National College of Ireland

Enabling scatter-gather communication between serverless function on a novel framework

Research Project
MSc Cloud Computing

Saurabh Kumar
Student ID: x18193188

School of Computing
National College of Ireland

Supervisor: Manuel Tova-Izquierdo

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Saurabh Kumar
Student ID:	x18193188
Programme:	MSc Cloud Computing
Year:	2020
Module:	Research Project
Supervisor:	Manuel Tova-Izquierdo
Submission Due Date:	17/07/2020
Project Title:	Enabling scatter-gather communication between serverless function on a novel framework
Word Count:	5348
Page Count:	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

I agree to an electronic copy of my thesis being made publicly available on TRAP the National College of Ireland's Institutional Repository for consultation.

Signature:	
Date:	16th August 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enabling scatter-gather communication between serverless function on a novel framework

Saurabh Kumar
x18193188

Abstract

Serverless computing is a concept in the cloud computing world where the developer writes code in any of the high-level languages with all its dependencies pushes it to the cloud platform for execution. The code written by the developer is called as serverless functions, these functions are executed on the trigger of an event. There are many providers of the serverless platform like the lambdas by Amazon Web Services, azure functions by Microsoft Azure and cloud functions by Google Cloud Platform are the well-known commercial enterprise level providers of the serverless architecture.

The most prevalent use case of the serverless platform is IoT and microservices. However, the serverless platform is incapable of handling distributed computing tasks that involve communication between the serverless functions. The communication patterns like scatter-gather and point to point communication are difficult to establish. As the address of the function spawned is unknown to the developer, communication between the functions is impossible.

Since the cloud computing infrastructure has access to vast resources and incomprehensive computing power, the power of distributed computation can be leveraged on the resources provided by the platform. As a part of the thesis, an artifact is developed which solves the scatter and gather problem on the serverless platform. The application could launch containers on uploading a file and distributed its chunks to the containers to process it parallelly and then get results. Using this approach, the serverless platform can be used for solving scatter and gather communication problems. This artifact shows that the address of the spawned nodes can be known to a master process.

This artifact is a library that shows that it is possible to establish communication between serverless functions, but it lacks features of container creation, keeping the system highly available and fault-tolerant.

1 Introduction

Serverless functions have become ubiquitous among developers and software development companies, they give the advantage of focusing on the core business logic of the application rather than spending money and hours on managing the servers and systems for smooth running. Most of the problems that the serverless platform caters to are in the domain of IoT and trigger-based applications. Microservices are the best example of trigger-based use case where a monolithic application comprising of many functionality are broken

down into individual services which promotes development teams to work on separate features and develop a refined product.

The cloud platform has a diverse and vast availability of resources, these resources should not be kept idle when its not used by the consumer application, therefore it has been argued in the academic community that why the serverless platform cannot be used to run distributed computing applications[1][2]. The serverless function running on the platform are short lived, i.e. the functions can be a process running on a random container that is scheduled by a container orchestrator like kubernetes[1] or mesos[2] depending on a algorithm that ranks nodes based on resource availability[3]. The address of the running process is not known to the application developer, which makes it difficult to write distributed applications that requires each of the processes to communicate with each other during the application execution. Figure 1 describes the communication patterns that are required for distributed computing applications to work on the serverless platform.

