Dyninka: a FaaS framework for distributed dataflow applications

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

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

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

Abstract

The Internet of Things (IoT) requires applications to deal with a large amount of data - streamed, processed and stored from small devices to analytical systems. Cloud computing offers a hardware solution to this issue, providing on-demand resources to process IoT data. The newer programming paradigms simplify the use of those cloud resources. The Function-as-a-Service (FaaS) and the Serverless paradigm transform the conception of microservices applications to the definition and the composition of several callable functions. Although defined as distributed architectures - mostly publicly available solutions rely on either a gateway or an internal messaging middleware. These architectures create a *single point of failure* in exchange for more straightforward service to service communication.

In this article, we present Dyninka, a framework to rapidly prototype FaaS-based distributed dataflow applications. Its programming model gathers the definition and the composition of services within a single file using the *multitier programming* paradigm and compiles them into a multitude of services deployable on cloud computing infrastructure. Dyninka is built without a gateway or a messaging platform, and services communicate directly with each other or with the cloud abstracted infrastructure. As a result, we reduce the network and the computation overheads introduced by commercial FaaS frameworks such as OpenFaaS.

We validate Dyninka on a Fog computing scenario with limited resources and several load profiles. For all scenarios, Dyninka shows better stability, throughput and a reduced overhead compared to OpenFaaS.

CCS Concepts: • **Software and its engineering** → *Application specific development environments.*

Keywords: FaaS, Multitier Programming, Macro Programming, Distributed System, Dataflow

ACM Reference Format:
Patrik Fortier, Frédéric Le Mouël, and Julien Ponge. 2021. Dyninka: a FaaS framework for distributed dataflow applications. In *Proceeding of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS'21), October 18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3486605.3486789

1 INTRODUCTION

The *microservice architecture* (MSA) has become more popular over the last few years with the rise of cloud services like Amazon Web Service and Microsoft Azure. This architectural style favors the development of multiple services where each 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 or mobile devices provide. The flow of information from IoT devices can be modelled by the Dataflow model. Data is processed, stored, and exposed by distributed applications, 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. 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.

Multitier programming provides abstractions to create distributed systems. Within the same file, the developer includes all components and how they communicate with each other. The compiler generates services as deployment units. Using a unified codebase and a unique language allows static type-checking through the entire data flow. Implementations can completely abstract communication code or keep simplified communication constructs to handle network failures and separate local and remote values [16].

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

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 multitier 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 fundamental properties of functional programming, and multitier programming removes the developer's burden to handle service communication and non-business-oriented logic, keeping the application complexity low.

Our evaluation shows that Dyninka introduces a very 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.

In summary, the work presented in this paper - Dyninka, a FaaS framework for distributed dataflow applications - makes the following contributions:

- We propose a build module that deploys generated dataflow in an infrastructure agnostic manner.
- We propose an alternative to traditional serverless message passing.
- We implement a prototype of Dyninka based on Kotlin, targeting Kubernetes as deployment infrastructure.
- We conduct experiments on real-world dataset to compare the performance of Dyninka with serverless solution OpenFaaS.

The remaining of the paper is structured as follows: Section 2 presents a usecase application on which the paper relies to introduce Dyninka's mechanisms and features. Section 3 puts this work in perspective to the challenges encountered with some background. Section 4 describes the system architecture and its design. The evaluation is presented in section 5, Dyninka is validated and compared to other FaaS frameworks using a macro-benchmark through a smart building use case showcasing stream processing capabilities. We review related work in section 6. Section 7 opens on future work and concludes this work.

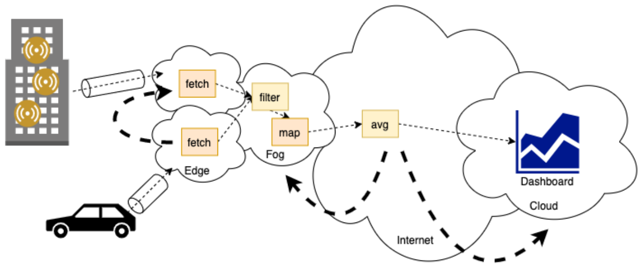


Figure 1. Use case: processing smart building data

2 USECASE APPLICATION

Figure 1 and listing 1 present the usecase 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 builds a moving average of the building's temperature. All metrics are collected, but we filter all messages which do not provide temperature information. We map the resulting messages to extract temperature and compute the average temperature using a moving average. In order to receive information from the sensors endpoint, we used a front HTTP client on the *get()* service. We use an accumulator placed on the *movingAvg* service gathering incoming temperature data and computing the moving average. The new value is stored in the accumulator. Data is communicated between services uses JSON as a serializable data format.

