Towards a Message Broker Free FaaS for Distributed Dataflow Applications

Patrik Fortier, Frederic Le Mouel, Julien Ponge

Towards a Message Broker Free FaaS for Distributed Dataflow Applications

Patrik Fortier
Univ Lyon, INSA Lyon, Inria, CITI, EA3720
Villeurbanne, France
patrik.fortier@insa-lyon.fr

Frédéric Le Mouël
Univ Lyon, INSA Lyon, Inria, CITI, EA3720
Villeurbanne, France
frederic.le-mouel@insa-lyon.fr

Julien Ponge
Red Hat
Lyon, France
jponge@redhat.com

***Abstract*—We present an extended implementation of Dyninka, a framework to prototype FaaS-based distributed dataflow applications. Its programming model gathers the definition and the composition of services within a single file using the *multi-tier programming* paradigm, and compiles them into multiple services to be deployed on cloud computing infrastructure. Our framework is built without a gateway or a messaging platform. Services communicate directly with each other within the cloud abstracted infrastructure. As a result, we emancipate ourselves from message brokers and reduce the network and computation overheads introduced by other FaaS frameworks such as OpenFaaS. We validated our approach on a Fog computing scenario with limited resources and several load profiles. Our framework shows better stability, throughput, and a reduced overhead compared to OpenFaaS.**

***Index Terms*—FaaS, Multi-tier Programming, Macro Programming, Distributed Systems, Dataflow**

I. Introduction

The *microservice architecture* (MSA) has become more popular over the last few years with the rise of cloud platforms such as *Amazon Web Services* or *Microsoft Azure*. This architectural style favors the development of multiple services where each one focuses on one functional concern and exposes one or several APIs over the network, typically using HTTP, web sockets, and messaging systems. An application is then the composition of microservices. In the current context of the Internet of Things (IoT), applications are now processing the large amount of data that sensors and mobile devices provide. The flow of information from IoT devices can be modeled by the dataflow model, where data is processed, stored, and exposed by distributed applications and where each service provides one operation to be performed on received data. Although the main benefits of this architecture are the scalability and reusability of services, MSA developers still need to provide boilerplate code like network communications, protocols, and data representation.

From the IBM definition [1], *Function-as-a-Service* is a cloud computing paradigm allowing one to execute code in response to events without the complex infrastructure typically associated with building and launching microservice applications [1]. These techniques also introduce the cost of running a function in their design since it is the pricing practice of all FaaS. Application services need to run as efficiently as possible and stop immediately after - to save money. With the rise of *edge computing* and *fog computing*, infrastructure is being shifted towards the edge of the network for latency purpose, at the cost of computing power scarcity since massive datacenters are not as available as in the backbone network. A cluster can then be composed of several edge clouds with computing power and storage, potentially synchronized with the rest of the cluster. The outcome is multiple clouds formed by several edge data centers. Frameworks for operating Kubernetes on edge and fog architectures like *KubeEdge* [2] have been developed. They provide core infrastructure support for networking, application deployment, and metadata synchronization between cloud and edge environments. Platforms like OpenShift [3] or Anthos [4] allow the unification of multiple cloud infrastructures into the view of a consolidated cluster managed by a higher level control plane. Other orchestration solutions like *Apache Mesos* are left out, although deployments actions between the two orchestrators are similar: create, modify, delete and observe deployments units. Cloud platforms provide tools to create services individually and connect them through a WYSIWYG[1] interface. Such tools are mostly specific to deployment infrastructure, but modern applications often use different cloud infrastructures to avoid bottlenecks.

Regular programming languages provide few abstractions of network communications and deployment to cloud infrastructures. We believe modern languages should implement means to make distributed programming easier.

Serverless use centralized message brokers to handle communication between services. Implementations are using either a middleware for communication. This approach is a performance bottleneck when the message broker is under congestion.