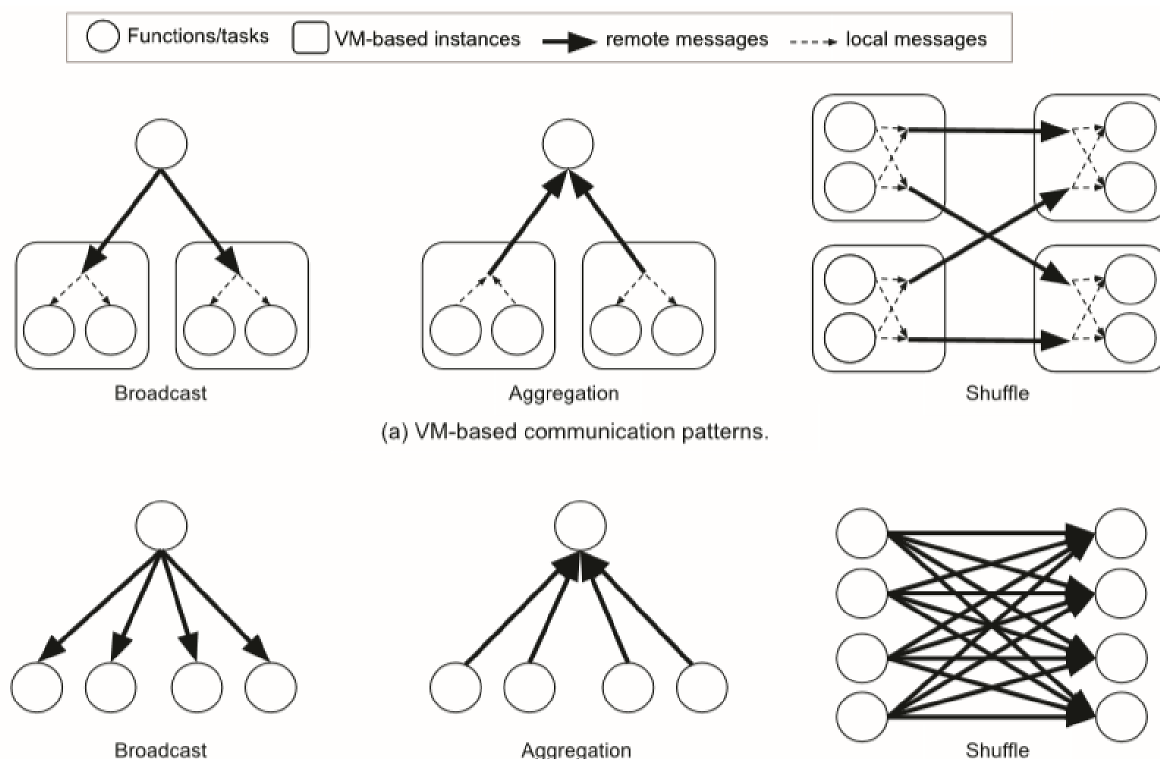

Figure 1: Communication patterns that are required for distributed computing applications on serverless[1]

The communication patterns like broadcast, scatter-gather and peer to peer communication is difficult to achieve when the address of the function is not known. There has been research conducted in the academic community for developing process aware manager that has knowledge of the address of the function spawned. PyWren[4] is one of the projects which uses distributed computation on the serverless architecture to run map-reduce type jobs. ExCamera[5] is another framework on the serverless platform

[1]k8's docs: `https://kubernetes.io/docs/home/`

[2]mesos docs: `http://mesos.apache.org/documentation/latest/`

for processing 4K quality videos. ggIR[6] framework that used distributed computation to create an intermediate representation of the libraries a software program uses, and containerize it so it can be deployed on the serverless architecture. There are many distributed processing frameworks like map-reduce - CIEL [7] and MARLA [8] on serverless platform in academia.

All the above frameworks discussed have communication between the worker and master/coordinator as a common problem. Since the functions are ephemeral, the address of the functions or process is important to communicate data and signals. There is no common daemon process or library that these applications use to find the address of the functions/processes in real time. There are several strategies discussed in the literature[9] [10] for establishing communication between serverless functions, they are actors, tuplespaces, pub/sub and distributed hash tables(DHTs).

It has also been claimed that direct communication between workers has been established by using the NAT-traversal techniques[6] on the AWS Lambda platform but the lack of features such as high availability and fault-tolerances make it impossible to use it for distributed computation. This problem begs and motivates the academic community to answer the following question

Research Question - Can the serverless platform be leveraged to develop process/function aware manager to support communication patterns such as broadcast, peer-to-peer and scatter-gather?

The artifact developed as a part of this thesis is aimed to prove that peer to peer communication can be established between master and worker nodes. The master node distributes data to the worker nodes and get the results back from them which shows a two-way communication between master and workers. There are some assumptions made for the full-functioning of the artifact.

It has been assumed here that replicas of the workers donot exist and the master is not working in highly available configuration. The scheduling of the workers is handled by kubernetes default scheduling algorithm. The worker process is pre-built into an image using docker which is dynamically loaded and started on the cluster by the artifact. The image doesnot contain any minimalist operating system. NATS[3] server is used which provides a communication paradigm that supports pub/sub as well as request-response type of communication between two communication processes on the cluster.

The code for the worker is written in golang[4] which complies to binary and doesnot need a virtual machine for execution, which makes the size of the image small i.e. 5.6 megabytes. NATS server is used for a two way communication between nodes on the cluster. The worker and the master should know the address of the NATS server and need to connect to it for the communication to take place. This paper will drill down into details on how the artifact works and the way this concept can be used by distributed computing applications in the cloud.

The remainder of the paper is divided into six sections. Section 2 contains related work which is further divided into five sub themes, Section 3 talks about methodology, Section 5 entails implementation and Section 6 compares the artifact with existing state of the art scatter-gather communication pattern. Section 7 talks about limitations and future works of the artifact.

[3]NATS docs: `https://docs.nats.io/`
[4]golang docs: `https://golang.org/doc/`

2 Related Work

Study of related work consist of different frameworks used in software open source community and academia. It helps in understanding the mapping of frameworks to the problems they are trying to solve. Moreover, it gives a thorough view into the advantages and limitations of the different frameworks. Additionally, it gives an insight into the design of a new framework that could leverage the serverless platform to allow processes/functions to communicate with each other.

2.1 Distributed computing frameworks on serverless

Fouladi *et al.*[5] claims to have developed a video processing framework on the serverless architecture that is capable of processing 4K and VR quality videos.

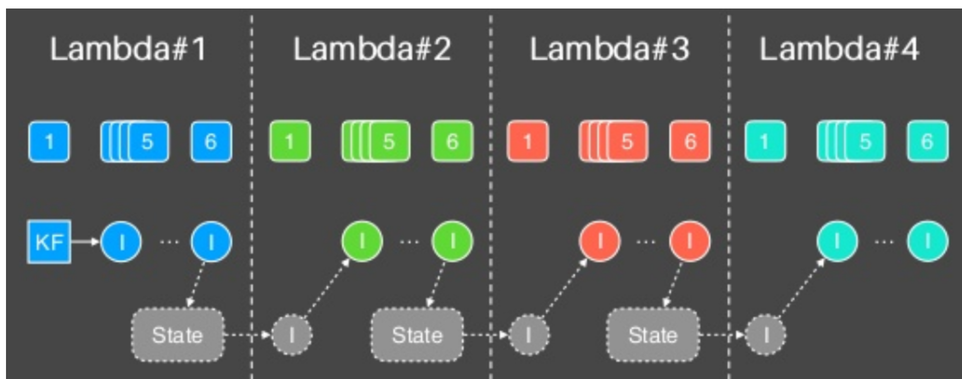

Figure 2: Linking state of one lambda to the next[5]