3 BACKGROUND

3.1 FaaS Computing

From the IBM definition, *Function-as-a-Service* is a cloud computing technique allowing you to execute code in response to events without the complex infrastructure typically associated with building and launching microservice applications [4]. With the rise of *edge computing* and *fog computing*, infrastructure is being shifted towards the edge of the network for performance purpose. A cluster can then be composed of several edge clouds with computing power and storage which can be 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*[18] have been developed.
They provide core infrastructure support for networking,

```
fun get(): SensorValue {
  ...
}

fun SensorValue.filterTemperature(): Temperature {
  ...
}

fun Temperature.map(): TemperatureValue {
  ...
}

fun TemperatureValue.movingAvg(){
  ...
}

fun main(){
  get()
  .filterTemperature()
  .map()
  .movingAvg()
}
```

Listing 1. Dyninka example

application deployment and metadata synchronization between cloud and edge. Other tools like Google Anthos [6] allow the unification of multiple cloud infrastructures into one single cluster managed by a higher level control plane. This approach is yet very limited since it requires the use of *Google Kubernetes Engine*. Other orchestration solutions like Apache Mesos are found left out although deployments actions between the two orchestrators are quite similar: create, modify, delete and observe deployments units.

3.2 FaaS Programming

On one hand, developers are not expected to manage the environment on which they run their software. On the other hand, IT operational skills for managing the deployment infrastructure is mainly separated from. The DevOps sets of practice originated from the Agile software development model is an organizational answer to this problem where deployment becomes more taken into account in the software delivery process. The FaaS and Serverless paradigms come as a solution for the developers, allowing them to deploy an application without required knowledge of the deployment infrastructure. The continuous deployment approach in a microservices context tries to deploy the newest available service version without disturbing traffic in progress. This practice is very well integrated in FaaS and Serverless architecture where a service is just a small unit easily updatable. This technique requires additional tool chains in the from of Continuous Integration workflows and their integration comes with challenges [7].

4 DYNINKA DESIGN

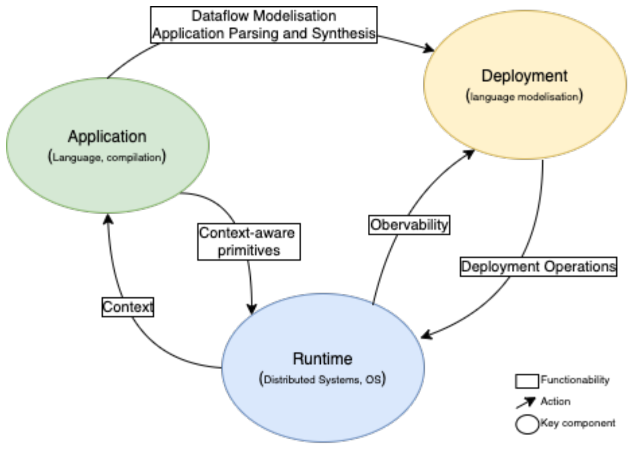


Figure 2. Dyninka Contributions

4.1 An 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. The complete circular adaptation process is presented in Figure 2. Dyninka handles building, compiling and deployment for distributed dataflow applications. In this paper, we focus on the dataflow model and the deployment operations. Context-oriented primitives, observability and context expression is out of the scope of this paper. Figure 3 shows the deployment workflow. Dyninka handles every aspect of software building process. It parses the source codebase, generates the resulting code and wrap each microservices 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 the scope of the paper). Dyninka's approach promotes a point-to-point communication system instead of a centralized messaging system, more common in FaaS Serverless approaches. Services communicates directly with the next one instead of handling messages to or requesting them from a middleware.

4.2 Programming model

The Dyninka programming model is function-based and can be integrated with any other functional programming language. As Kotlin is the programming language supported in our implementation, the following description considers its jargon.

Overall, a Dyninka application strongly resembles to a tierless single-threaded functional program, besides some additional annotations. Table 1 summarizes the key programming abstractions used in Dyninka and are detailed later

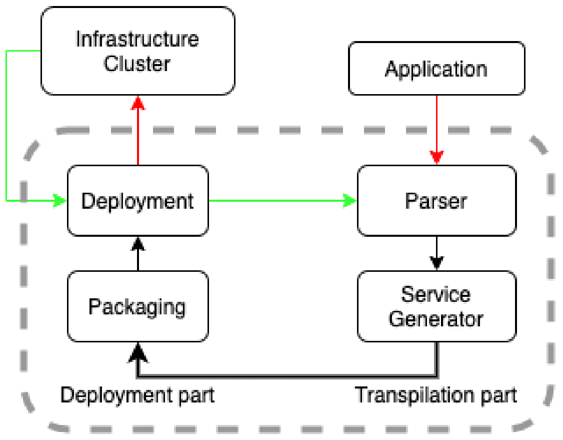


Figure 3. The framework of Dyninka

Abstraction	Description
@Service	Services are defined like function annotated. Dyninka generates a service with the same capabilities than the function. The annotation @Service is used to identify functions from which we generate services.
@Placed(val location: String)	User-defined local variable. The annotation *@Placed(val location:String)* is used to place variables. This technique is used for introducing local memory to normally stateless functions and services.
Abstracted synchronization	Generated services provides asynchronous constructs (Promise, Future) to perform dead-lock free communication.
Client side type safety	*T :->Future<T>* Type safety performed on client side is respected when converting functions into asynchronous services.

