Location, Proximity, Affinity – The key factors in FaaS

David Haja[1], Zoltan Richard Turanyi[2] and Laszlo Toka[3]

***Abstract*—The Function-as-a-Service paradigm emerged not only as a pricing technique, but also as a programming model promising to simplify developing to the cloud. Interestingly, while placing functions across hosts under the service platform is believed to be flexible, currently the available platforms pay little attention to co-locate connected functions, or data with the respective processing function in order to improve performance. Even though the local function invocation and data access might be an order of magnitude faster than their remote intra-cloud counterparts. In this paper, we therefore propose a Function-as-a-Service platform design that reaps the performance benefits of co-location. We build the platform on WebAssembly, a secure and flexible tool for efficient local function invocations, and on a distributed in-memory database, which allows arbitrary data placement. On top we advocate smart placement strategies for function executions and data, decoupled from the functions. Hence we envision good horizontal scaling of functions while keeping the experienced processing latency to that of a single machine case.**

***Index Terms*—Function as a Service, FaaS, WebAssembly, platform, runtime, co-location, performance**

I. INTRODUCTION

In recent years Function-as-a-Service (FaaS), often referred to as serverless computing, has become one of the popular paradigms in cloud computing. Numerous projects managed by companies and academic institutions offer FaaS services, such as Amazon's AWS Lambda [1], Apache OpenWhisk [2] Google Cloud Functions [3], Microsoft Azure Functions [4]. Using FaaS, developers do not need to care about resource allocation, scaling or scheduling, since the platform handles these. Most of these platforms operate with container technologies; the user's executable code is packed into a container that is instantiated when the appropriate function call request first arrives. With this relatively lightweight technology it is easy to arrange process isolation and resource provisioning.

FaaS platforms and solutions gain more and more attention in the Infocom domain, and researchers investigate its usability for the backend of e.g., tactile Internet applications, and edge computing-based applications. While its usage for telco services requires massive performance as well, the low end-to-end delay of such platforms is an absolute minimum in general. For example the deployment of a multi-party Augmented Reality application in an edge computing infrastructure with a FaaS platform on top provides excellent context for scaling ephemeral functions to the need of the users, but this use case also demonstrates the necessity for low latency in processing information shared between the users.

Even though communication is significantly faster the closer the parties are (e.g., in same data center, same rack, same server machine, same process), currently available FaaS platforms miss to co-locate entities that often communicate with each other. Furthermore, most of the available FaaS platforms suffer from cold-start latency when a new virtual machine (VM) or container has to be launched. The size of the image to mount, the programming language used, the number of libraries and dependencies all have an impact on this latency.

WebAssembly [5] (or Wasm) was announced in 2015, designed by the World Wide Web Consortium (W3C) for enabling high-performance applications inside browsers [6]. Since then a new industry partnership, the Bytecode Alliance [7], has been formed by Mozilla, Fastly, Intel, and Red Hat. The alliance's purpose is to implement standards and to propose new ones that decouple WebAssembly from JavaScript and make runtimes outside of the browser feasible. WebAssembly can essentially provide the same security capabilities and language independence as containers, but it allows for lighter composition in terms of startup and function call.

We argue that the right placement of function executions and data helps FaaS platforms to reap performance benefits of co-location. We therefore propose a novel FaaS platform design based on WebAssembly for isolation and fast local function calls; a distributed in-memory datastore to allow data mobility; and smart location selection strategies to co-locate data and function execution as the main novelty of our presented platform design. The contribution of this paper is twofold: i) we suggest to use WebAssembly as an emerging virtualization technique; ii) we consider advanced strategies for both determining the location of user functions and data. Furthermore, we define the role of each system component and major capabilities that those components need to provide. We also present the benefits, identify the challenges and missing capabilities that need to be defined for a complete system.

This paper is organized as follows. In Section II we present the related state of the art. In Section III we introduce WebAssembly, present the activity that allows WebAssembly to operate outside the browsers, highlight the benefits and the challenges of using WebAssembly in FaaS platforms. In Section IV we introduce a novel FaaS system design based on WebAssembly and locality awareness. We conclude the

[1] MTA-BME Network Softwarization Research Group, Budapest University of Technology and Economics (e-mail: haja@tmit.bme.hu)
[2] Ericsson Research, Hungary (e-mail: zoltan.turanyi@ericsson.com)
[3] MTA-BME Network Softwarization Research Group, MTA-BME Information Systems Research Group, Budapest University of Technology and Economics (e-mail: toka@tmit.bme.hu)

DOI: 10.36244/ICJ.2020.4.3

paper in Section V. All existing and missing features of WebAssembly mentioned throughout the paper reflect the status at the time of writing this paper.