It can be clearly seen from the results of this paper that the mu framework can solve the video-editing problem on the serverless platform. However, can the mu framework be leveraged to solve the problem with the communication pattern that is required for high performance computation? Since it is a video editing framework, a few missing data points can be tolerated, but can this approach be adopted by the distributed computing paradigm. The authors have not clearly described the fault-tolerance and high availability mechanisms that can be used to recover from failures and lost processes. Is the framework aware of the data locality while spawning threads? These are the few questions that need to be further answered for this framework to be used for the distributed computing problem.

L. Ao *et al.*[11] claims to have developed a video processing platform similar to ExCamera that uses a modified version mu framework internally. It consists of the coordinator but not a rendezvous server. The coordinator is a long running server that runs on the user platform in VM or a container. When a video file is uploaded it sends request to the lambda gateway to launch the processes. Each process is coupled with a daemon process which is controlled remotely the coordinator, the figure 3 shows the sequence of control flow of how the coordinator communicates with the daemon process using remote procedure calls to get the status of the lambda process.

The sprocket framework is a good contender of video processing framework on the serverless platform but lacks the communication pattern that could be employed in high performance communication. The cloud functions that process the distributed chunks of

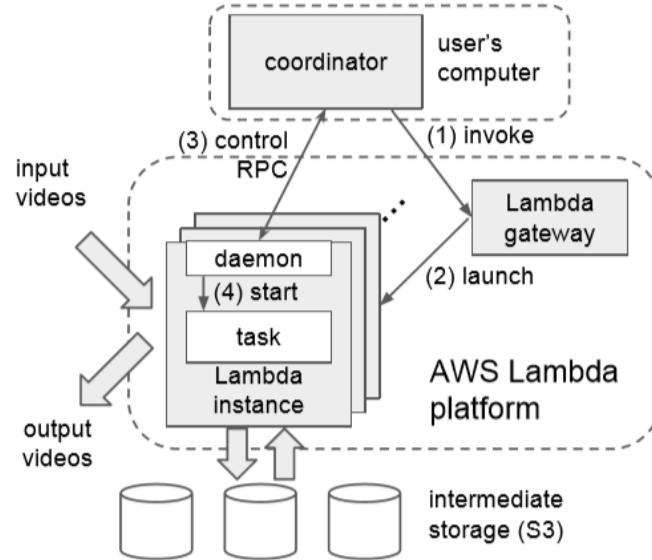

Figure 3: Sprocket video processing framework on serverless

the video file store the partial output to an intermediate storage. According to the paper there is a significant delay in storing of the intermediate data on the available storage on the serverless platform, then linking all the partial outputs adds to the delay. Remote procedure calls (RPCs) have been used for communication between the master and the worker process. This particular approach is good for scenarios where video processing involved but will not work for communication patterns related to distributed computation on scale.

Jonas *et al.*[4] and his team have developed a map-reduce like framework in python on the serverless platform. The PyWren[4] framework is implemented on burst synchronous processing (BSP) model. According to this model, multiple tasks are processed in parallel. As we know that a Map-Reduce phase consists of shuffle operations at various stages e.g. at the time of writing from buffer memory to disk and when map outputs are picked from different sources and sorted. Shuffle intensive workloads tend to induce latency and consume network bandwidth.

Since the user doesnot have an agreement with the cloud service provider to view the location of the stored data for security reasons, the data local map jobs cannot be run. This framework is not good for tasks that are long running and require coordination between them. The PyWren model re-spawns a thread when one fails, it processes on the data from the beginning and not from the state where it failed. Therefore, it doenot have a stateful failure recovery mechanism, which adds up to the processing time. The threads that are spawned are processes that donot communicate the progress to the master, they simply write the results to the intermediate storage. As there is no communication between the threads, there is no way to know the progress of the tasks. Therefore, it is required to build a system where the master process running in highly available configuration is able to know about the progress of the processes running on the serverless platform.

Shi *et al.*[12] claims that map-reduce and spark application frameworks have different use cases. The paper claims that Spark is faster than MapReduce by a factor of 2.5x, 5x and 5x for word count, k-means and PageRank problems. However, the sort phase

during reduce phase for MapReduce is faster at 2x than Spark.

There are trade-offs between parallelism and context switching, in-memory and on-disk caching, serialization and memory consumption. The above parameters can help to design a better framework for running serverless mapreduce[8]. The communication between the worker and master is an important part of the distributed computing platform, such types of communication patterns should be supported by the serverless architecture.

Giménez-Alventosa *et al.*[8] has studied the viability of MapReduce execution model on serverless platform AWS Lambda, the author and his team have developed a MapReduce like distributed processing framework in python called MARLA (MapReduce on AWSLambdas).

In his paper, the author claims that MARLA is an improvement over the previous map-reduce frameworks on serverless platform such as PyWren[4], Corral[5] and Ooso[6]. The results of the study show that the MARLA framework cannot handle failure recovery and coordination activities without the need for a local application manager. However, this framework is still tightly constrained to solve problems concerned with map-reduce problems. There is no mention of the progress updates of workers that is sent of the manager on regular intervals. Nevertheless, the study of this framework has given deep insight into the development of framework that supports different communication patterns on serverless.