Table 1. Dyninka programming abstractions

Dataflow Our dataflow is modeled using a functional programming approach. Each function forms a process.

Service boilerplate To write a distributed dataflow application as a microservice composition, developers first build its logic.

Data placement When the developer needs to introduce some memory in his logic, e.g when implementing an accumulator, he can introduce such element using the @Place annotation.

Service synchronization Function arguments translate to service dependencies. Developers may want to wait for a service result before invoking another one. Dyninka relies on asynchronous communication and uses Future to avoid deadlocks. Future allows immutable inputs which may not be available at all. Each service needs to wait for its Future input to be completed in order to process its value.

Asynchronous types Since generated services are using Future as synchronization mechanism, we abstract asynchronous types. We wrap the type used by allowing the programmer to use any types and convert them if needed to asynchronous structures.

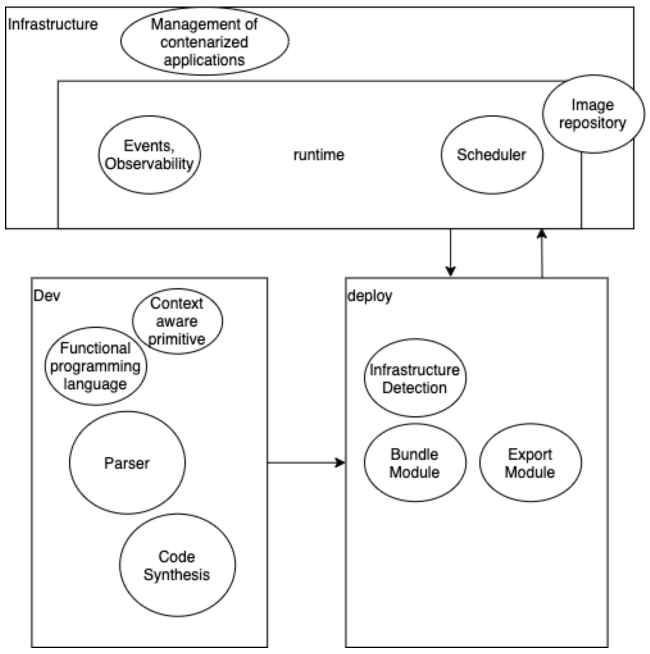


Figure 4. Architecture of Dyninka

4.3 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 to generate the associated code in order to create a distributed implementation of the application.

(2) A deployment module to bundle and deploy in a infrastructure agnostic manner each service of the application. Its role is to identify on which infrastructure the service is going to be deployed and send the proper commands to the container orchestration system.

(3) A container orchestration runtime to manage deployed application, to retrieve contextual data and to 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 4 gives a high-level overview of the system's architecture.

The bottom left-hand part gives details on the dataflow application generation. 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 in order to analyse the source code and recover information helpful to the service generation. And a code generator using 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 writing complex behavior for his application to adapt itself to the changing deployment infrastructure.

The bottom right part the deployment and bundling module. It contains an infrastructure detection module allowing Dyninka to detect which deployment infrastructure we are using. We also use tooling solution in order to create a bundle of our service deployable and runnable on any platform.

The top part is the runtime, managing actual deployment 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 fetch images from external sources, separated from the development environnement. It also offers observability entities to third parties through deployment monitoring and event systems. This allows to recover context information about the deployment and the infrastructure in order to adjust the application.

4.4 Language

The language used by Dyninka is based on Kotlin. We are using Kotlin as our first source language supported because it is a modern general-purpose programming language. It uses many abstractions and primitives commonly used in stream processing languages and support dataflow as a programming model with several implementations. Kotlin is also a great language to be extended as they provide tools to create domain-specific language from their generic-purpose language (GPL).

