FaasIn: Bringing UNIX pipes to serverless cloud function pipelines

Keywords: cloud, Function-as-a-Service, systems programming, containers, performance

Context

Cloud computing, or in other words, renting remote computing resources, is the mainstream way of deploying any application. This deployment model had a feedback effect on the applications' *architectures*: they are more and more "cloud native". The newest paradigm in cloud applications is Function-as-a-Service (FaaS): an application's features are served by chaining and replicating elementary functions.

However, *chaining function executions in today's FaaS platforms is not efficient* [1]–[4]. It is true that the control plane of the platform is fast enough to trigger the execution of the next function as soon as the previous one terminates within hundreds of milliseconds [3], [5]. Nonetheless, *passing output data is extremely slow*. Indeed, the Serverless nature implies that all data, in order to persist beyond the execution of one function, must be copied out of the function that produced it, and written to an external service (be it block or object storage, message queue, etc.) [2], [5]. Data movement between the FaaS platform and this remote service is slow by nature [1], [4], and constitutes a big drawback to porting many applications to the FaaS model.

For instance, a video processing pipeline requires moving chunks of video between many software components. Each of the latter lends itself to the FaaS paradigm; but the data movement in the FaaS platform becomes prohibitively expensive (around 30% of any function execution time) [4], whereas in a non Serverless setting, data would be *locally streamed* from one software component to the other. Another example is a chain of functions that all execute very quickly, and produce small intermediary results. Those could easily be kept locally, or even sent directly as argument to the next function in line, but this is impossible in current FaaS platforms. For the former alternative, only a remote service can offer such persistence, which comes with a *latency penalty* that is disproportionate in the face of the size of the data and the latency of execution of the next function. As for the latter alternative, passing the data as arguments – on which there is most often a size limit anyway [3] – to the next function requires *marshaling*, imposing a CPU and latency overhead.

Related work

Previous works are mostly of two kinds: automatic smart scheduling policies, and optimized data passing layers. Other works outside of this categorization only focus on specific applications and apply optimizations limited to their domains [6].

Of the first kind are Faastlane [3] and SONIC [2]. Faastlane automatically repackages functions in a workflow to leverage thread locality when possible, whereas SONIC leverages machine-learning to chose between data passing strategies depending on the size of the data and on the workflow topology: local storage, direct passing or remote storage. In essence, this first kind of solution is about *maximizing data locality through scheduling*, which requires heuristics to balance between efficient usage of the platform's resources and inter-function latency.

Of the second kind are Crucial [1] and SAND [4]. Crucial is an example of trying to work around the stateless property of the FaaS, and it does so by implementing a shared memory layer across the data-center. As for SAND, it implements a hierarchical message bus to provide both for very fast local message passing, and slower communication at the data-center level. Another example is Pocket [7], that implements a highly scalable data store, thus trying to solve the problem at the remote storage layer. A final example of optimized data passing layers comes from Faasm [5], that implements functions as threads of a WebAssembly (WASM) virtual machine; by doing so, it also proposes shared state in the form of shared byte arrays. A summary of this kind of solution, is they propose *solutions to the performance problem of the communication layer*, be it through networking and state passing, or through storage.

In contrast, the goal of this project is to implement a solution that is an optimized data passing layer, and that takes advantage of function locality while remaining efficient without locality, thus eschewing the need for scheduling heuristics.

Goals

From the previous examples, it follows that a good solution is *to keep the data local*, in a way that the next function does not have any processing to do to get it (i.e., no marshaling). Accounting for distributed data will be dealt with later.[1]

The goal of the internship is to explore the following new way of passing data to the next function in chain: to t*ransfer data between functions by streaming through a local in-memory pipe*, as illustrated in Figure 1. With this system, a chain of two functions A then B, is executed by:

1. creating the pipe between A's output and B's input;
2. running A: pass parameters, set one end of the pipe as an output stream in its context;
3. running B: pass parameters, set the other end of the pipe as an input stream in its context;
4. terminate A: retrieve its return value and close its end of the pipe when it terminates;
5. terminate B: retrieve its return value when it terminates (may not be when the pipe closes)

This is very similar to UNIX pipelines of commands: when a command line such as `grep pattern | wc -l` is executed, the standard output stream (`stdout`) of the command `grep` is piped (i.e., sent through a pipe located in local memory) to the standard input stream (`stdin`) of the command `wc`. Note that both commands are written with the expectation that they receive input to process on `stdin`, and they send output to `stdout`.[2] The implementation will indeed try t*o mimic the semantics of reading from and writing to pipes* to lower the usage barrier. Thus, the envisioned implementation will provide functions with new file descriptors that correspond to the input and the output of their step in the pipeline. The input stream from the function chain is called Function-as-a-Service input (`faasin`), and the output stream is called Function-as-a-Service output (`faasout`); thus the name of the project: FAASIN (pronounce "phase in").