II. RELATED WORK

One of the most popular FaaS computing platforms is Amazon's AWS Lambda [1]. Since the appearance of Amazon's AWS Lambda, numerous research initiatives [8]–[11] have improved their FaaS platform performance, with advanced dependency loading and caching, grouping functions related to the same application within the same container, or co-locating functions with their data. None of these works consider co-locating functions that communicate with each other and they all consider VMs or containers as virtualization technology.

Faasm [12] is a novel serverless runtime that executes distributed stateful serverless applications across a cluster. Faasm uses Faaslets, which extend the traditional WebAssembly modules with custom code that provide a minimal serverless-specific POSIX environment to support a host interface. The authors propose a two-tier state architecture: i) the local tier provides in-memory sharing; ii) the global tier supports distributed access to state across hosts. In contrast to our proposal, Faasm is a runtime implementation, it is designed to integrate with existing serverless platforms and it relies on the underlying platform's scheduler, e.g., Knative's [13] scheduler. Therefore their function scheduling strategy does not take into account the functions' behavioral patterns. Furthermore, the authors do not consider moving the data.

Cloudflare's Workers [14] service has recently started to support creating and hosting FaaS functions compiled to Wasm binaries. A novel FaaS platform is presented in [15] that uses WebAssembly and operates on an edge computing environment. AccTEE [16] is a sandbox solution that offers remote computation service with resource accounting, leveraging on two technologies: hardware-protected trusted execution environments, and WebAssembly. All of the platforms [14]–[16] are limited in the sense that they use JavaScript runtime with a wrapper JavaScript program that calls the binary instances, and returns their results. aWsm [17] is a native Wasm compiler and runtime that can operate under a serverless system. It enables Wasm shared-objects and multiple functions and their invocations within a single process.

The significant difference between all the aforementioned platforms and our design is that our design takes into account the frequency of the communication between functions and data, and strive to provide co-location for them.

III. WEBASSEMBLY FOR FAAS RUNTIME

WebAssembly defines a binary code format that is portable and efficient in size and execution speed on modern CPUs. It was originally created for executing programs in web browsers, written in languages other than JavaScript, e.g., C, C++, Rust.

A. Background

A *module* in WebAssembly [18] is the distributable, loadable and executable unit of code. WebAssembly modules are language-agnostic, which means a software developer may write its source code in any optional high-level language, such as C++, Rust, or Go, which then can be compiled, with the appropriate toolchain, to a portable binary that runs on a stack-based virtual machine.

The first technology for compiling C or C++ codes to WebAssembly binaries was Emscripten [19]. Emscripten's toolchain relies on LLVM [20] for the following key features: translating high-level code from languages (like C++) to an intermediate representation (IR), optimization and also dead code elimination. Beside Emscripten, numerous other toolchains have been developed and published for compiling other language source codes beside C and C++ to WebAssembly binaries. An example view of the compilation process is presented in Fig. 1.

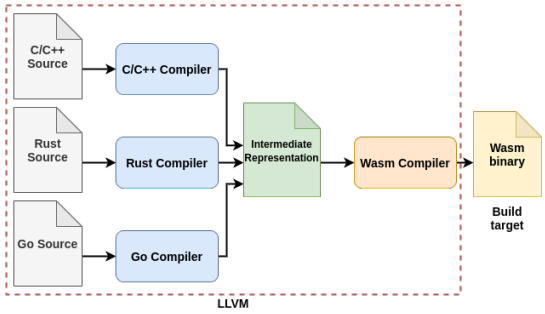

Fig. 1. Compilation to Webassembly with LLVM toolchain

One of the key features of WebAssembly is its secure execution environment. Unlike a native binary component, WebAssembly modules have access only to a part of the process memory, allowing secure isolation within a process. WebAssembly provides a sandboxed environment for the functions of a module, which by default do not have access to external APIs and system calls. To allow the interaction with anything outside the module, one has to explicitly authorize the module for the function or syscall. Taking all together, the pattern of the usage of these features is called WebAssembly "nanoprocess", which makes it possible to have similar isolation to that of a process, but with lower overhead. Another key feature is that the Wasm engine can copy directly between a caller and a callee's memories, even if they are separated in two modules and/or not compiled from the same language. This means that serialization and deserialization of communicated data may be avoided.

B. Outside the browser

WebAssembly was created as a browser runtime environment faster than JavaScript. During the compilation process Emscripten [19] created the Wasm binary and JavaScript glue code, which communicated with the browser and consequently with the API provided by the OS. This JavaScript glue code was not meant to be a standard or even a public interface. As WebAssembly became more popular and more powerful, the community realized its potential for use outside the browser. Consequently, standardization started to propose an

interface, the WebAssembly System Interface (WASI) that would connect Wasm binaries to regular operating systems. WASI allows runtimes to be independent of browsers, Web APIs and JavaScript. The execution of a WebAssembly module requires a runtime that supports the standardized WASI; multiple open-source implementations of such toolchains exist, such as Wasmtime [21], Lucet [22], Wasmer [23], WAVM [24], Wasm3 [25]. These various toolchains, proposed by the community and listed in Table I, differ in their source code language, compiler framework, or compilation process.