Aytekin *et al.*[13] and Johansson *et al.*[13] have developed master-worker framework on AWS Lambda which is capable of solving large scale optimization problems. It uses parallelly working workers with coordination between them. The authors have elaborately identified system-level bottlenecks and limitations and proposed improvements and solutions.

The framework developed by the authors can achieve significant performance gains which is relative speedups up to 256 workers and an efficiency of 70% up to 64 workers. Still the framework is unbale to answer the statelessness of the serverless functions, as the algorithm requires to maintain state running for long time. The serverless functions cannot accept inbound connections which is another problem, therefore distributed communication protocol like scatter-gather and point-to-point communication is difficult to achieve.

2.2 Application containerization on serverless in academia

Weinstein *et al.* [6] and his team claims to have successfully addressed the most prominent problems of the serverless architecture like limited runtime of the workers, limited on-worker storage memory, constraint on the number of workers provided by the cloud provider, failures of worker at runtime and reusability of libraries on applications on the cloud.

The workaround to address the above challenges was to develop a custom application containerization abstraction called as "Thunk" which is representation of a linux x86-64 container which can be scheduled and executed on the cloud functions. While the dependency management strategy of the framework helped to gain huge benefits, it does not provide any answer to the direct communication between workers. In fact, the paper has stated its limitations clearly that this framework is suitable for burst parallel processing

[5]github repo: `https://github.com/bcongdon/corral`
[6]github repo: `https://github.com/d2si-oss/ooso`

but not embarrassingly parallel jobs. As serverless functions cannot accept incoming connections [9], a mechanism should be devised to enable communication between them once their address is known.

Oakes *et al.* [14] claims that the high-level languages tend to make a bigger software footprint on the serverless platform, the author claims that since high level languages tend to run a virtual machine, their compilation time and bytecode is more the low level languages such as C. The results clearly state that the strartup of an application written in higher level language is 10X slower than its equivalent written in C.

Lambdas are containers which are running in an isolated manner which have their sandboxing overheads[15]. Typically, serverless platforms have to wait for hours or minutes for de-allocation of resources if the functions are idle or un-billed[16]. If the resources are de-allocated quicker, it will help reduce the cold start-up time of application containers. The author has developed an application containerization framework called SOCK[14] to address the above limitation, the drawback of SOCK is that it's caching of the dependency libraries is only supported for python and not other high level languages. This paper does not provide any insight into the type of communication patterns that can be used to leverage distributed computation on serverless platform.

Perez *et al.* [17] have designed an application container on docker in a high level language with dependencies in a minimal memory footprint. The framework is called SCAR [17]. The author claims that it supports deployment of scientific application that have dependencies. Moreover, it also supports languages that are not natively supported by serverless platform. Since the application is containerized, feature such as auto-scaling can be leveraged.

The limitation of the SCAR framework is the AWS Lambda platform on which it is run. AWS Lambda only supports 5 minutes of execution time, restricted memory (3008 MB) and limited disk capacity of 512 MB. Moreover, SCAR is highly dependent on udocker for image creation and execution on the serverless platform. The other drawback is python support, since udocker is developed in python, it needs a python runtime. The framework supports communication between docker containers in the user space, this idea can be leveraged to design serverless functions that allow communication between them to support distributed and parallel computation.

2.3 OpenSource serverless platforms

Li *et al.* [18] Kulkarni *et al.* [18] and Ramakrishnan *et al.* [18] tried to compare the opensource platforms like Knative [7], Kubeless [8], Nuclio [9] and OpenFaas [10] for serverless computing. These frameworks have been evaluation against the three parameters i) role and interaction of different components ii) configuration parameters iii) mode and operation of auto-scaling.

The results of the baseline performance tests show that Nuclio performs much better than the other counterparts. Concurrent request handling capability of Nuclio is much better than the other frameworks. Auto-scaling strategies are insufficient to meet the increasing workload demand. Also, study and benchmarks of these frameworks are conducted from the perspective of running microservices and trigger-based applications. The

[7] github repo: `https://github.com/knative`
[8] `https://kubeless.io/`
[9] github repo: `https://github.com/nuclio/nuclio`
[10] github repo: `https://github.com/openfaas/faas`

study has not conducted any tests to show the viability of distributed computation on these platforms as well as no benchmarks of communications between these functions have been shown in the results.

Hendrickson *et al.*[19] and Sturdevant *et al.*[19] and team have claimed to develop a serverless platform for research called OpenLambda[11] where researchers can modify the code of the vanilla serverless platform for conducting workload studies, developing optimized package management capabilities for high level language, optimizing execution engines like docker containers, sandboxing and just-in-time containers.

This platform can also be used for experimenting with the communication pattern for scatter and gather on OpenLambda, the authors have mentioned it as one of the topics open for research. Driving inspiration from OpenLambda, an artifact has been developed for the purpose of this research on Kubernetes platform which serves as a multi-cloud worker orchestration and management engine.

Kuntsevich *et al.* [20], Nasirifard *et al.* [20] have investigated bottlenecks and limitations of the Apache OpenWhisk[12] serverless platform. The results of their study shows that auto-scaling guarantees cannot be offered by the framework. Benchmarks were carried out to evaluate the performance of CPU intensive task such as calculation of Nth prime number, memory consumption was tested using matrix multiplication, I/O bottlenecks were measured loading it with HTTP requests.