To solve this issue, we propose *Dyninka*, a function and language-based framework that provides multi-tier language features to create distributed dataflow applications and deploy them on any deployment infrastructure available to the compiler like Kubernetes and Apache Mesos. This language abstracts all network communication through function chaining and keeps properties of microservices, including separate business models and data ownership. The compiler reads the single mono-linguistic application codebase, generates separate entities for each compiled service, and deploys them to the targeted and available platforms. The solution keeps

[1]WYSIWYG is an acronym for "What You See Is What You Get"

978-1-6654-9350-5/22/$31.00 ©2022 IEEE
DOI 10.1109/FiCloud57274.2022.00009

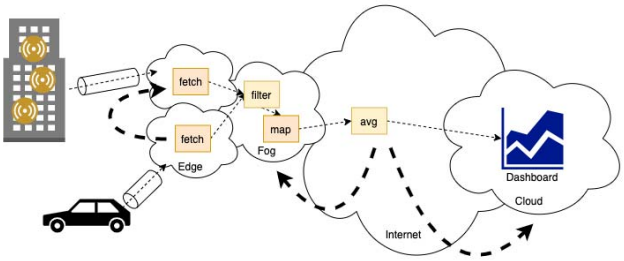


Fig. 1: Use case: processing smart building data

```
fun get(): SensorValue {}
fun SensorValue.filterTemperature(): Temperature{}
fun Temperature.computeAverage(): TemperatureValue {}
fun TemperatureValue.display(){}
fun main(){
  get().filterTemperature().computeAverage().display()
}
```

Listing 1: Dyninka example

the fundamental properties of functional programming, and multi-tier programming removes the developer's burden to handle service communication and non-business-oriented logic, keeping the application complexity low. Our contributions are the following.

- We enrich a programming language with deployment annotations. We provide a complete compilation and packaging chain that deploys the generated dataflow in an infrastructure-agnostic manner.
- We propose an alternative to traditional serverless message passing by generating direct service-to-service message routing.
- We implement a prototype of Dyninka based on Kotlin, targeting Kubernetes as a deployment infrastructure.
- We conduct experiments on a real-world dataset to compare the performance of Dyninka with OpenFaaS.

Our evaluation shows that Dyninka introduces a low overhead compared to serverless frameworks for representative dataflow applications. It can even outperform OpenFaaS on heavy traffic, running on a dedicated cluster with the same orchestration system, highlighting the bottleneck introduced by a central message broker.

II. Use-Case Application

Figure 1 and Listing 1 present the use-case application of the work. This simple program is a filter, map, reduce type application in the context of a smart building: a building is equipped with sensors, providing a whole set of information: weather, electricity consumption, and other metrics smart buildings can provide. We build an application that collects, filters, and maps information from sensors and computes a moving average of the building's temperature. This usecase allow us to compare Dyninka with solutions from the industry is a generic Fog computing configuration. Services are placed in nodes with limited resources where overhead introduced by the framework is more impactful on the performance. Operations offloaded to a remote cloud like previous value storage is excluded.

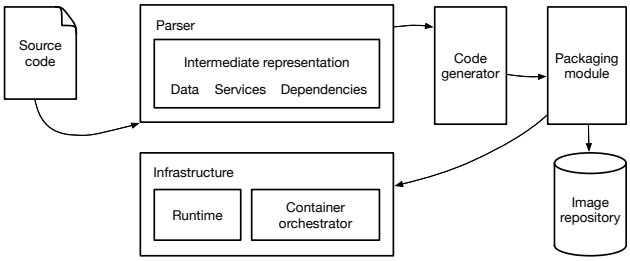


Fig. 2: Architecture of Dyninka

III. The Design of Dyninka

A. Overview of Dyninka

Dyninka is a framework designed to help developers create distributed dataflow applications for the Cloud from a single source file. It adapts the application deployment process to the appropriate infrastructure without specification by selecting from the available choice in the environment variables. By abstracting the deployment infrastructure, the developer focuses on the business logic rather than on the deployment process. Dyninka handles the building, compiling, and deployment of distributed dataflow applications. This paper focuses on the dataflow model and the deployment operations. Dyninka handles every aspect of the software building process. It parses the source codebase, generates the resulting code, and wraps each microservice up as deployment units ready to be run on cloud infrastructures. Deployments are then to be managed and modified by a Context-aware runtime (out of this paper's scope). The Dyninka approach promotes a point-to-point communication system instead of a centralized messaging system, more commonly found in FaaS Serverless approaches. Each service communicates directly with the next one, instead of handling messages or requesting them from a middleware.