We use macros as a metaprogramming technique to detect services to be generated and to add specifications to them. We defined two macros in the Listing 2. *@Service* identifies a function which will become a service. *@Placed* is a macro used to attach a variable to a specific service. It takes as argument the name of the service on which the variable should be placed. After compilation, the variable will be available in the service's companion object. This technique is used in Dyninka to keep values between different service invocations. Macros are readable by our parser and allow us to adapt our service to the need of the developer based on a defined set of properties. Global variables and functions which are not annotated will not be processed by the parser.

```
@Placed("get") val vertx: Vertx = Vertx.vertx()

@Placed("movingAvg") var counter: Int = 0

@Placed("movingAvg") var movingAvr: Float = 0F

@Service fun get(): Future<JsonObject>
{
  return WebClient.create(vertx)
    .get("sensor","/").expect(SC_OK)
    .send().map {
      it.bodyAsJsonObject()
   }
}

@Service fun Future<JsonObject>.filter(): Future<JsonObject>
{
  val filtered = Promise.promise<JsonObject>()
  this.onSuccess {
    if (it.getString("Variable") == "Temperature") {
      filtered.complete(it)
    } else {
      filtered.fail(Throwable("not a temperature sample"))
    }
  }
  return filtered.future()
}

@Service fun Future<JsonObject>.map(): Future<JsonObject>
{
  return this.map { t: JsonObject ->
    JsonObject()
    .put("Value", t.getFloat("Value"))
  }
}

@Service fun Future<JsonObject>.movingAvg(): Future<JsonObject>
{
  val promise = Promise.promise<JsonObject>()
  this.onSuccess {
    val next = it.getFloat("Value")
    movingAvr = (next + counter * movingAvr) / (counter + 1)
    counter++
    val result = JsonObject().put("Average", movingAvr)
    promise.complete(result)
  }
  return promise.future()
}

fun main()
{
  get().filter().map().movingAvg()
}
```

Listing 2. Macro listing

4.5 Language Parser

The language parser main role is to process the written application and to populate a data structure with all primordial information from source code to build distributed microservices. Core components we are looking for are Services, Imports, Data structures and the flow that follows the data.

We generated a Kotlin parser using ANTLR and the Kotlin grammar provided by Kotlin itself. We use ANTLR generated parser to build a walkable abstract syntax tree. ANTLR also generates tools to run through the tree. We use the Visitor pattern to browse our AST and look for all the core components defined earlier.

The Intermediate Representation gathers all core components found by the visitors. The code generator uses it as input to generate our microservices. It is composed by the list of *Services*, the *dataflow*, the *dependencies* and the *data structures* used in the application.

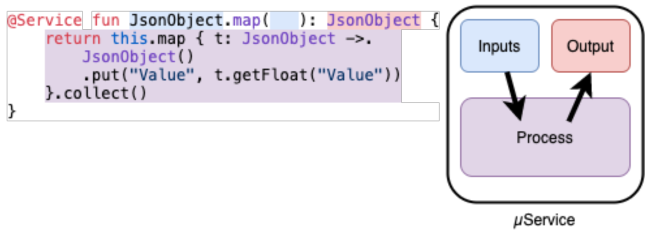
Services are describe by the name of the function, the implicit and explicit inputs' names and types, the explicit output type, and the statements used in the function. It allows us to generate the microservice with the proper input and output. When a function is written as asynchronous using Future and Promises. We adapt the service to serve asynchronous results to the next service.

The dataflow is a list of function calls representing the path taken by the data through the services. The call stack is defined as the reversed dataflow, representing the order in which services are called to process one request to the application. To generate our distributed application we are using the call stack and the developer writes the dataflow in his main function.

Dependencies are all imports the developer is adding to his application. The strategy used is to add all import made by the developer and let the compiler deal with the unused one.

The data structures are all classes introduced by the developer into his application, perhaps to be used as input or output for his functions. A class is defined by its name and its attributes. Kotlin uses special type of classes called data classes, these are identifiable with their keyword *data*.

4.6 Service generator



```
@Service fun JsonObject.map(    ): JsonObject {
    return this.map { t: JsonObject ->.
        JsonObject()
        .put("Value", t.getFloat("Value"))
    }.collect()
}
```

Figure 5. Dyninka parser generating a microservice.

The service generator component is in charge of writing the microservice code using the intermediate representation populated in section 4.5.

The Service generated is divided in 3 parts: The process function, the inputs and the output. The figure 5 shows how each part of a function is transcribed into a microservice.

The process function contains the statements gathered in the service data structure. Some keyword modifications need to be applied in order to fit the new execution context. Because we aren't using the Method chaining syntax anymore. The keywords *this*, *self* and *me* working as an immutable reference to the current object represent in this case the implicit input of the so we rewrite this keywords to the keyword *implicitInput.*

The web server boilerplate brings all components to have a fully working web server serving the service. It returns the result of the process function or any exception that would be thrown. Boilerplate counts as the major part of the final result. Although it doesn't add any meaningful feature to the service, its design impacts on performance. We opted for a asynchronous behavior in order to remove any bottleneck.

For each inputs described in the Service data structure, we introduce a web client request. Such request is asking another service for the information needed for the process function to be invoked. Inputs can be implicit when defined before in the function's signature like in Kotlin (ex: ***Int****.add(...)*) or explicit when used as an argument of the function. Implicit inputs is omnipresent in linear function composition *a().b().c()* where the result of *a()* is an implicit input of *b().*

We also introduce the asynchronous result of the client request as a future. We create a composition of them where we invoke the function process when all future are completed. If one future is failed, we are propagating the failure towards the service caller.