Since the serverless applications are used for microservices and trigger-based applications, the openwhisk platform was compared with a springboot[13] microservice, the results show that there is a significant latency on the serverless platform and requires modifications and changes to increase performance. For the purpose of the artifact development, the openwhisk platform makes a strong contender.

3 Methodology

The study of the different frameworks on the serverless platform gives an insight into their limitations and strengths. It has been seen that frameworks such as PyWren[4] and ExCamera[5] uses a third party framework "mu" which has coordination and rendezvous server for communication between two functions on the serverless platform. MARLA [8] uses a fire and forget kind of an approach where the failures of the map processes are unaccounted. Moreover, there is no communication between the master and worker processes. The processes are spawned on trigger without any communication about the updates.

It is known that the serverless functions are spawned on the trigger of an event. There might be a function or many replicas of the same function in a highly available configuration being launched. The address of these functions is unknown to the programmer/developer. To develop distributed applications on the serverless platform, it is important to know the address of the functions spawned so they can communicate.

From the study in section 2, it has been seen that most peer to peer or master to worker communication is done using NAT hole punching. It's been described in the figure4 the steps involved in UDP NAT hole punching. NAT hole punching[21] requires a rendezvous server to get the public address in order to communicate with the other

[11]OpenLambda: `https://github.com/open-lambda`

[12]OpenWhisk: `https://openwhisk.apache.org/`

[13]Springboot: `https://spring.io/projects/spring-boot`

peer behind the NAT. The rendezvous server can be TURN[22] or a STUN[23] server depending upon the requirement. Since this is the only medium of communication in the serverless domain that has been discussed so far, there is a need for a new communication paradigm that could support the communication between serverless functions.

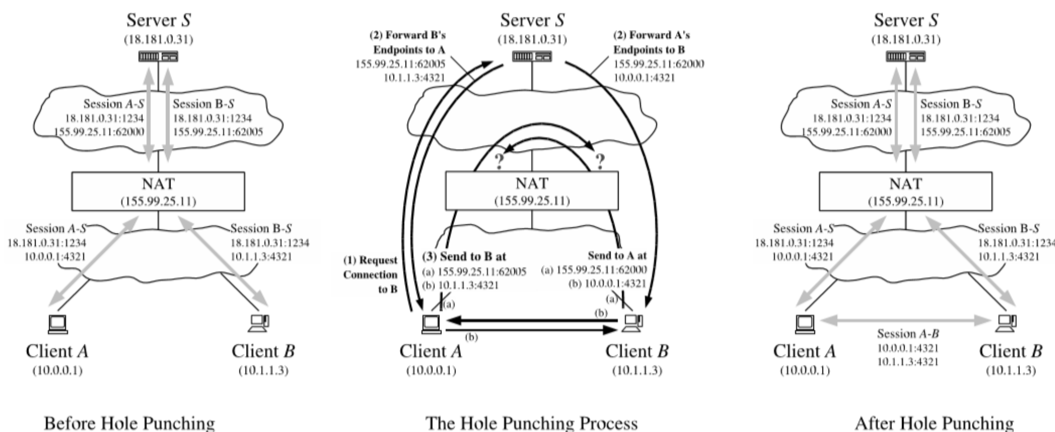

Figure 4: NAT hole punching[21]

Scatter and gather is the most rudimentary communication pattern that is required in the world of distributed computation, for leveraging distributed computation on the serverless platform, it is required to develop a framework that has knowledge of the addresses of the functions being launched so that the master process can communicate with the worker and perform scatter and gather kind of operations.

The artifact developed is responsible for keeping a list of the addresses of the functions spawned, it is also instrumental in letting know the user/developer change in the status of the container in which the process is running. Figure 5 shows the components used to achieve scatter-gather communication.

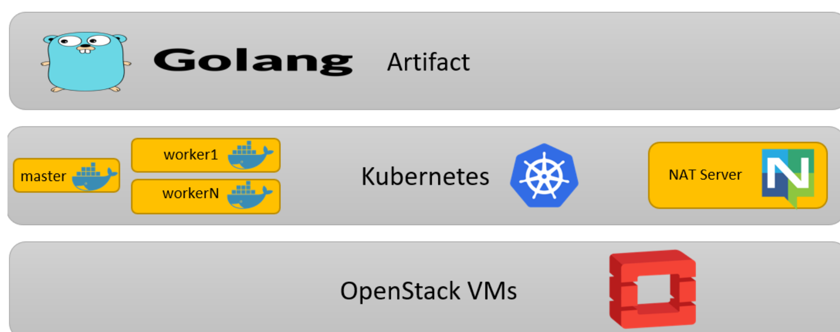

Figure 5: Components used in artifact

For achieving scatter-gather communication pattern on the serverless, the following communication constructs will be used as per the new artifact.