B. Architecture

Dyninka consists of three modules.

1) A DSL to generate the dataflow application as deployable microservices. Its role is to identify each component of the dataflow and generate the associated code to create a distributed implementation of the application.
2) A deployment module to bundle and deploy in an infrastructure agnostic manner each service of the application. Its role is to identify the infrastructure the service is going to be deployed, and then send the proper commands to the container orchestration system.
3) A container orchestration runtime to manage the deployed applications, retrieve contextual data, and adapt deployments to constraints. Its role is to execute the received deployment operations and also to gather information about the state of our deployment on a service granularity and expose it.

Figure 2 gives a high-level overview of the system's architecture.

In the dataflow application generation module, an application is written in the form of a function composition where each function represents a microservice and their chaining is the abstraction of communications between them. It requires a functional language to perform the chaining. A parser is used to analyze the source code and recover information helpful to the service generation. Finally, a code generator uses information from the parser to synthesize a service with all the properties recovered from the original function. We also include Context-oriented primitive, helping the developer in writing complex behavior for his application to let it adapt itself to the changing deployment infrastructure.

The deployment and building module contains an infrastructure detection module allowing Dyninka to detect which deployment infrastructure we are using. We also use the tooling solution to create a bundle of our service so that it can be deployed and run on any platform.

The runtime module manages deployments received from the deployment module. It is supported by a physical infrastructure specialized in the deployment of containerized applications. One module offers deployment mechanism through scheduling and fetching images from external sources, separated from the development environment. It also offers observability entities to third parties through deployment monitoring and event systems. It also allows recovering context information about the deployment and the infrastructure to adjust the application.

C. Dataflow & Message-passing

From a high-level perspective, when a developer writes a distributed application using Dyninka, (s)he first writes a set of annotated functions and writes the composition dataflow in the main function. Running the *Gradle* build will parse the code and generate microservices. Running the deployment script with build all services and deploy them on the specified docker registry. The deployment script will also generate one deployment script per service for the detected infrastructure and will run the scripts to deploy all services as containers. Services communicate with each other using the infrastructure network and name resolver. An example of a generated service is available at https://gist.github.com/p-fortier/7990c7f2a414b8dd732aca6430378f94.

Our services rely on DNS services discovery provided by the infrastructure. Each of the services communicates with another using a domain name, defined as the original function name. To perform any routing, we introduce the routing context in the request header in the form of an ordered list of names that will be used to reach the next service in the call stack defined in previous work [5]. A user can build his own dataflow by specifying in his request the list of functions to call or to use the default route implemented in the last service called in the main function.

In classic Serverless implementations, the sequencing of service invocation is done by writing compositions as sequences using a command-line interface, a graphic interface or a custom

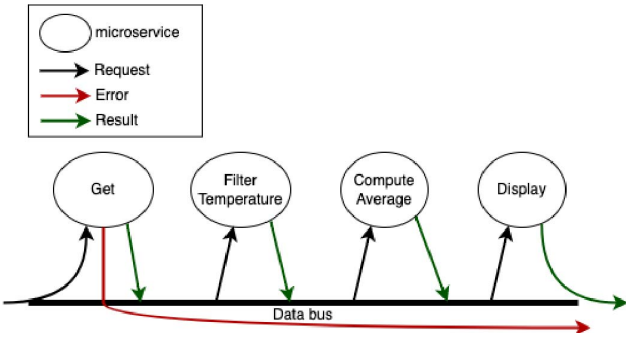


Fig. 3: Message passing in Serverless

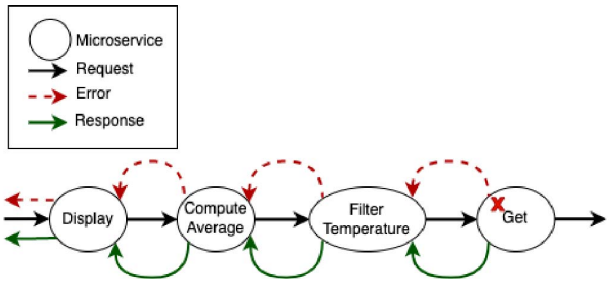


Fig. 4: Message passing in Dyninka

program. Compositions are stored in a database and are queried in order to invoke the next service or recover inputs. This method induces infrastructure overhead. Serverless functions communicate with each other using a centralized messaging solution. Figure 3 shows how microservice applications communicate in a centralized messaging system. Services append data in a message broker and fetch from it.