The output is the response of the web server boilerplate. 3 different answers can be propagated to the calling service: First the result of the process function, when no issue has been encountered. We then write the result as a JsonObject with a field *implicit* set to the serialized returning value of the process function.

If any of the input asynchronous value is failed, we are directly writing the failure throwable as response to the caller's request. Last case, when the process function throws an exception, we simply forward this exception in the response.

4.7 Microservice packaging and deployment

In order to be deployed on the infrastructure, services need first to be packaged into container images. According to docker documentation [3], a container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. In order to create such images, we move generated microservices into standalone projects that can be built as container images using the google JiB plugin. We use a generic project based on a software project management and building tool in which we replace the main application with our generated service and perform some operations of rewriting to get a new project for one specific service.

Building the project leads to creating a container image and publish it to the target repository. Which can be a public repository like Dockerhub or a private registry.

Deployment is done using a descriptive document depending on the infrastructure. For Kubernetes, we use a YAML file and Mesos coupled with the framework Marathon uses a JSON file to specify deployment.

```kotlin
class MapService {
  companion object {
    val vertx: Vertx = Vertx.vertx()

    fun process(implicitInput: Future<JsonObject>):
      Future<JsonObject> = implicitInput.map { t:
              JsonObject ->
                JsonObject().put("Value",t.getFloat("Value"))
              }
            }
}

fun main() {
  ...
  router.route().handler { routingContext ->
    ...
    val implicitInput = Promise.promise<JsonObject>()
    val list = mutableListOf(implicitInput.future())
    client.get(next, "")
            .expect(SC_OK)
            .send { res ->
              if (res.succeeded()) {
                implicitInput.complete(res
                                        .result()
                                        .bodyAsJsonObject()
                                      )
              } else if (res.failed()) {
                implicitInput.fail(res.cause())
              }
            }
    @Suppress("UNCHECKED_CAST")
    val future = CompositeFuture.all(list as List<Future<Any>>?)
    future.onComplete { res ->
      if (res.succeeded()) {
        MapService.process(implicitInput.future()).onComplete {
          res ->
          if (res.succeeded()) {
            routingContext.response().end(res.result()
                          .encodePrettily())
          } else if (res.failed()) {
            routingContext.fail(500, res.cause())
          }
        }
      } else {
        routingContext.fail(500, res.cause())
      }
    }
  server.requestHandler(router).listen(port)
}
```

Listing 3. The generated Map service

4.8 Application runtime

A container orchestration infrastructure manages the lifecycle of containers. It is used to automate many tasks like the provisioning and deployment of containers. It offers some services like the allocation of computer resources between container (network, volumes, exposure outside the orchestrator) and guaranties properties like the availability of a container by redeploying containers or scaling them up and down.

Our services rely on the DNS service offered by the infrastructure. Each service communicates with another using domain name which is 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 which will be used to reach the next service in the call stack defined in section 4.5. An user can build his own dataflow by specifying in his request the list of functions to call or use

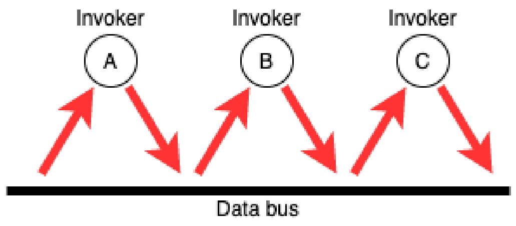


Figure 6. Message passing in Serverless

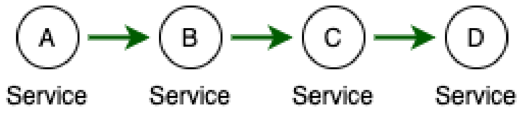


Figure 7. Message passing in Dyninka

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 or a graphic interface. Compositions are stored in a database and are queried in order to invoke the next service or recover inputs. This method induce infrastructure overhead. Serverless functions communicate with each other using a centralized messaging solution. Figure 6 shows how microservice applications communicate using a centralized messaging system. Services append data in a message broker and fetch from it.

Our solution cuts this approach and instead makes the service 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 7 shows point to point communication for Dyninka. When service A invoke service B, it provides the path towards services C and D. Service B will then continue the process and serve service C means to invoke service D.

4.9 Execution lifecycle

From a high level perspective, when a developer writes a distributed application using Dyninka, he first writes his set of annotated functions and writes his dataflow in the main function. Running the gradle build will parse his code and generate microservices. Running the deployment script with build each 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. Infrastructure pulls the required docker images from the registry and start service instances according to the specifications in the deployment script. Services communicates with each other using the infrastructure name resolver and network. Infrastructure like Kubernetes offers autoscaling to scale numbers of instances based on CPU load.