- Request/Response
- Publisher/Subscriber

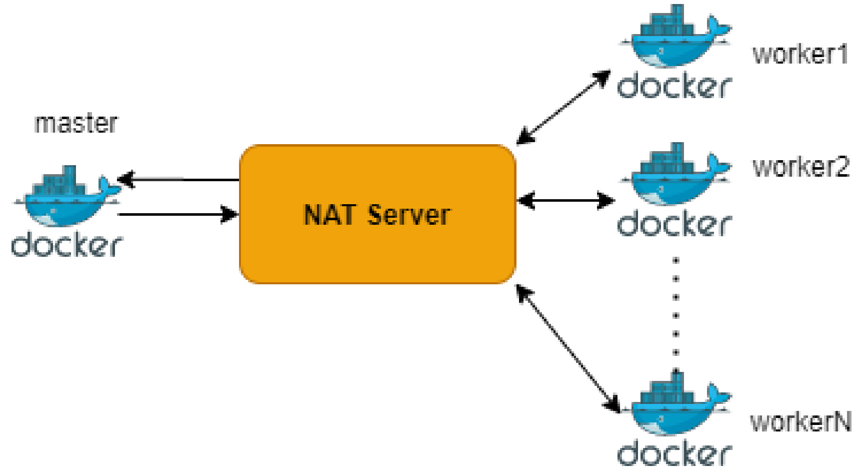

Figure 6: Scatter-gather communication between master and workers using NATS[3] server

The distinguishing feature of the artifact and the way it is different from other frameworks discussed in the literature review is that it has a controller for listening to all the events from the Kubernetes api server. The api server of the Kubernetes platform launches containers on pods, the event is captured by the artifact to maintain addresses of functions launched. Figure 6 shows how scatter-gather communication will be achieved.

The functions that are spawned is master and worker functions. For simplicity and for the development of the artefact, a file is submitted and partitioned into chunks by the master and distributed to its workers, the workers find the wordcount and submit the results back to the master. The distribution of the chunks acts as a scatter pattern and returning of the results acts as a gather pattern.

The NATS[3] server is instrumental in achieving scatter-gather communication pattern. How does it do that? The artifact spawns worker function depending on the number of file chunks created. When it is spawns a worker, a topic name is supplied as argument parameters to which the worker must listen to. The master process has a list of the argument parameters, it used them to rendezvous with all the workers, thus achieving scatter-gather communication pattern.

4 Design Specification

The artifact is designed for enabling scatter-gather communication pattern on a developed serverless platform, It shows that with the use of the components mentioned in section 3, a master can be launched that has the information about the addresses of the workers. This concept can be further leveraged on to other open-source serverless platforms for distributed computations. The next few sub-sections contain technical diagram which provide a deep insight into how the artifact is designed.

4.1 Class Diagram

Figure 7 represents class diagram of different entities in the artifact.

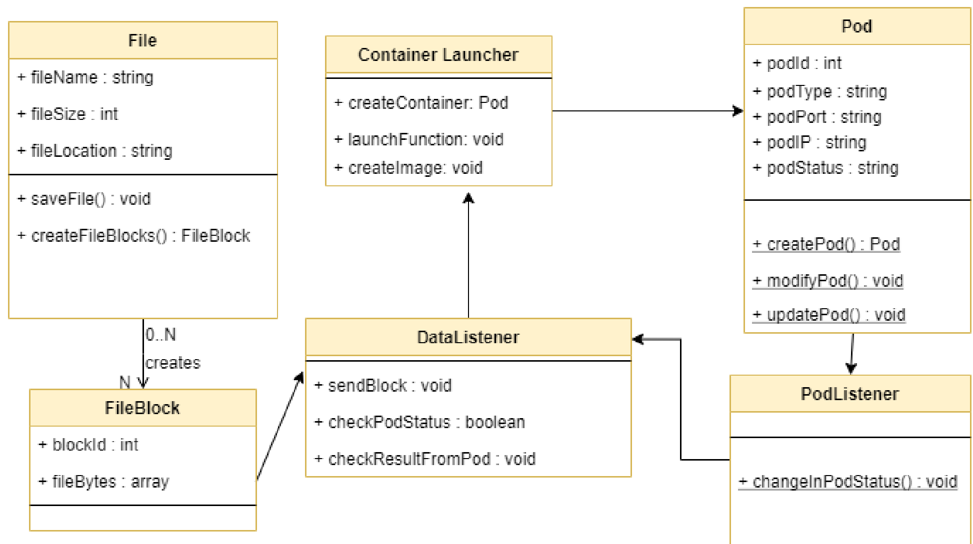

Figure 7: Class diagram of the artifact

4.2 Component diagram

Figure 8 refers to the components of the artifact. It shows the components that have been used in the development of the artifact and their interaction with each other.