The two most common compilation techniques are Ahead-of-time (AOT) and Just-in-time (JIT) compilation. Most of the toolchains in Table I use Cranelift [26] and LLVM as their compiler framework. Cranelift is a low-level code generator that translates a target-independent intermediate representation into various executable machine code. LLVM is a collection of modular and reusable compiler and toolchain technologies. The goal of LLVM is to provide a modern compilation strategy that is capable to support both static and dynamic compilation of arbitrary programming languages.

TABLE I
WASI COMPATIBLE TOOLCHAINS

	Source languages	Compiler framework	Compilation
Wasmtime [21]	Rust, C++	Cranelift, Lightbeam	JIT
Lucet [22]	Rust	Cranelift	AOT
Wasmer [23]	Rust, C++	Cranelift, Dynasm.rs, LLVM	JIT
Wavm [24]	C++, Python	LLVM	JIT
Wasm3 [25]	C	Custom	Interpreted

WASI is in the middle of a standardization process, but its two key design goals are already set: portability and security. In the current proposal [27], WASI consists of a modular set of standard interfaces, one of which is called *wasi-core*. Wasi-core has a similar feature set as POSIX, so it contains the very basic interfaces that functions need, like random numbers, files, network connections, etc. Although wasi-core will not implement all features of POSIX, those missing can be handled by other modules inside WASI. This way the platforms can decide, which functionality they want to use.

C. Interface types

The Minimum Viable Product (MVP) of WebAssembly [28] defines only numbers as data types. Interface types define a set of types that describe abstract, high-level types. It gives the possibility of describing complex values, e.g., strings, sequences, records, and variants, without committing to a single memory representation or sharing scheme. The complex value descriptions are mappings between multiple sets of basic types to the abstract types, where these mappings are not hardcoded in the engine, instead, a module comes with its booklet of mappings. Most of the time the compiler takes care of this information, by adding a custom section that holds the interface types, to WebAssembly modules. In cases when two Wasm modules communicate, they both give their booklets, which define how they map their functions' types to the abstract types. This allows to automatically generate code to convert between value representations of the same type in an extensible and efficient manner.

D. Benefits for FaaS platforms

Portability: Since WebAssembly is machine agnostic, the underlying operating system and processor architecture, e.g., x86, ARM, is irrelevant from the users' perspective and only one compiled binary is needed even in case of a heterogeneous server park. This portability extends even to the clients via integration of WebAssembly into the browser. This opens up the possibility of extending a FaaS system to the client.

Language agnosticism: There are numerous open-source projects [29] that make several programming languages compilable to Wasm binary. The service provider needs only one runtime implementation for all compatible source code languages, and above all, developers can enjoy the programming language of their choice. Furthermore, with WebAssembly, users can easily make function calls across languages.

Security: WebAssembly provides function invocation in a secure way by default, as the design of modules contains isolated memory and system call sandboxing, which prevents buffer overflow type of security exploits or sensitive data leaks. This, coupled with access control of the databases, makes it easier to rely on third party, untrusted code.

Low execution overhead: WebAssembly uses nanoprocesses as a virtualization technique, to provide isolation and safety during the execution of Wasm modules. This results in less computational overhead provisioned for function execution than any other technique [30].

Reduced and reliable cold-start latency: Cold-start latency is not negligible in FaaS type services. It has been shown that using Wasm modules for function invocation reduces the cold-start latency [15], so it also reduces job completion times.

Fast communication between functions: Since WebAssembly will support shared memory [31], the communication between functions on the same host can be fast and efficient, reducing serialization, value transformations to the minimum.

E. Gaps and challenges

As all technologies, WebAssembly also has its tradeoffs. Here we present the general gaps that stem from its use.

The execution of Wasm functions is slower than executing the same function natively [32]. Although the cold-start latency can be reduced significantly compared to container-based systems, the slower execution may erode that gain.

For letting Wasm modules reach host resources, like the file system, one must provide the necessary capabilities to the executor explicitly, i.e., users should provide all the capabilities and access rights that their applications require. In addition, users have to provide bindings to other Wasm modules manually.

Interface types allow modules to use more complex types than numbers. This concept is under construction, so it can change as the community moves forward. Furthermore, at the

time of writing, interface types are only available for Rust programming language.

