A Provider-Friendly Serverless Framework for Latency-Critical Applications

Simon Shillaker
Imperial College London

ABSTRACT

Serverless is an important step in the evolution of cloud computing. First virtualisation enabled sharing a physical machine, then containers enabled sharing an operating system, now serverless targets sharing a runtime. Serverless platforms allow users to build distributed systems from individual functions, knowing little about the underlying infrastructure. They are free from concerns around configuration, maintenance and scalability. Meanwhile providers guarantee timely execution in a secure, isolated environment with pay-as-you-execute billing.

Although many prominent serverless platforms exist there are still several significant open problems. The most notable is the highly variable performance seen by users, which goes hand-in-hand with unpredictable resource consumption seen by the provider. Another sticking point is the almost universal lack of statefulness, which precludes many applications.

Containers are used extensively to provide an isolated environment for serverless functions, but introduce undesirable overheads and obstruct sharing resources. I plan to build a new multi-tenant serverelss language runtime focused on low overheads and resource reuse, using lightweight isolation built into the runtime itself. This has potential to greatly reduce variance in latency, improve resource efficiency and allow for smarter scheduling decisions. I will also investigate native support for stateful functions backed by distributed remote memory, hence opening the door to previously unfeasible data-intensive applications.

BACKGROUND

Serverless computing raises the level of abstraction such that a user is totally decoupled from the execution environment. In so doing, it makes distributed systems more accessible, flexible and potentially much cheaper. The associated programming model of small, parallelisable functions fits perfectly with microservice architectures and has potential applications in big data processing [1]. The technology has further wide-ranging use-cases in mobile backends, internet-of-things and downstream data processing. In spite of its clear potential, serverless is yet to gain significant traction and several open problems remain.

An almost ubiquitous design decision is to run each individual serverless function in its own Docker container, as is the case in OpenLambda [2], OpenWhisk [3] and OpenFaaS [4]. This is easy to reason about and gives good isolation guarantees. Platforms can also take advantage of existing container orchestration frameworks like Kubernetes [5] and get load balancing, networking and container management features for free. Finally, containers are familiar to many users and mature tooling makes them easy to work with.

12th Eurosys Doctoral Workshop, April 23rd 2018, Porto, Portugal
.

An unfortunate downside to containers is their start-up overhead, contributing to the “cold start” problem in serverless platforms. A cold start occurs when a container is not available to service a request, and booting a clean execution environment causes a spike in latency and resource consumption. Container boot times can be of the order of hundreds of milliseconds [6], an order of magnitude higher than the duration of many requests in latency-critical systems. This unpredictable hit to response times makes implementing low-latency serverless applications unfeasible. This issue is covered in more detail in [2] and [7], especially in relation to very low throughput when users see a high percentage of cold starts.

Cold starts can be mitigated by up-front provisioning of resources, an approach taken in OpenFaaS [4]. OpenFaaS keeps a single container running for every function that’s deployed, hence solving part of the problem but adding overhead for idle users. This approach does not mitigate the other form of cold start experienced when the system scales horizontally.

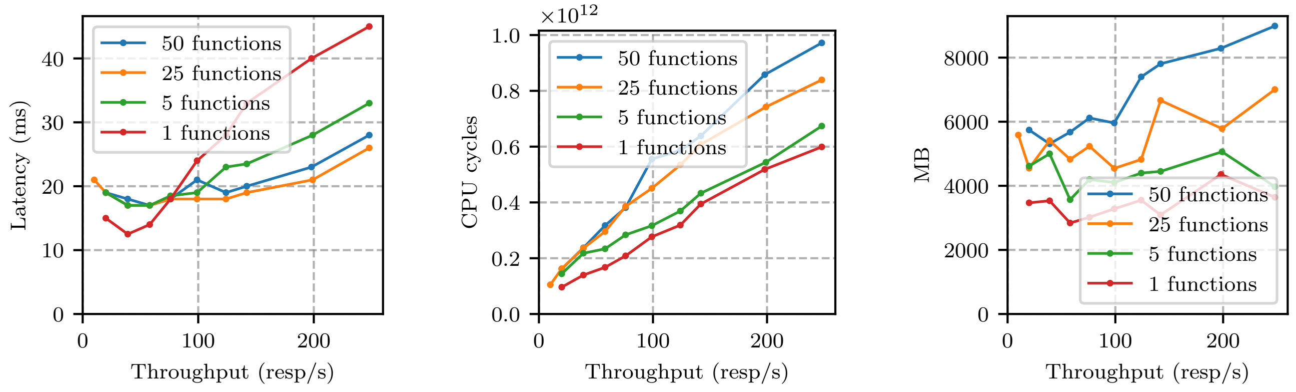
By isolating functions in their own individual containers we lose much of our ability to share and reuse resources. This leads to rapid growth in overheads as more functions are deployed. Serving thirty requests a second to a single function may require a single container, conversely thirty functions each receiving one request a second requires thirty separate containers yet may be performing an equivalent amount of work. When functions have a common runtime or similar dependencies, this hard isolation seems wasteful and there is much to be gained from alternative approaches that enable sharing and reuse.