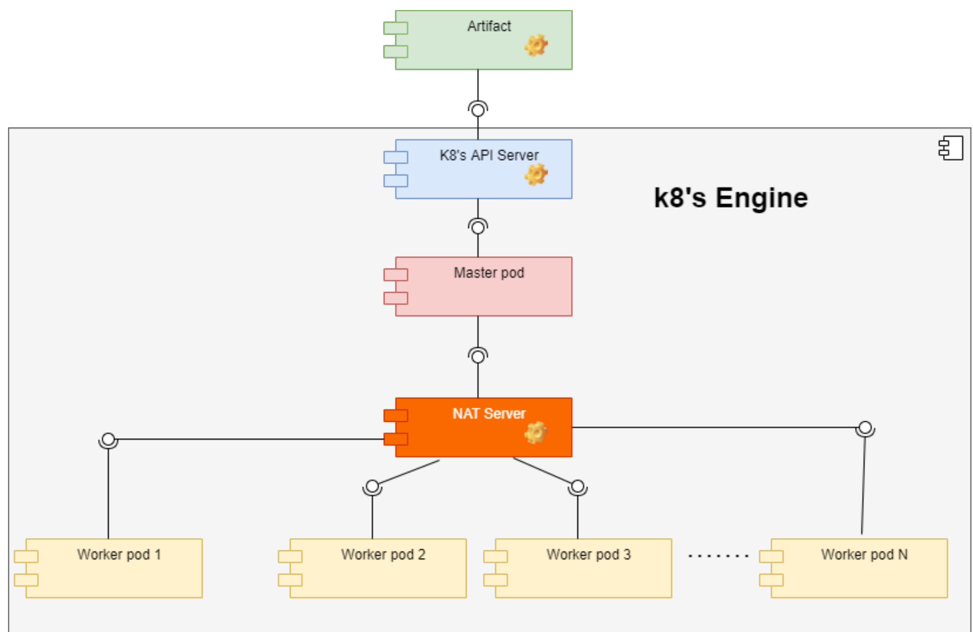

Figure 8: Component diagram of the artifact

4.3 Sequence diagram

Figure 9 shows the sequence diagram of the artifact, here moving from left to right, we see that a file is uploaded of a certain size, the request reaches the controller, a master function is launched which calculates the number of chunks in which the file is divided.

Depending on the number of chunks, equal number of worker nodes are launched by the artifact, since all these workers are containers inside pods, they take time to initialize, they keep sending status report to the artifact, when they are in running status, the data chunk is sent to the individual workers for execution. When the execution is finished by the worker the result is sent back to the master function.

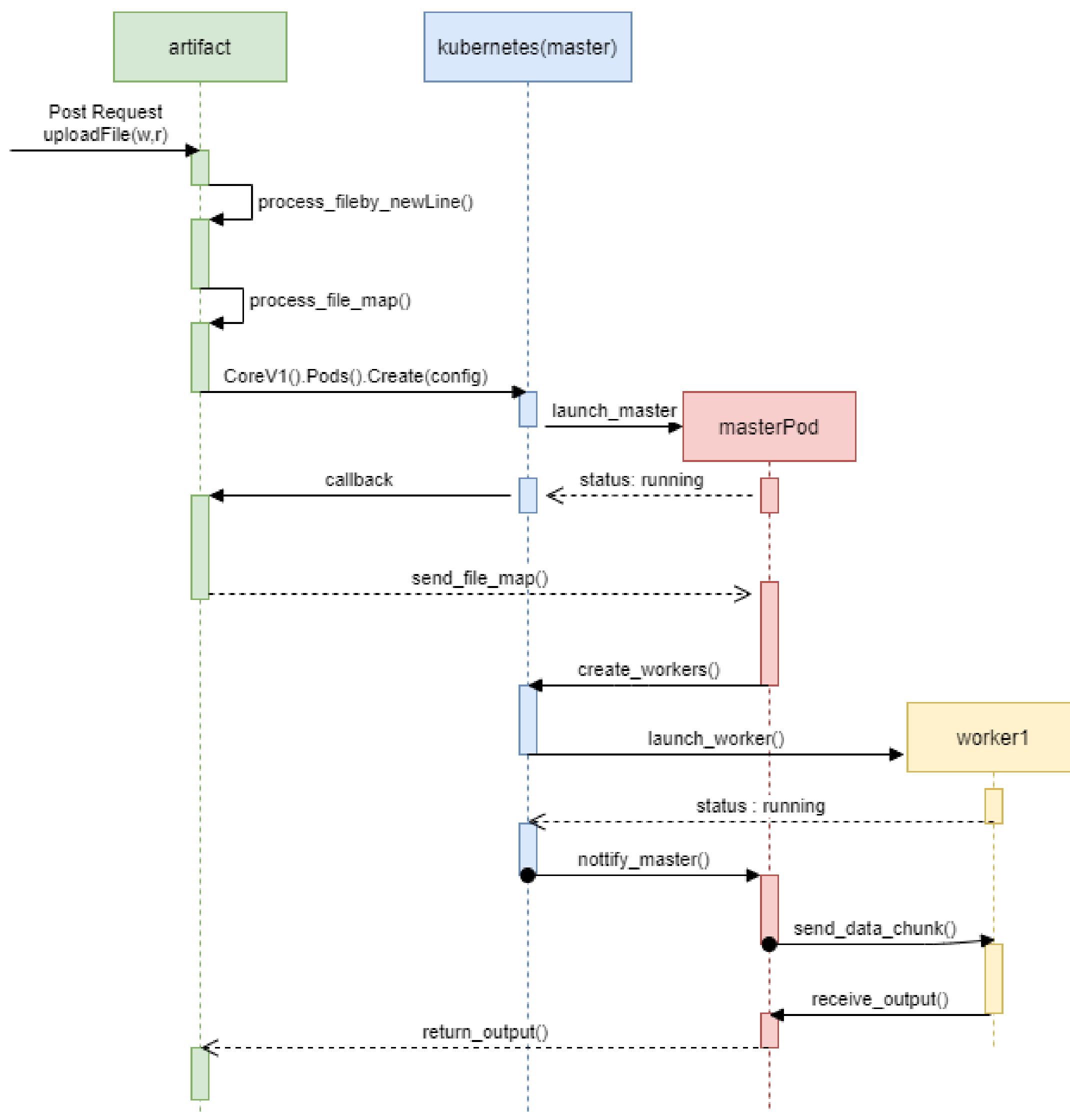

Figure 9: Sequence diagram of the artifact

4.4 Algorithms

The artifact consists of a master and worker functions and their interaction between them in a scatter-gather form of communication, the algorithm related to master and worker

functions are described below.