Our solution cuts this approach and instead makes services directly communicate with each other. Each service has knowledge of the next service of the sequence. This is done by feeding the first service with the entire sequence of service which will feed the next service with the remaining of the sequence, until the sequence is depleted. Figure 4 shows point-to-point communication for Dyninka. This implementation choice allows us to remove the overhead introduced by the messaging system in order to gain performance. By removing the message-oriented middleware supporting the framework, we also remove the properties it introduces such as persistent event storage and message transformation during message delivery.

Because we are distributing our application and abstracting the communication between services, we need also to handle fault tolerance. We are abstracting error handling in communications between services, data serialization, and deserialization. We let developers handle errors within the functions they write, while insuring their propagation to each the concerned services.

Exceptions can come from two sources, the process function within the service or the network. If the process function returns an exception, the context of the dataflow failed and the failure is transmitted downstream through the protocol used for

communication. When a service receives an exception from an upstream service, they immediately forward it downstream until the user of the application receives the exception.

In case of network fragmentation or a failed node, we rely on the timeout threshold used to detect and throw an exception downstream.

IV. Implementation and Evaluation

This section presents implementations and experimental results, assessing that our approach is similar to other FaaS platforms but also provides a quick and easy prototyping solution for dataflow distributed applications.

We first validate the general design of Dyninka with macro-benchmarks - using a simple use case described in section II. We show that our system, based on a point-to-point communication model, introduces an overhead comparable to OpenFaaS when composing functions in a dataflow. Next, we highlight the benefits of Dyninka decentralized routing compared to a centralized messaging model when increasing the throughput. The end of this section explains how Dyninka simplifies the programming of distributed dataflow applications over a serverless infrastructure.

Evaluation setup All experiments are run on a cluster composed of 2 machines running virtual machines. Each VM uses 2Go of RAM and 1 vCPU. Although it seems not much, it is the equivalent of a *t2.small* AWS EC2 instance. In the context of FaaS computing, where computing is equivalent to cost, it is plenty enough. In order to run OpenFaaS, one node has been tagged as master and was set up with 5Go of RAM.

Dataset We used the *Urban Observatory of Newcastle*[2] to provide a real-life dataset of connected building sensors. The dataset we are using contains 550k sensors weather values spanned over one-month duration and multiple metrics. We filtered the dataset to use only the temperature values, which count for 50k values.

Load testing We use Hey to perform load testing. We removed any TCP Keep-Alive feature to have the most consistent testing profile between each scenario.

OpenFaaS Configuration Because Dyninka does not perform auto-scaling under heavy load, we removed the autoscaling from each function in OpenFaaS.

A. Dyninka Implementation

Dyninka applications are written in Kotlin and use Gradle to manage dependencies and compilation. Kotlin parser is generated with *ANTLR* using Kotlin's grammar [6]. In its current state, our system runs on Kubernetes and its lightweight version K3s to deploy, run and manage generated microservices. Concurrent and asynchronous programming are performed using the Vert.x[3] toolkit.

In the current implementation, services' state is not shared across multiple instances. Stateful services are out of this paper's scope and are discussed in related works. Common practices on Serverless and FaaS include shared states outside the service [7] or handed over a middleware [8].

[2]https://newcastle.urbanobservatory.ac.uk

[3]https://vertx.io

TABLE I: Error rate comparison

Hey profile	Dyninka	OpenFaaS
1 worker	0.25%	0.0%
5 workers	0.0%	0.0%
10 workers	0.0%	94.32%
50 workers	18,81%	99.03%

B. Comparison with OpenFaaS

Setup For this comparison, we provide the same infrastructure as Dyninka: a local cluster composed of 3 machines running virtual machines.

Deployment We deployed OpenFaaS on top of Kubernetes using Helm charts.

Composition Function composition is not native to OpenFaaS but is doable on OpenFaaS using an orchestrating function. This approach uses the OpenFaaS gateway to call each member of the dataflow, gather results and transmit them to the next step.