One of the biggest promises of WebAssembly is its standardized form that gives compilation target for many programming languages, although some of the source languages are not yet supported with WASI, e.g., Go [33]. Furthermore, the interpreted languages like Python, cannot be compiled to Wasm binary directly. The workaround that currently allows compiling programs written in such languages is to compile both the interpreter VM with the user's functions together in one Wasm binary. When one has many functions, this may be a significant overhead.

Currently features, like threads, exception handling, Single Instruction Multiple Data, shared memory [31], are only planned for WebAssembly [34]; their status varies between the proposal and the feature standardization phases. Some of these capabilities are indispensable features for users' functions. They are still changing and uncertain when and how they will be supported by standards.

Often one wants to execute numerous function instances in a distributed fashion, therefore the modules demand a method that provides the safe communication ability between each other. A major conceptual challenge is to provide a secure and efficient way for WebAssembly modules on different hosts to communicate through the network.

IV. Advanced WebAssembly-based FaaS design

In this section we present our envisioned FaaS system architecture: we define each component's functionality, and we emphasize the importance of the location of user functions' execution within a cluster. Throughout this paper the user denotes the application/function developer and provider that realizes the application and wishes to execute it in the platform.

A. Motivation for co-location

The main goal of our proposed system design is to enable FaaS functionality for users, with both horizontal scaling, and with small function composition and data access overhead, while retaining the flexibility and security of the currently available platforms. Our key observation is that local data access and function invocations within the same server are typically an order of magnitude faster than remote ones [35], [36]. Some illustrative overhead values can be found in Table II. Naturally the overhead depends heavily on the underlying system and network characteristics, although it is visible that they are not negligible compared to a function initialisation time. In the table we list data access times measured in DAL [35] and in S6 [37], two distributed in-memory key value stores. Also, we depict the range of function invocation latencies with Faasm [12], [17] and with gRPC [38], [39], which is a high-performance, open source universal RPC framework.

One can see that the overhead of remote operations can be heavy, especially if they have to be repeated several times. Since the novel applications and services usually require multiple instances of the same function, and numerous different

TABLE II
FUNCTION CALL AND DATA ACCESS OVERHEAD WITHIN A DATA CENTER

	Overhead
Wasm function initialisation	$185\mu s - 5ms$
Local data access (DAL)	$1\mu s$
Remote data access (DAL)	$20\mu s$
Local data access (Redis)	$70 - 90\mu s$
Remote data access (Redis)	$70 - 90\mu s$
Local data access (S6)	$0.16\mu s$
Remote data access (S6)	$18\mu s$
Wasm - Wasm call locally (Faasm)	$235\mu s - 2.9ms$
gRPC call (between GCE VMs)	$74 - 629\mu s$
gRPC call (over Ethernet 40G)	$80\mu s - 6ms$

functions that constitute a long pipeline, we cannot ignore the extra overhead that these operations incur. Therefore, to accomplish our performance goals, we maximize the locality effect by co-locating functions with their data, and co-locating caller and called functions.

B. Architecture

We want to hide the distributed nature of the platform infrastructure from the application developer, and make the assumption on the cost of data access and function invocation to be in the order of what is typical in a single-machine application. WebAssembly helps this endeavor by offering low-overhead function calls between sandboxes. Our main architectural choices are: i) functions are compiled to Wasm binaries, uploaded and reachable from every worker host; ii) functions are stateless and store their data in a separate, distributed in-memory database accessible from anywhere in the system; iii) data in the database is automatically moved around to minimize access latency; iv) a function may invoke another functions locally or remotely, the choice is based on performance considerations.

We present the components of our FaaS system architecture, and a high-level overview is depicted in Figure 2.

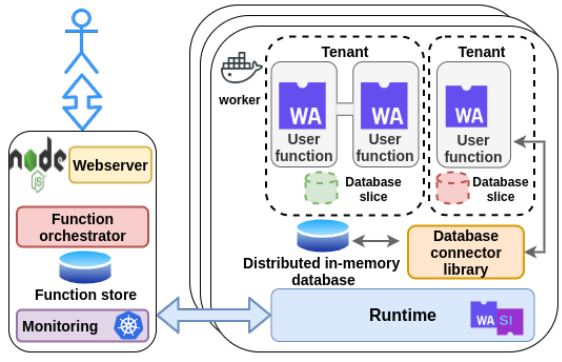

Fig. 2. Architecture of our envisioned FaaS system design

User functions: They realize the application that users wish to execute. The user can develop functions in any source language that can be compiled to Wasm binary, then the system stores only the compiled modules and executes them upon request. User functions can be invoked externally via a HTTP API, by another functions' direct calls (synchronously,

asynchronously or as co-routines) or by assigning functions to trigger events, such as a change of a database item, an error (e.g., function failing) or another system event, like scaling.