Another interesting problem in serverless computing is that of shared state. Most serverless platforms only support stateless functions, requiring users to provide and manage external storage to share data between requests. This rules out most data-intensive applications as I/O to this external storage on every function invocation is prohibitively time consuming. Running data-intensive applications on OpenLambda is dealt with in PyWren [1] which uses a combination of system tweaks and external storage to achieve good performance and elastic scaling. Unfortunately this is still unfeasible for the average user and a long way from a fully-fledged big data environment. Preliminary attempts have been made to build stateful serverless frameworks such as AWS Step Functions [8] and Apache Openwhisk Composer [9], but these are still in their infancy.

INVESTIGATION

Openwhisk

Apache OpenWhisk [3] is a prominent open-source serverless platform built using the Akka framework [10]. Users are able to submit functions written in several supported languages to be run on demand or according to triggers. If their desired language is not



(a) Round trip latency **(b) Total CPU cycles required to sustain throughput for 1 minute** **(c) Average memory required to sustain throughput**

Figure 1: OpenWhisk behaviour with increasing throughput spread across different numbers of functions

supported by default they are able to specify their own custom Docker images to execute their code. Functions have no resources assigned up-front, so a container with the required dependencies is created upon receipt of the first request. Containers are paused after processing a request and will be cleared away after a certain timeout. If a repeat request is made within this timeout, the container is unpaused and reused. If more frequent requests are made, multiple containers will be created to run the same function. When the system is overloaded, paused containers will be destroyed to make room for new ones. This approach is similar to that of OpenLambda which is discussed in detail in [7].

Scheduling is centralised and handled by one or more “controller” nodes which forward requests to “invoker” nodes where they are executed. The controllers will always try to route each function’s requests to the same invoker node to maximise the probability of a “warm start”. If the desired invoker is overloaded, the controller will pick another invoker which will begin creating more containers to run the function.

Investigation

Although Openwhisk is a popular high-profile framework, it seems that little existing work has focused on its performance. The following experiments were run on an OpenWhisk cluster with two invoker machines and one controller machine. Each invoker had an Intel Xeon E3-1220 3.1Ghz processor and 16GB DRAM. All functions were the same no-op written in Java (note that language choice made little different to the results). Invokers were allowed up to 64 containers in their pool, giving the system potential to run up to 128 containers concurrently.

Under very low load we observe high latency as containers are instantiated, used and cleared away on every request. This is same effect as discussed in [2] on OpenLambda.

Figure 1 shows OpenWhisk’s behaviour under moderate to high loads. All plots show system metrics at equivalent throughputs spread over different numbers of functions, i.e. for 10 functions each function is handling one tenth of the throughput and for one function a single function is handling all of the load.

In Figure 1a we see how latency increases with throughput. The increase for smaller numbers of functions is sharp as the system begins queueing requests. When the load is spread across more functions this increase is less steep as the system is able to spread the work across more containers. All requests at this level of throughput will be served by warm starts so latency is in the tens of milliseconds.

In Figure 1b and Figure 1c we see the change in resource requirements with increasing throughput. Figure 1b shows the total number of CPU cycles used by the invoker machines to sustain a given throughput for one minute. As is shown, this number increases significantly as we use more functions. Figure 1c shows the average memory requirements on the invoker machines and tells a similar story. The difference between memory requirements for different numbers of functions is significant, exacerbated by running a separate JVM in every container.

At a certain throughput invoker machines start hitting their container limit and are unable to scale out any further. When this happens Openwhisk evicts warm containers, hence forcing a higher rate of cold starts than normal. This behaviour causes a downward spiral and results in thrashing as demonstrated in Figure 2. Here we see the rate of requests submitted vs. the actual throughput achieved. At certain points the system can no longer handle the incoming requests and the response rate collapses. This is down to both the scheduling approach and the isolation imposed by containers.