State State is handled the same way as Dyninka. It is not shared among service instances.

Because OpenFaaS support JVM languages through templates, we were able to write the scenario in Kotlin. OpenFaaS setup uses three nodes. One node labeled *core* holds every component, such as databases and key-value stores. All three nodes also serve as invokers, running the functions on pods created for this sole purpose. In order to recover traces to measure performance, we added logs messages that would be introduced by Dyninka. We are tracing service invocation, main process execution, and service exit.

C. Error Rate

We need to understand how the system reacts to heavy load during our experiments. Table I shows the error rate for every Hey profile run. For low worker count, we do not have performance issues on neither OpenFaaS nor Dyninka. When increasing the load, OpenFaaS gateway node and service nodes crash and are constantly redeployed, thus the high error rate. Figure 7 shows the rupture of OpenFaaS when 10 workers send requests. The system never manages to recover from the failure and keeps crashing. This result states that Dyninka is more suited than OpenFaaS for limited environments used in Edge and Fog Computing.

In a scenario using 5 workers, both Dyninka in figure 6 and OpenFaaS in figure 5 are showing normal behavior.

In a scenario using 10 workers, we can observe in figures 7 and 8 the first difference in error rate between OpenFaaS and Dyninka. After 90 seconds, OpenFaaS starts to reject requests, whereas Dyninka does not.

With 50 workers, both systems reject requests, but they do not behave the same way. In figure 9 OpenFaaS reject 99,3% requests. At 230, 480 and 550 seconds into the scenario, the gateway pod in charge of handling both internal and external requests crashes and Kubernetes tries to create a new