1 There is already a promising lead.
2 Although in this specific example, both `grep` and `wc` can read their input from files passed as arguments.

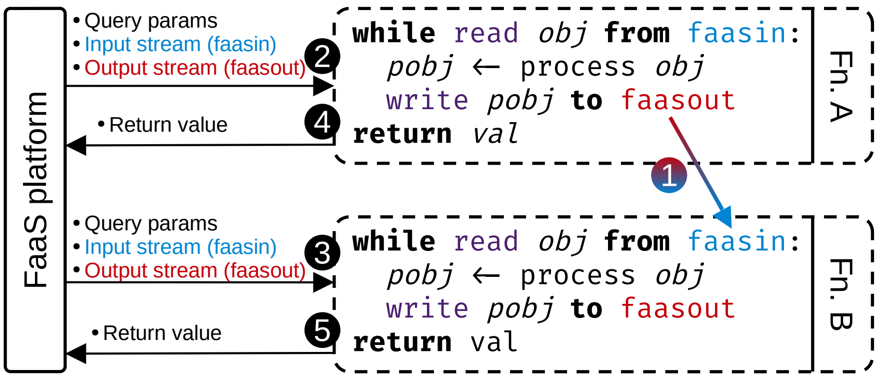

Figure 1: Illustration of offering UNIX pipes to FaaS functions.

Current serverless functions are not platform-agnostic, so requiring changes to the programming API (here, to read and write to different file descriptors) is acceptable. In addition, serverless platforms most often provide their own language runtimes (Java, Python, NodeJS, etc.) which means that it is also acceptable to change the runtimes or adapt the shims used by the platforms (here, to open `faasin` and `faasout`). At the platform level, functions are most often run in containers, such as Docker. In this case, they provide isolation by running different functions (and functions instances) in different processes. So any implementation of this mechanism may also require *modifying the container engine to pass `faasin` and `faasout` around*.

Note that this is *different from allowing direct communication* from one function to the next through a networking layer using sockets. First, direct communication between functions is usually not permitted because executing a function only creates a nameless, interchangeable instance. Second, going through the network and using sockets requires *marshaling*, which has already been described before as a consequent CPU and memory penalty. In the case of FAASIN, only raw bytes are sent and received, so no marshaling is required. Zero-copy data transfer could also be studied.

For now, the steps and goals of the internship to implement FAASIN are:

- implement the general mechanism of opening `faasin` and `faasout` in Docker containers;
- implement it at the function runtime and at the function programming language level;
- implement it in a FaaS platform;
- demonstrate the performance and programmability gains on real-world applications.

Perspectives

While not realistically a part of this project, further leads will be worth exploring in the future:

- **Cross-node data passing:** a clear limitation of this project is that it forces scheduling all functions of a chain on the same host node. It comes from the technical implementation itself: in-memory pipes. However, this same technique can be used to implement cross-node data passing thanks to Remote Direct Memory Access (RDMA), which essentially allows to locally map remote memory areas, in an extremely efficient way latency-wise.
- **Zero-copy:** FaaS is felt to be a very good fit for big data processing. In this setting, copying data when not necessary degrades performance, so further work could explore zero-copy implementations of this streaming mechanism.

The supervisor is Mathieu Bacou (mathieu.bacou@telecom-sudparis.eu).

Bibliography

[1] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, “On the faas track: Building stateful distributed applications with serverless architectures,” in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 41–54.

[2] A. Mahgoub *et al.*, “SONIC: Application-aware Data Passing for Chained Serverless Applications,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[3] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, “Faastlane: Accelerating Function-as-a-Service Workflows,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[4] I. E. Akkus *et al.*, “SAND: towards high-performance serverless computing,” in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USA, Jul. 2018.

[5] S. Shillaker and P. Pietzuch, “Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Jul. 2020.

[6] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, “Sprocket: A Serverless Video Processing Framework,” in *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, Oct. 2018, pp. 263–274. doi: 10.1145/3267809.3267815.

[7] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: elastic ephemeral storage for serverless analytics,” in *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*, USA, Oct. 2018, pp. 427–444.