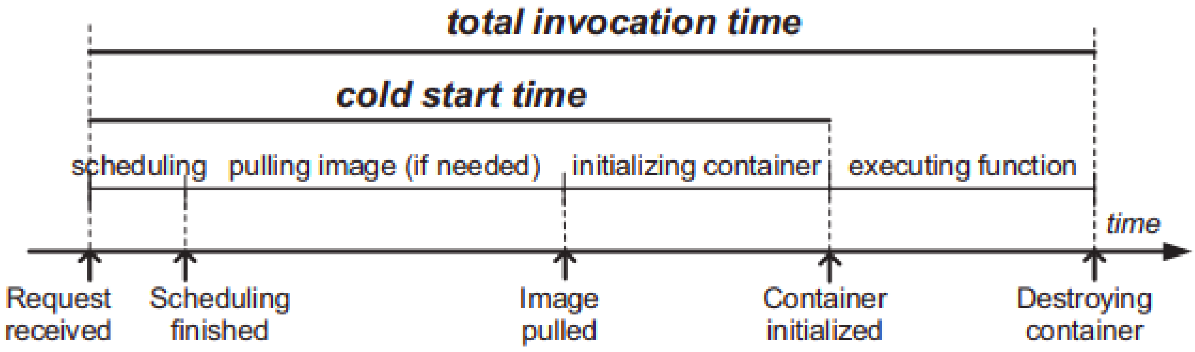

Figure 4: Function initialization process [QIN+20]

hard drive, an operation that is extremely inefficient. It also makes it impossible to scale to 0 because resources are never cleared.

With proper understanding of container software and lifecycles it is possible to optimize performance and throughput of a serverless machine. Some solutions that exploit the traits of containers for optimization purposes are described in the next section.

5 Solutions

There will be 3 different approaches discussed to counteract the cold start problem.

5.1 Näive Approach

The first approach is called pinging. The strategy involves periodically 'pinging' the function with a request to keep it active. This uses the fact that generally FaaS providers keep some containers warm a fixed amount of time. It avoids cold functions by making them permanently active.

The pros of this approach are that it simple to implement, it requires few lines of code and keeps the serverless function warm. The disadvantages are that it is resource intensive because the container is never booted out of memory and the processes can always be running. This defeats one of the core objectives of serverless, which is to use as few resources as possible. Also the container idle-time may not be constant, so you would need to check if your pinging job is running at the correct frequency.

5.2 Machine learning to determine idle time

This approach uses a reinforcement learning neural network to calculate the optimal time that the function would need to stay idle for best performance. This is accomplished by creating a function that rewards a few cold starts but penalizes high memory usage. At the end this can give you up to 12.73% less memory [VFA22] without creating more cold starts.

The advantage of using ML is that you get the same performance as keeping the container idle for about 10 minutes but use much less memory in the process. The disadvantage is that aggregating an neural network that can scale to the size of some of the providers may be hard. As well as the actual implementation.

5.3 Improving container start up time

One way to improve container start up time is with application sandboxing. This approach plays with the application isolation. Instead of thinking of function level isolation it considers two distinct level application and functions between same application. Here there must exist a strong level of isolation between applications but functions for each application do not require the strong isolation [VFA20].

This gives the possibility of starting processes instead of a new container. It also gives the possibility of sharing common dependencies.

An advantage for this is that starting a process is faster. Sharing dependencies allows memory to be saved but if the application is running it uses more resources than a unique function, so in some cases it may be worse memory performance.

6 Conclusion and Reflections

In conclusion, the cold start problem is deeply linked with the lifecycles of the docker containers and how the different stages provide different trade-offs. Where the FaaS provider needs to find a solution that gives good performance without overusing the server resources and keeping the advantages that serverless provides. Currently, several solutions have been attempted but none are perfect or give a permanent solution to the problem. In the near future FaaS will likely continue to gain popularity, but the cold start delay will be a deterrent for developers that want to create applications using this technology.

It may be possible in the future to create a new container engine with the same interface as Docker that can create containers at little cost. A new engine tailored to serverless should include fast initialization, low pausing cost and a cache of commonly used libraries speeding the whole process up. In the meantime, current software is being optimized in creative ways to minimize the expensive parts of the container lifecycle while also minimizing the computing resources used.

References

[BKB20] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. "Using application knowledge to reduce cold starts in FaaS services". In: *SAC '20: Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic, Mar. 30–Apr. 3, 2020). 2020.

[QIN+20] Shijun QIN et al. "Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing". In: *2020 IEEE International Conference on Joint Cloud Computing (JCC)* (Oxford, UK, Aug. 3–6, 2020). 2020.

[VFA20] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. "Cold Start in Serverless Computing: Current Trends and Mitigation Strategies". In: *2020 International Conference on Omni-layer Intelligent Systems (COINS)* (Barcelona, Spain, Aug. 31–Sept. 1, 2020). 2020.

[VFA22] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. "Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach". In: *IEEE Internet of Things Journal* 15 (2022), pp. 1–11.

[Gar] Ekta Garg. *Moving from monolithic to microservices? What, why and how to design microservices*. URL: https://medium.com/@_ektagarg/moving-from-monolithic-to-microservices-what-why-and-how-to-design-microservices-858aed666357.

[Kal] Scott van Kalken. *What Are Namespaces and cgroups, and How Do They Work?* URL: https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/.

[Ser] Amazon Web Services. *Serverless on AWS: Build and run applications without thinking about servers*. URL: https://aws.amazon.com/serverless/.

[Unka] Unknown. *Adoption and Market Data*. URL: https://www.ematop3.com/serverless-research-highlights.html.

[Unkb] Unknown. *docker*. URL: https://docs.docker.com/engine/reference/commandline/docker/.