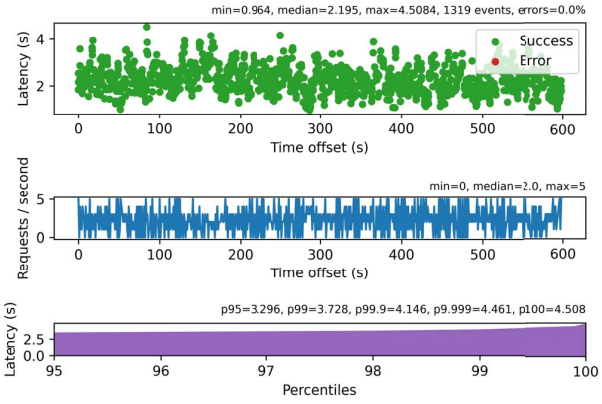


Fig. 5: OpenFaaS behavior with 5 workers

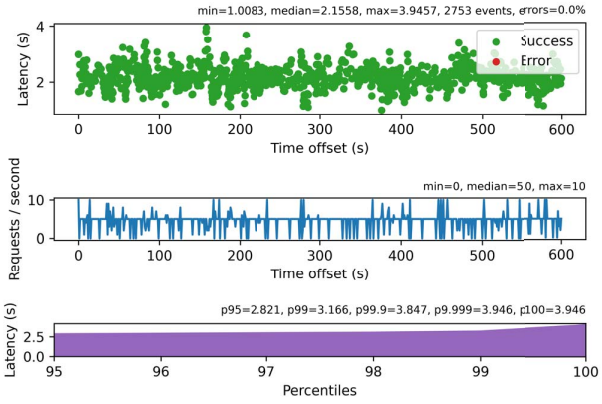


Fig. 8: Dyninka behavior with 10 workers

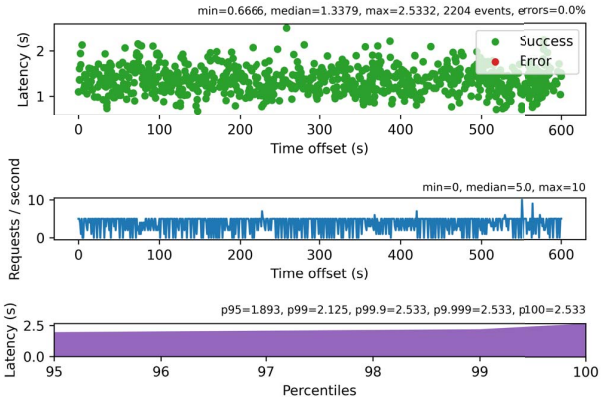


Fig. 6: Dyninka behavior with 5 workers

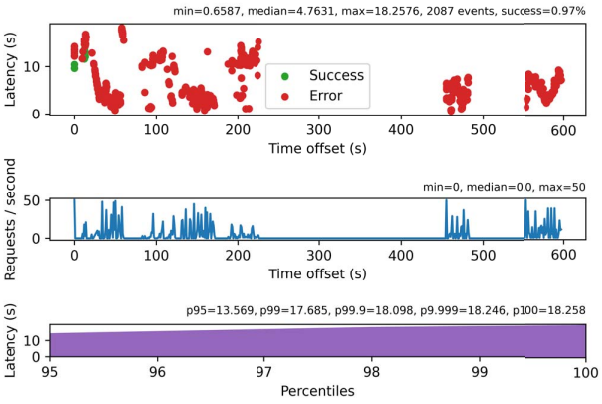


Fig. 9: OpenFaaS behavior with 50 workers

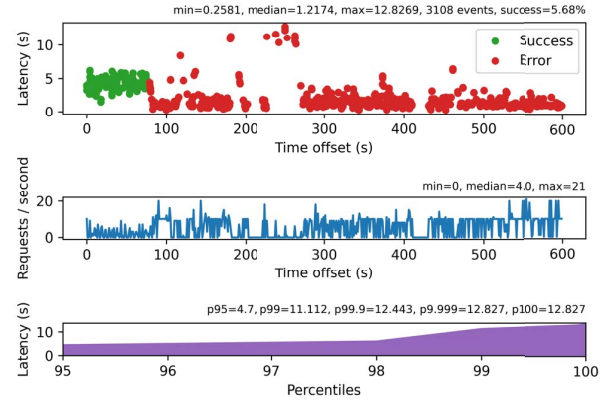


Fig. 7: OpenFaaS behavior with 10 workers

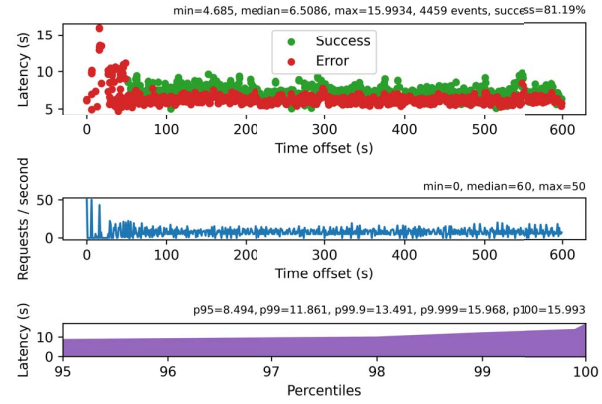
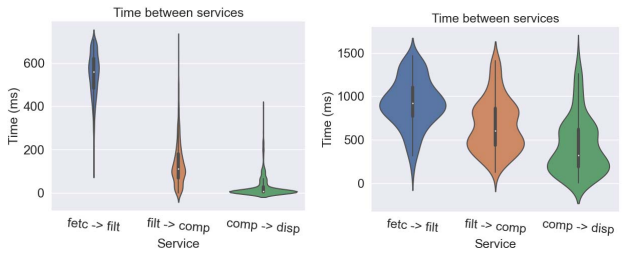
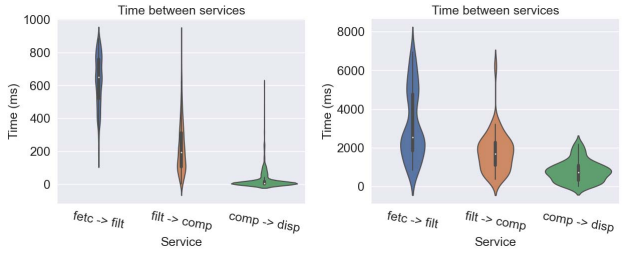


Fig. 10: Dyninka behavior with 50 workers



(a) Dyninka (b) OpenFaaS

Fig. 11: Time between services execution for 10 workers



(a) Dyninka (b) OpenFaaS

Fig. 12: Time between services execution for 50 workers

instance in parallel of incoming requests. This result shows the hardware limitation introduced by our experimental setup. For Dyninka scenario in figure 10, we can observe a complete rejection of incoming requests for the first 40 seconds and then a continuous rejection of overflowing requests counting for 18,81% of requests.