Worker: Contains and manages the system components used for user functions' execution and communication. The workers can be either physical or virtual machines. We can use Kubernetes [40], as the underlying cluster infrastructure manager, where the pods represent our worker nodes.

Runtime: The WASI-compatible runtime is responsible for the execution of the User functions, memory management, sandbox creation and job scheduling. It handles code signature validation, other Wasm module dependencies and function invocations, either locally, or across different workers. It is also responsible for authorizing the sandboxes appropriately, e.g., whether functions may access certain parts of the database, files, network or some static data in a shared filesystem. We assume that all uploaded functions (and all their dependencies) are available (eventually) at every worker. This means that every worker can execute any function, allowing for execution locality for each invocation independently. For example, function *F* working on data *D1* and *D2* may be invoked at different workers depending on the location of *D1* and *D2*. Consequently, there are no resources allocated on a per function basis - most prominently functions do not get their own containers. This removes the need to manage per-function resources (e.g., scaling them) and makes it lightweight to trade resources between functions (just by making workers execute different ones). Instead, resources are managed on a per tenant (or per application) basis.

Distributed in-memory database: Since User functions are stateless, a distributed database, e.g., a key-value store, provides state sharing between function invocations. Functions may have restricted access to (parts) of the database for security and modularization purposes. The realization of this feature is crucial, as due to the stateless nature of the functions, access to the database can be a bottleneck. Databases [35], [37] that allow controlling the location of items and offer optimized local access are best suited to our architecture, as they help exploit the benefits of locality. Note that there may be more than one kind of database integrated into the system offering different semantics. Besides simple key-value stores, strongly transactional databases, conflict-free replicated data types, graph databases or any kind of distributed database can be integrated.

Database connector library: Provides connection between the User functions and the Distributed in-memory database. It offers an API directly accessible from the User functions.

Tenants: They represent the users of the system. We provide tenant isolation that prohibits users accessing binaries and data of other tenants. It also involves careful management of CPU and memory resource usage, both of the Runtime and of the Distributed in-memory database. The database slices, in terms of resource usage, are depicted with dashed line contoured database icons in Figure 2.

Function orchestrator: This manages User function execution and data locality. Since at function execution there is very little time to make a decision, a set of easy to evaluate rules must be applied to decide on the place of executions. The task of the Function orchestrator is to observe system behaviour, i.e., cross-function invocations and data access patterns, i.e., which function executions invoke what other functions, and access what data, respectively, and evolve the rules.

Webserver: Responsible for handling external requests through an HTTP API. It authorizes incoming requests, processes and invokes the appropriate User functions through the Runtime component. After the invocation, the Webserver is responsible for sending the results back to the client. The tenants may also expose (parts of) the database through this component. A general HTTP API implementation, like Node.js, is applicable for the functionality our platform requires. The performance improvement of our platform stems from the reduced communication overhead on the frequent calls between functions and between function and data. Since the Webserver does not take an active part in these communications, only in the rare user interaction, it may run on a dedicated node or on the Workers as well.

Function store: Tenants can upload and manage functions, modules, versions and aliases via this component. It is also responsible to distribute the uploaded functions to the Workers.

Monitoring: This component extends Kubernetes' cluster manager functionality with log collection, trace and alarming indicated by the User functions of a Tenant or application. It allows a unified view of the system and helps troubleshooting; it also collects performance metrics and counters.

C. Workflow

We present in Figure 3 the workflow of i) function development, upload; ii) invocation; iii) our Function orchestrator component in our envisioned FaaS platform.

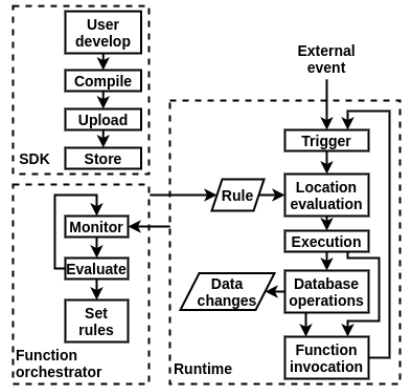

Fig. 3. Workflows of our platform

The workflow begins with the development of the user's function. As we stated before, a wider range of programming languages can be used than in the currently available FaaS platforms thanks to the WebAssembly ecosystem. After the development, either the system or the user compiles the source code to WebAssembly binary.

The binary and its dependencies will be uploaded and stored in the Function Store and distributed to all workers. Distribution may simply mean the publishing in a networked

file system. When a WebAssembly module is uploaded, a URL is assigned to it to make the function externally accessible.

The users can invoke their functions via HTTP requests through the assigned endpoints; authorization is managed by the webserver. The process of a function invocation is presented with the solid lines in Figure 3.