After updating one service, the developer can run the gradle build again to generate service. All services will be rewritten. He can specify the service he wants to redeploy. The image on the registry will be updated and the infrastructure can update the service by requesting a new instance which will use the updated image. On Kubernetes, one can create a rolling update to seamlessly change the version of the service used.

4.10 Fault tolerance

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 but we let the developers handle error within the functions they write while we insure its propagation to each 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 is 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 is receives the exception.

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

5 IMPLEMENTATION & EVALUATIONS

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 macrobenchmarks, using a simple use case, we show that our system, based on a point to point communication model introduces an overhead comparable to OpenFaaS overhead when composing functions in a dataflow. Next, we highlights 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 local cluster composed of 2 machines running virtual machines. Each VM uses 2Go of RAM and 1 vCPU. In order to run OpenFaaS, one node has been tagged as master and was setup with 5Go of RAM.

Dataset We use the Urban Observatory of Newcastle[2] to provide real life dataset of connected building sensors. The dataset we are using contains 550k sensors values spanned over one month duration of which we focus on 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 scenarios.

OpenFaaS Configuration Because Dyninka doesn't perform auto-scaling when under heavy load, we removed the autoscaling from each function in OpenFaaS.

5.1 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[9]. KotlinPoet is used as our service generator. Vert.x 4 is handling request as our default web server and client with HTTP as the protocol used.

Generated Services are deployed on a docker repository using the *Google Jib* Gradle plugin [3].

Our system in its current state runs on Kubernetes and its lightweight version K3s to deploy run and manage generated microservices. The type system allows to write applications transporting JSON objects, asynchronous structures like Future and Promises and primitive types. JSON serialization is performed by the Vert.x implementation.

In current implementation, services' state is not shared across multiple instances. This is a limitation of our solution. Common practices on Serverless and FaaS puts shared state outside of the service [5].

5.2 Usecase application implementation

We have developed our sample application using Dyninka and using OpenFaaS. We handcrafted a service to serve our dataset. This sensor service runs a Vert.x web server and serve a line of the dataset csv file for every request. The chaining of action in OpenFaaS is made possible using an orchestrating function. The function is written in Kotlin in a similar way to the services and make asynchronous HTTP requests to each service through the OpenFaaS gateway and forward the results to the next request. Listing 4 shows how the orchestrating function in OpenFaaS operates.

5.3 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.

[2]https://newcastle.urbanobservatory.ac.uk
[3]https://github.com/GoogleContainerTools/jib

```kotlin
val webclient = WebClient.create(routingContext.vertx())
webclient
  .post(port, gateway, "/fetchdatafromsensor")
  .sendJson(value)
  .onFailure(failureHandler)
  .onSuccess { node1 ->
    webclient
    .post(port, gateway, "/filtertemperature")
    .sendJson(node1.bodyAsJsonObject())
    .onFailure(failureHandler)
    .onSuccess { node2 ->
        webclient.post(port, gateway, "/computeaverage")
        .sendJson(node2.bodyAsJsonObject())
        .onFailure(failureHandler)
        .onSuccess { node3 ->
            webclient.post(port, gateway, "/display")
            .sendJson(node3.bodyAsJsonObject())
            .onFailure(failureHandler)
            .onSuccess { node4 ->
                routingContext.response()
                .end(node4.bodyAsJsonObject()
                            .encodePrettily())
                }
            }
        }
    }
}
```

Listing 4. Sample code of the orchestrating service

Composition Function composition isn't 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 components such as databases and key-value stores. All three nodes also serve as invoker, 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.

5.4 Error rate

During our experiments, we need to understand how the system reacts to heavy load. Table 2 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 8 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.

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%

Table 2. Error rate comparison

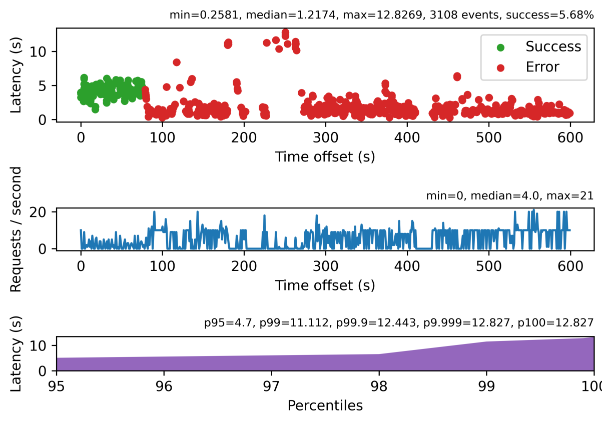
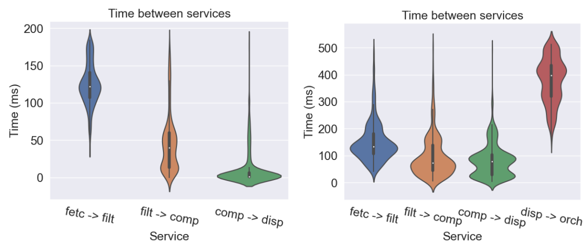