D. Latency

To evaluate the latency introduced by our system, we specified the test case by placing successive services in different nodes. This scenario maximizes the latency between services since every communication goes through the physical network. Figures 11 and 12 show the runtime overhead between each service execution. In both scenarios, Dyninka has a smaller overhead than OpenFaaS.

E. Time Spent Between Services Execution

We measured the time spent between services' execution by comparing the time when one service ends and the time when the following service starts its process function. This allows us to better understand how the overhead is distributed among all services of the application generated by Dyninka. For a low worker count, Dyninka has noticeably lower time spent between services for each service to service communication execution for the time between *fetchdatafromsensor* to *filter* which has the same median values but not the same distribution. Dyninka's distribution shows a smaller overall overhead. Figures 11 and 12 show that for a high worker count, time spent between services for Dyninka and OpenFaaS are quite different in quartile values and distributions. In Dyninka's results, time spent between services does not significantly increase with the number of workers, where it does in OpenFaaS.

TABLE II: Throughput comparison (in req/s)

Hey profile	Dyninka	OpenFaaS
1 worker	2.04	1.35
5 workers	3.67	2.20
10 workers	4.59	5.18
50 workers	7.44	3.49

TABLE III: Dyninka experiment results

Hey profile	Throughput (req/s)	Mean latency (s)	Error(%)
1 worker	2.04	0.49	0.25%
5 workers	3.67	1.36	0.0%
10 workers	4.59	2.18	0.0%
50 workers	7.44	6.75	18,81%

F. Throughput

In order to evaluate the limiting throughput (i.e., the maximum number of requests per second), we executed several scenarios with increasing worker counts. We ran the experiment using Hey as the load tester. We ran load testing for 10 minutes with several profiles: 1 worker, 5 workers, 10 workers and 50 workers. Tables III and IV highlight the resulting throughput, the mean latency, and the error rate. Table II compares Dyninka and OpenFaaS mean throughput for several workers number. In each scenario, Dyninka outperforms OpenFaaS. We also need to keep in mind that for high worker numbers, the performance of OpenFaaS is impacted by the high error rate. Figure 6 shows the behavior of Dyninka when under increasing load. While the median request rate increase from 2 to five, latency does not significantly increase.

V. Related Works

The relationship between programming language and serverless architectures is not a common topic in academic studies. The industrial serverless frameworks implement more and more runtimes in order to be more inclusive with usable programming languages and provide tool to create native applications. We discuss research that influenced our work in terms of techniques. We consider related work on multi-tier languages, Functional reactive programming languages, and languages that combine both.

FaaS Academic works bring examples of FaaS framework implementation dedicated to Edge computing. Michel et Al. Pfandzelter and Bermbach [9] propose a lightweight implementation of a FaaS framework adapted to Edge computing, although their approach's current limitation does not allow multiple nodes as FaaS workers to execute functions. Cheng et al. [10] go further with a more specialized implementation for

TABLE IV: OpenFaaS experiment results

Hey profile	Throughput (req/s)	Mean latency (s)	Error(%)
1 worker	1.35	0.74	0.0%
5 workers	2.20	2.24	0.0%
10 workers	5.18	1.71	94.32%
50 workers	3.49	6.21	99.03%

Edge-fog computing architectures handling multiple workers in the IoT context. Spillner presented through Snafu [11] an approach for quick prototyping of FaaS-based application, consuming python functions from a codebase to generate callable functions.

Serverless When Jonas et al. [12] introduced serverless architectures, they specified how cloud platforms need to have data dependencies knowledge to avoid lousy function placement in the cluster and minimize communication to increase performance. There is academic work on improving mutable share state in stateful applications, the whitepaper from Fox et al. [7] brings the consensus to keep the state of a distributed application out of the microservices allowing them to be stateless. The consensus is adopted in most academic work around stateful FaaS applications. Barcelona-Pons et al. [8] introduce a user-defined shared object stored in a data store and bring a concurrency model for serverless functions when accessing shared objects.