A function running may access the database through the connector library, reading and writing data. Such writes constitute the side effect of the function and may trigger other functions if they were assigned to the change of that key. The functions themselves may also trigger other functions directly. They may also fail (or throw an unhandled exception), which may also serve as function startup trigger. In any case the runtime makes a decision on whether the new function shall be locally or remotely executed. This decision is based on a few simple rules set by the Function Orchestrator. In case of remote execution, a network message is sent to the selected worker containing the name of the function to execute and its arguments. In case of local execution a WebAssembly sandbox may be created (if isolation requires it) for the function, the arguments are moved into the sandbox and execution is passed to it. The runtime may maintain a set of pre-allocated sandboxes, for performance reasons. After local or remote function execution, if the result of the function is relevant it is transported back to the caller, either as a syncronous return value or as a promise/future, or a callback argument or a co-routine yield, depending on what construct the source programming language supports.

The Function Orchestration needs to specify the rules for execution location by observing data access and function invocation patterns. E.g., for functions that typically execute quickly and do not access the database at all, it is beneficial to always run them at the node of trigger. Similarly, functions having modest compute requirements, little database access, but large input/output data sizes are best executed locally. In contrast, functions accessing a lot of data that is remote may perform better at the remote location.

Naturally, every invocation of a function may have a different access and compute pattern, thus a single rule per function will result in many bad decisions. Not only the arguments to the function, or that of its caller can be taken into account, but metadata supplied by the caller. For example, in a telecom system, which handles a lot of users concurrently, the identity of the user can be supplied for every function execution. This allows the system to observe data access and function invocation patterns per user and group all functions working on the same user.

D. Co-location aware function scheduling

In general, function scheduling is a hard problem with multiple constraints [41]. In our platform we consider the joint orchestration of user functions and the used data. The goal is to minimize the cost that comes from data movement, remote data access and remote function calls, as all the communication through the network and data serialization-deserialization put additional overhead on the execution of user applications. Nowadays, the most widely used distributed, in-memory databases, like Riak [42], can store multiple replicas for each data (or function state), but do not allow to define the host of the state or its replica. To control data locality we need a distributed, in-memory database that provides an API, which lets the service provider define the host of each replica.

For functions that typically execute quickly, invoke each other and do not access the database at all, the best strategy is to run them at the node of the trigger (locally). Similarly, functions having modest compute requirements, rare database access, but large input/output data are best placed locally. In contrast, functions intensively accessing remote data (hosted at a node different from the node of the trigger) may perform better at the remote location, in order to access data locally there, with minimum cost.

In our envisioned system the Function Orchestrator defines rule sets that summarize its knowledge about how functions behave. To construct an adequate placement strategies, we have to identify the functions' behavioral patterns. In addition to the trivial cases there are numerous more complicated ones that need more advanced strategies. For example, we want to place a pipeline on the same host with the data that it accesses, if its functions work on the same data. To accomplish this, the pipeline needs to be decoupled and the relations between its components must be identified.

In addition, every invocation of a function may have a different access and compute pattern, thus a single rule per function will result in bad decisions. On the other hand, quite many things may influence how a function executes (such as its input parameters and the state of the context it works on) and it is not realistic to consider the state of the entire database in rules. As a compromise, we could allow the caller of the function to provide hints (such as a hash of some ID relevant to the function) that can be made part of the rules. For example, in a telco system, which handles many users concurrently, (the hash of) the identity of the user can be supplied for every function execution. This allows the system to observe data access and function invocation patterns per user and group all functions working on the same user.

Scheduling functions and data jointly implies countless open questions that we do not cover in this paper, but it is already clear that identifying the appropriate patterns is crucial for effective scheduling. Finding the right mix of programmer input and machine learning is another area of open research.

When a function is invoked, the runtime can build a directed graph that present the communications relationship between functions and data. An illustrative graph is presented on Figure 4, where $G = (F, D, E)$; $F = \{Functions\}$; $D = \{Data\}$; $E = \{f_i- > f_j; f_i- > d_k; d_k- > f_j; |\forall f_i, f_j \in F; \forall d_k \in D\}$. A node in the graph can be a data or a function instance and the directed edges present a function invocation or data access from the caller/event to the called/data. The runtime can use postorder traversal to define each entity location by applying the actual set of rules on all nodes. This constructed graph is similar to an *extended service call graph* [43].

Evaluating the rules, the runtime makes a decision on whether the new function shall be locally or remotely executed. In case of remote execution, a network message is sent to the selected worker containing the name of the function

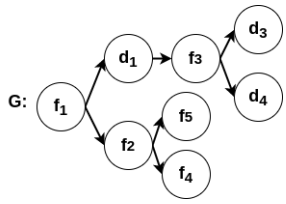

Fig. 4. Communications relationship

to execute and its arguments. In case of local execution, a WebAssembly sandbox may be created (if isolation requires it) for the function, the arguments are moved into the sandbox and execution is passed to it.