Figure 8. OpenFaaS behavior with 10 workers



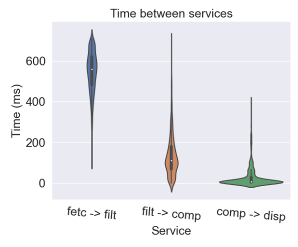
(a) Dyninka **(b)** OpenFaaS

Figure 9. Time between services execution for 1 worker

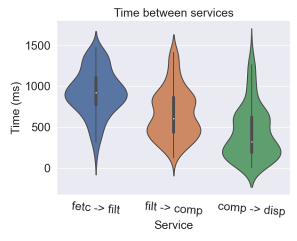
5.5 Latency

To evaluate the throughput of our solution, we specified the test case by placing successive nodes in different nodes. This scenario maximizes the latency between services since every communication is going on the physical network. We applied a linearly increasing load of clients in order to identify a breakpoint in the latency. Figures 9 and 10 show the runtime overhead between each service execution. In both scenario, Dyninka has a smaller overhead than OpenFaaS.

To evaluate the overhead introduced by the service generation, we compare the time spent inside the service, from request reception to response emission with the time spent

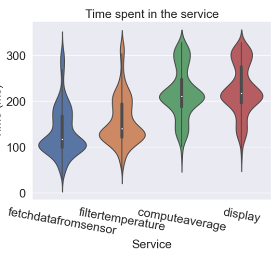


(a) Dyninka

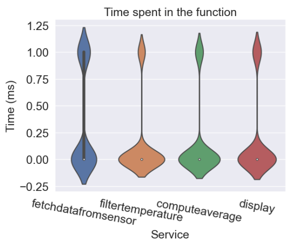


(b) OpenFaaS

Figure 10. Time between services execution for 10 workers



(a) Full service



(b) Function

Figure 11. Overhead for Dyninka (1worker)

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 3. Throughput comparison (in req/s)

in the process method. We can see in figure 11 the time spent in both the process method and the whole function. Most of the time spent is done inside the overhead part of Dyninka.

5.6 Throughput

In order to evaluate the limiting throughput, the maximum number of request per second, we executed several scenarios with increasing number of workers. We ran the experiment using Hey as load tester. We ran load testing for 10 minutes with several profiles: 1 worker, 5 workers, 10 workers and 50 workers. Tables 4 and 5 highlight the resulting throughput, the mean latency and the error rate.

On throughput, table 3 compares Dyninka and OpenFaaS mean throughput for several workers number. On each scenarios, Dyninka outperforms OpenFaaS. We also need to keep in mind that for high worker number, the performance of OpenFaaS is impacted by the very high error rate.

Figures 12 and 13 show the behavior of Dyninka when under increasing load. While the median request rate increase from 2 to five, latency does not significantly increase.

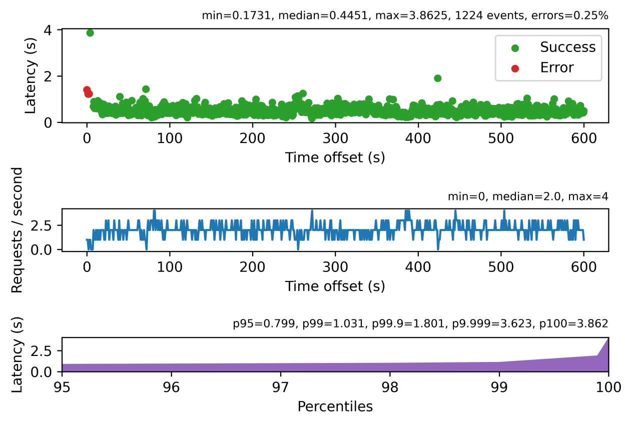


Figure 12. Dyninka response to load testing with 1 concurrent user

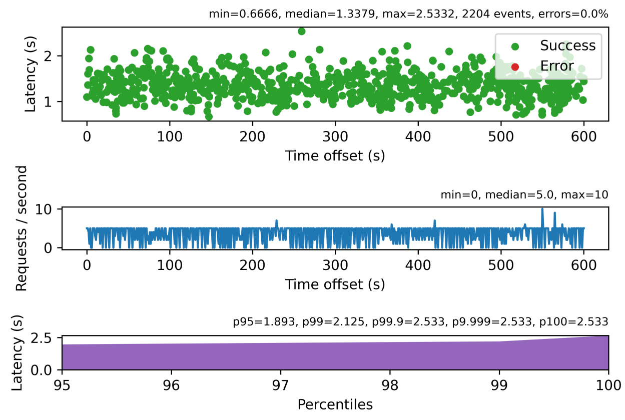


Figure 13. Dyninka response to load testing with 5 concurrent user

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%

Table 4. Dyninka experiment results

6 RELATED WORK

The relationship between programming language and serverless architectures is not a very common topic in academic studies. Industrial serverless framework implements more and more runtimes in order to be more inclusive with usable

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%

Table 5. OpenFaaS experiment results

programming languages and provides tool to create native applications.