Multi-tier programming The history of multi-tier programming comes from web development. Serrano et al. [13] proposed a language to write a complete web application with client and server code in the same codebase. Philips et al. [14] introduced a tier splitting tool to write client and server-side Javascript as a single codebase. The most influential work for Dyninka focuses on the valuable abstractions for writing distributed applications. While [15] gives good pointers on a modular approach to multi-tier programming, Weisenburger et al. [16] [17] introduce language constructs to perform multi-tier programming with placable data and abstracted remote access to data with a more generic usage than just web related Javascript declination. The macro programming used by ScalaLoci is a direct influence on Dyninka's macro programming style. Sokolowski et al. [18] applied Scalaloci to the HPC context, showing a different use of multi-tier programming than Web applications.

VI. Conclusion

This paper presented Dyninka, a FaaS framework to quickly develop, test, and run distributed dataflow applications on a container orchestrator. Dyninka is built using a point-to-point communication system where the data dependency is expressed in the language using function composition computed at compilation, which can be dynamically overwritten. This approach differs from other implementations of FaaS - relying on a gateway or a messaging middleware to convey requests and data through each microservices. We showed that Dyninka performs better in terms of throughput, latency, and stability than OpenFaaS in a fog computing scenario, while keeping a smaller overhead. We believe that Dyninka, through its multi-tier programming touch and compose facilities, can accelerate the prototyping and the automatic deployments of FaaS-based applications. Dyninka is the groundwork for research work focusing on the expression and the inclusion of execution context in a programming language.

References

[1] I. C. education, "Faas (function-as-a-service)," 2019. [Online]. Available: https://www.ibm.com/cloud/learn/faas

[2] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend Cloud to Edge with KubeEdge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct. 2018, pp. 373–377.

[3] J. Duncan and J. Osborne, *OpenShift in Action*. Manning Publications Co, 2018.

[4] Google, "Google anthos." [Online]. Available: https://cloud.google.com/anthos

[5] P. Fortier, F. Le Mouël, and J. Ponge, "Dyninka: a faas framework for distributed dataflow applications," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, 2021, pp. 2–13.

[6] Kotlin, "Kotlin grammar." [Online]. Available: https://kotlinlang.org/docs/reference/grammar.html

[7] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research," *arXiv:1708.08028 [cs]*, 2017.

[8] 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*. Davis CA USA: ACM, Dec. 2019, pp. 41–54.

[9] T. Pfandzelter and D. Bermbach, "tinyFaaS: A Lightweight FaaS Platform for Edge Environments," in *2020 IEEE International Conference on Fog Computing (ICFC)*. Sydney, Australia: IEEE, Apr. 2020, pp. 17–24.

[10] B. Cheng, J. Fürst, G. Solmaz, and T. Sanada, "Fog Function: Serverless Fog Computing for Data Intensive IoT Services," *arXiv:1907.08278 [cs]*, Jul. 2019.

[11] J. Spillner, "Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation," *arXiv:1703.07562 [cs]*, 2017.

[12] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," *arXiv:1902.03383 [cs]*, Feb. 2019.

[13] M. Serrano, E. Gallesio, and F. Loitsch, "Hop: A language for programming the web 2.0," in *Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA '06*. Portland, Oregon, USA: ACM Press, 2006, p. 975.

[14] L. Philips, W. De Meuter, and C. De Roover, "Poster: Tierless Programming in JavaScript," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 831–832.

[15] G. Radanne, J. Vouillon, and V. Balat, "Eliom: A Language for Modular Tierless Web Programming," *arXiv:1901.11411 [cs]*, Jan. 2019.

[16] P. Weisenburger, M. Köhler, and G. Salvaneschi, "Distributed system development with ScalaLoci," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–30, Oct. 2018.

[17] P. Weisenburger and G. Salvaneschi, "Implementing a Language for Distributed Systems: Choices and Experiences with Type Level and Macro Programming in Scala," *Programming*, vol. 4, no. 3, p. 17, Feb. 2020.

[18] D. Sokolowski, P. Martens, and G. Salvaneschi, "Multitier Reactive Programming in High Performance Computing," p. 8, 2019.