Previously, we have considered placing function executions, although data can be moved around in our system, as well. One simple strategy would be to move the data (and its replicas) to the locations with most accesses. This decouples function and data placement and considers cases in which data movement is cheaper than moving functions.

V. Conclusion

Although the appearance of WebAssembly outside the browser and of its system interface is recent, the community has already started to work on solutions that extends its usage to serve as a rapidly emerging virtualization technology under novel services. While exploiting its features of very light virtualization, available FaaS platforms do not take into account the co-location of functions, hence they neglect this potential source of performance increase. (Note that most microservice or service mesh tools also disregard locality: usually no effort is made to co-locate microservices often invoking one another.)

In this paper, we proposed a FaaS system design that offers horizontal scaling at the performance promise of host internal operation. The platform offers scaling to several server machines, when more compute power is needed. At the same time, through intelligent placement of data and function executions it minimizes the need for remote data access and remote function invocation. Not all application setups permit the reduction of remote operations under high-load, e.g., there are computing problems with mesh-like interaction of functions and data, in which case it is not possible to systematically reduce the fraction of remote operations. On the other hand, most of the computing problems exhibit clusters of often communicating functions and their data. These applications run faster in a FaaS platform considering locality.

The use of WebAssembly is instrumental in reducing the overhead of local function invocations. In addition, it offers security and portability benefits and is programmable in a wide range of languages. Although WebAssembly lacks some of the necessary features that are not implemented or defined yet, we presented how WebAssembly and its new extension WASI could fit in a FaaS system architecture. Beside the benefits, we also demonstrated the tradeoffs that will be encountered when we build the platform: the current challenges and gaps that need to be implemented before our vision becomes reality.

In our future work, we want to identify the function invocation and data access patterns based on the state-of-the-art. After the identification, we will compose and implement the Function orchestrator component that is able to construct those rule sets, and thus to improve the co-location of the functions, achieving better application performance than the available FaaS platforms.

Acknowledgments

Project no. 2018-2.1.17-TÉT-KR-2018-00012 and 135074 have been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary under the 2018-2.1.17-TÉT-KR and FK_20 funding schemes.

References

[1] *AWS Lambda*, https://aws.amazon.com/lambda/, Accessed on October 25, 2020.

[2] *Apache OpenWhisk*, https://openwhisk.apache.org/, Accessed on October 25, 2020.

[3] *Google Cloud Functions*, https://cloud.google.com/functions, Accessed on October 25, 2020.

[4] *Microsoft Azure Functions*, https://azure.microsoft.com/services/functions/, Accessed on October 25, 2020.

[5] A. Haas *et al*., "Bringing the web up to speed with WebAssembly," in *ACM SIGPLAN*, 2017. DOI: 10.1145/3062341.3062363.

[6] *WebAssembly Working Group*, https://www.w3.org/wasm/, Accessed on October 25, 2020.

[7] *Bytecode Alliance*, https://bytecodealliance.org/, Accessed on October 25, 2020.

[8] S. Hendrickson *et al*., "Serverless computation with openlambda," in *USENIX HotCloud*, 2016.

[9] I. E. Akkus *et al*., "SAND: Towards High-Performance Serverless Computing," in *USENIX Annual Technical Conference*, 2018.

[10] E. Oakes *et al*., "SOCK: Rapid task provisioning with serverless-optimized containers," in *USENIX Annual Technical Conference*, 2018.

[11] V. Sreekanti *et al*., "Cloudburst: Stateful Functions-asa-Service," *arXiv preprint arXiv:2001.04592*, 2020.

[12] S. Shillaker and P. Pietzuch, *Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing*, 2020. arXiv: 2002.09344 [cs.DC].

[13] *Knative: Kubernetes-based platform to deploy and manage modern serverless workloads*, https://knative.dev/, Accessed on October 25, 2020.

[14] *Cloudflare's Workers service*, https://workers.cloudflare.com/, Accessed on October 25, 2020.

[15] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019. DOI: 10.1145/3302505.3310084

[16] D. Goltzsche *et al*., "AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting," in *Proceedings of the 20th International Middleware Conference*, 2019. DOI: 10.1145/3361525.3361541.

[17] P. K. Gadepalli *et al*., "Challenges and Opportunities for Efficient Serverless Computing at the Edge," in *SRDS*, 2019. DOI: 10.1109/SRDS47363.2019.00036.

[18] *WebAssembly Specification*, https://webassembly.github.io/spec/core/index.html, Accessed on October 25, 2020.