RESEARCH TOPICS

Runtime

As outlined above, a container-based runtime has some shortcomings. Boot times are unacceptable for low-latency applications and total isolation precludes resource reuse. I will investigate isolation within the language runtime itself, focusing on running untrusted functions side-by-side with low overheads and minimal start-up

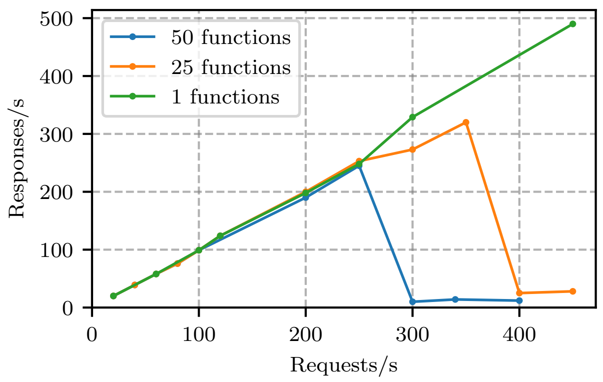


Figure 2: Frequency of responses received as throughput is spread over different numbers of functions.

times. The isolation mechanism will likely be based on OS abstractions, a common theme in existing literature [11–13]. To reduce boot times, support for language-specific dependency management and reuse is crucial. This topic is discussed in the context of serverless Python in [7]. The use of unikernels and fast-booting VMs to establish lightweight environments [6, 14] is interesting and something I will investigate. By building such a runtime I aim to reduce the magnitude of cold starts and allow for better resource management on the part of the provider.

Scheduling

Current open-source serverless platforms have fairly simple scheduling logic, lacking SLOs and offering minimal fairness guarantees. Most are scheduling short-lived containers with no benefits to colocation so are limited in how smart they can be. With the introduction of a runtime allowing resource sharing, it's possible to take advantage of colocation and dependency management when scheduling. By including these considerations and introducing stricter fairness, I plan to reduce latency seen by users while improving resource efficiency for the provider. Existing literature on scheduling cloud workloads, SLOs and fairness will be relevant here [15, 16] as will work on low-latency scheduling [17, 18].

Shared State

Lack of native support for state in serverless platforms puts a burden on the user whilst ruling out many use-cases, especially those handling large amounts of data. By bringing state management into the platform I will be able to investigate mutable distributed state, caching, colocation and their related programming abstractions. To achieve any goal in this area a suitable infrastructure for distributed remote memory is required. As discussed in [19], RDMA and faster networks have decreased the latency involved in remote memory by orders of magnitude, so systems like FaRM and RAMCloud [20, 21] could be extremely useful. Handling mutability, fault tolerance and synchronisation of system-wide state introduces many challenges already considered in the world of big data. An example of one approach is RDDs in Spark which provide both a programming abstraction and handle fault-tolerance [22].

CONCLUSION

Serverless computing is growing rapidly and quickly gaining acceptance as a new frontier of cloud computing. By tackling the problems of serverless runtimes, scheduling and shared state I plan to address several of the key open problems. Undoubtedly much complementary work will be done in the coming years and the space should evolve quickly into a fruitful area of research.

REFERENCES

[1] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. pages 1–8, 2017.
[2] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-dusseau, and Remzi H Arpaci-dusseau. Serverless Computation with openLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 33–39, Berkeley, CA, USA, 2016. USENIX Association.
[3] Apache Project. Openwhisk, 2016.
[4] Alex Ellis. OpenFaaS.
[5] The Linux Foundation. Kubernetes.
[6] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233. ACM, 2017.
[7] E Oakes, L Yang, K Houck, T Harter, A C Arpaci-Dusseau, and R H Arpaci-Dusseau. Pipsqueak: Lean Lambdas with Large Libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400, jun 2017.
[8] Amazon. AWS Step Functions.
[9] Apache Project. Openwhisk Composer.
[10] Lightbend. Akka Framework.
[11] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An {OS} Abstraction for Safety and Performance. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 49–64, Savannah, GA, 2016. {USENIX} Association.
[12] Y Chen, S Reymondjohnson, Z Sun, and L Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, may 2016.
[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *NSDI*, 2008.
[14] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-In-Time Summoning of Unikernels. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 559–573, Oakland, CA, 2015. {USENIX} Association.
[15] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 153–167, New York, NY, USA, 2017. ACM.
[16] Pawel Janus and Krzysztof Rzadca. SLO-aware Colocation of Data Center Tasks Based on Instantaneous Processor Requirements. *arXiv preprint arXiv:1709.01384*, 2017.
[17] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 97–110, New York, NY, USA, 2015. ACM.
[18] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.
[19] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 121–127, New York, NY, USA, 2017. ACM.
[20] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*. USENIX âĂŞ Advanced Computing Systems Association, apr 2014.

[21] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1—-7:55, aug 2015.

[22] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, page 2. USENIX Association, 2012.