We discuss research that influenced our work in terms of techniques. We consider related work on multitier languages, Functional reactive programming languages and languages that combine both.

FaaS Academic work bring examples of FaaS framework implementation dedicated for Edge computing. Michel et Al. Pfandzelter and Bermbach [10] propose an lightweight implementation of a FaaS framework adapted to Edge computing, although the current limitation of their approach doesn't allow multiple nodes as FaaS worker to execute functions. Cheng et al. [2] go further with a more specialized implementation for Edge-fog computing architectures handling mutliple worker in the IoT context. Spillner presented through Snafu [15] an approach for quick prototyping of FaaS based application, consuming python functions from a codebase to generate callable functions.

Serverless When Jonas et al. [8] introduced serverless architectures they specified how cloud platforms need to have data depedencies knowledge in order to avoid bad function placement in the cluster and minimize communication to increase performance. There is academic work on the improvement of mutable share state in stateful applications, the whitepaper from Fox et al. [5] 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. [1] introduce of user-defined shared object stored into a data store and bring a concurrency model for serverless functions when accessing shared object.

Multitier programming The history of multitier 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. [11] introduced a tier splitting tool to write client and server side Javascript as a single codebase. The most influencal work for Dyninka focuses on the useful abstractions for writing distributed application. While [12] give good pointers on a modular approach of Multitier programming,Weisenburger et al. [16][17] introduce language constructs to perform multitier 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. [14] applied Scalaloci to the HPC context, showing a different use of multitier programming than Web applications.

7 CONCLUSION

This paper presents 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 on compilation and that can be 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 show that Dyninka performs better in terms of throughput, latency and stability than OpenFaaS in a fog computing scenario and has a smaller overhead introduced. We believe that Dyninka, through its multitier programming touch and compose facilities, can quicken 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] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference*. ACM, Davis CA USA, 41–54. https://doi.org/10.1145/3361525.3361535

[2] Bin Cheng, Jonathan Fürst, Gurkan Solmaz, and Takuya Sanada. 2019. Fog Function: Serverless Fog Computing for Data Intensive IoT Services. *arXiv:1907.08278 [cs]* (July 2019). arXiv:1907.08278 [cs]

[3] Docker. [n.d.]. What is a Container? https://www.docker.com/resources/what-container

[4] IBM Cloud education. 2019. FaaS (Function-as-a-Service). https://www.ibm.com/cloud/learn/faas

[5] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2017. Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research. *arXiv:1708.08028 [cs]* (2017). https://doi.org/10.13140/RG.2.2.15007.87206 arXiv:1708.08028 [cs]

[6] Google. [n.d.]. Google Anthos. https://cloud.google.com/anthos

[7] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 197–207. https://doi.org/10.1145/3106237.3106270

[8] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv:1902.03383 [cs]* (Feb. 2019). arXiv:1902.03383 [cs]

[9] Kotlin. [n.d.]. Kotlin Grammar. https://kotlinlang.org/docs/reference/grammar.html

[10] Tobias Pfandzelter and David Bermbach. 2020. tinyFaaS: A Lightweight FaaS Platform for Edge Environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*. IEEE, Sydney, Australia, 17–24. https://doi.org/10.1109/ICFC49376.2020.00011

[11] Laure Philips, Wolfgang De Meuter, and Coen De Roover. 2015. Poster: Tierless Programming in JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 831–832. https://doi.org/10.1109/ICSE.2015.270

[12] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2019. Eliom: A Language for Modular Tierless Web Programming. *arXiv:1901.11411 [cs]* (Jan. 2019). arXiv:1901.11411 [cs]

[13] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. 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*. ACM Press, Portland, Oregon, USA, 975. https://doi.org/10.1145/1176617.1176756

[14] Daniel Sokolowski, Philipp Martens, and Guido Salvaneschi. 2019. Multitier Reactive Programming in High Performance Computing. (2019), 8.

[15] Josef Spillner. 2017. Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation. *arXiv:1703.07562 [cs]* (March 2017). arXiv:1703.07562 [cs]

[16] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 1–30. https://doi.org/10.1145/3276499

[17] Pascal Weisenburger and Guido Salvaneschi. 2020. Implementing a Language for Distributed Systems: Choices and Experiences with Type Level and Macro Programming in Scala. *Programming* 4, 3 (Feb. 2020), 17. https://doi.org/10.22152/programming-journal.org/2020/4/17

[18] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. 2018. Extend Cloud to Edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 373–377. https://doi.org/10.1109/SEC.2018.00048