[19] *Emscripten toolchain*, https://emscripten.org/, Accessed on October 25, 2020.

[20] C. Lattner, "Introduction to the llvm compiler system," in *Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy*, 2008.

[21] *Wasmtime: A small and efficient runtime for WebAssembly and WASI*, https://wasmtime.dev/, Accessed on October 25, 2020.

[22] *Lucet: Fastly's native WebAssembly compiler and runtime*, https://github.com/bytecodealliance/lucet, Accessed on October 25, 2020.

[23] *Wasmer: The Universal WebAssembly Runtime supporting WASI and Emscripten*, https://github.com/wasmerio/wasmer, Accessed on October 25, 2020.

[24] *WAVM: WebAssembly virtual machine*, https://github.com/WAVM/WAVM, Accessed on October 25, 2020.

[25] W*asm3: A high performance WebAssembly interpreter written in C*, https://github.com/wasm3/wasm3, Accessed on October 25, 2020.

[26] *Cranelift Code Generator*, https://github.com/bytecodealliance/cranelift, Accessed on October 25, 2020.

[27] *WASI: WebAssembly System Interface*, https://github.com/bytecodealliance/wasmtime/blob/master/docs/WASI-overview.md, Accessed on October 25, 2020.

[28] *WebAssembly Minimum Viable Product*, https://webassembly.org/docs/mvp/, Accessed on October 25, 2020.

[29] *Awesome WebAssembly Languages*, https://github.com/appcypher/awesome-wasm-langs, Accessed on October 25, 2020.

[30] *Building a secure by default, composable future for WebAssembly*, https://hacks.mozilla.org/2019/11/announcing-the-bytecode-alliance/, Accessed on October 25, 2020.

[31] *Multi Memory Proposal for WebAssembly*, https://github.com/WebAssembly/multi-memory, Accessed on October 25, 2020.

[32] A. Jangda *et al*., "Not so fast: analyzing the performance of webassembly vs. native code," in *USENIX Annual Technical Conference*, 2019.

[33] *wasm: support new WASI interface*, https://github.com/golang/go/issues/31105, Accessed on October 25, 2020.

[34] *WebAssembly Features to add after the MVP*, https://webassembly.org/docs/future-features/, Accessed on October 25, 2020.

[35] G. Németh *et al*., "DAL: A Locality-Optimizing Distributed Shared Memory System," in *USENIX Hot-Cloud*, 2017.

[36] M. Szalay *et al*., "Industrial-Scale Stateless Network Functions," in *IEEE CLOUD*, 2019. DOI: 10.1109/CLOUD.2019.00068.

[37] S. Woo *et al*., "Elastic scaling of stateful network functions," in *USENIX NSDI*, 2018.

[38] *gRPC Benchmarking*, https://grpc.io/docs/guides/benchmarking/, Accessed on October 25, 2020.

[39] R. Biswas *et al*., "Designing a micro-benchmark suite to evaluate gRPC for TensorFlow: Early experiences," *arXiv preprint arXiv:1804.01138*, 2018.

[40] *Kubernetes: Production-Grade Container Orchestration*, https://kubernetes.io, Accessed on October 25, 2020.

[41] E. Van Eyk *et al*., "A SPEC RG cloud group's vision on the performance challenges of FaaS cloud architectures," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2018. DOI: 10.1145/3185768.3186308.

[42] *Riak: Enterprise NoSQL Database*, https://riak.com/, Accessed on October 25, 2020.

[43] M. Obetz *et al*., "Static Call Graph Construction in AWS Lambda Serverless Applications," in *USENIX HotCloud*, 2019.

Dávid Haja is a Ph.D. student at Budapest University of Technology and Economics. He is a member of the High Speed Networks Laboratory (http://hsnlab.hu) at the Department of Telecommunications and Media Informatics. His main research interests include Edge Computing, Software-Defined Networking (SDN), Network Function Virtualization (NFV) and Resource Orchestration.

Zoltán Richárd Turányi received his M.Sc. degree in Computer Science from Budapest University of Technology and Economics in 1996. In 1997 he joined Ericsson's Traffic Analysis and Network Performance Laboratory (Traffic Lab). Since then he worked with various Mobile Core Network, Software Defined Networking and Network Function Virtualization projects within Ericsson research. Since 2014 he fills the role of 5G Network Architectures Expert within Ericsson Research.

László Toka is assistant professor at Budapest University of Technology and Economics, vice-head of HSNLab (http://hsnlab.hu), and member of both the MTA-BME Network Softwarization and the MTABME Information Systems Research Groups. He obtained his Ph.D. degree from Telecom ParisTech in 2011, he worked at Ericsson Research between 2011 and 2014. His research focuses on cloud computing and artificial intelligence.