Algorithm 1: Algorithm for creating file blocks to be processed by workers

---

```
Data: file of a definite size
Result: map of file id and its contents in bytes
numberOfBlocks, fileByteSize, byteCount = 0, blockByteSize = 300, lineLength
 = 1000;
line = byte[lineLength];
numberOfBlocks = fileByteSize / blockByteSize;
if fileByteSize%blockByteSize != 0 then
|   numberOfBlocks = numberOfBlocks + 1;
else
|   m
end
apper = map(int,[]byte) ;
eofStatus = scan first line of file;
count = 0;
while eofStatus! = false do
    line = append(line, newLineCharacter);
    byteCount = byteCount + len(line);
    if byteCount == blockByteSize then
        addByteToSlice(line, blockSlice);
        if count ≤ numberOfBlocks then
            mapper[count+1] = blockSlice;
            count++;
        end
        blockSlice = 0;
        byteCount = 0
    end
    if byteCount ≥ blockByteSize then
        if count ≤ numberOfBlocks then
            mapper[count+1] = blockSlice;
            count++;
        end
        blockslice = 0;
        addByteToSlice(line, blockSlice);
        byteCount = len(line);
    end
    addByteToSlice(line, blockSlice);
    eofStatus = scanner.Scan();
end
mapper[count+1] = blockSlice;
return mapper;
```

Algorithm 1 refers to the functionality of the master function. The master function receives the file which is broken into chunks based on a configurable value as shown in the algorithm represented by the variable blockByteSize, the default value is taken to be 300. The blocks of bytes are stored in a map where the key is an unique integer and the value is an array of bytes that is sent to worker for processing. The complexity of the

algorithm is depends on the number of bytes read by the function which is $O(n)$.

Algorithm 2: Algorithm for starting worker and watching them for status changes, and listening for results from workers

```
Data: map of id and bytes
Result: Launch workers and watch them for their status
i=0;
for i ← 1 to len(map) do
  | start a worker function with cmd argument parameter workeri ;
end
i=0;
for i ← 1 to len(map) do
  | if workerStatus == running then
  |   | send byte chunk to the workers for processing.;
  |   | set flag = true for worker i;
  | end
end
i=0;
for i ← 1 to len(map) do
  | listen to data from worker i;
end
```

Algorithm 2 refers to the master spawning worker containers on the kubernetes engine, the image is pulled from docker private registry and run on pods. The status of the container keeps changing, when the container reaches the running status, the data can be transferred for processing. Once the data reaches the workers, it is processed and returned to master function, the master function has to listen in a callback function, which is triggered once the data arrives from one of the spawned workers. The complexity of the algorithm is $O(n)$.

Algorithm 3: Algorithm for worker function that is accepting data bytes for determining word count

```
Data: bytes of data
Result: return a string of word count
s = convert bytes into string;
arrayOfLines = split s by "newLineCharacter";
wordBank = empty;
i=0;
for i ← 1 to len(arrayOfLines) do
  | arrayOfWords = split arrayOfLines[i] by " ";
  | wordBank.add(arrayOfWords);
end
wordMap = new map;
for i ← 1 to len(wordBank) do
  | if wordMap[i] == null then
  |   | wordMap[i] = 1 ;
  | end
  | wordMap[i] = wordMap[i] + 1;
end
return wordMap
```

Algorithm 3 refers to worker functions which is waiting for data from the master function. As it receives the data, it finds the word count and returns a map of word count. The complexity of the algorithm is $O(n)$.

5 Implementation

The following open-source projects will be used for developing the artifact that ensures scatter-gather communication patterns between master and the worker on a serverless platform.

- Docker[14] - It is used for wrapping function with its dependencies into an image, which is pulled by the k8's engine and run as containers.
- Kubernetes[1] - It is used for scheduling functions on its platform and sending the status of lunched containers to the artifact.
- golang[4] - Programming language used for the development of all the components, golang doesnot need virtual runtime environment, code directly compiles to machine code which makes it fast. Moreover, the dependency management of libraries used is optimized and compilation time is significantly reduced.
- linux[15] - All the components are installed on linux VMs.
- NATS[3] - NATS server is used as a messaging queue with request and response support, which helps in achieving scatter and gather communication pattern in a master and worker configuration.
- Openstack[16] - It is a opensource cloud platform that is capable of launching linux VMs of any configurable size.

The source code of the artifact is hosted on github[17] with a detailed readme.md file and the artifact is deployed on openstack.

The algorithms used for the artifact 1, master function 2 and worker function 3 are described in section 4.

```
saurabh@ubuntu:~/workspace/artifact$ kubectl get pods --watch
NAME           READY   STATUS    RESTARTS   AGE
natserverpod   1/1     Running   5          19d
```

Figure 10: NATS acting as a publisher/subscriber as well as request/response server

Figure 10 shows the state of the kubernetes cluster before application are launched by the artifact. It only consists of the nats server running which is waiting from connection from master and worker functions.

Figure 11 shows the state of the kubernetes cluster after application are launched by the artifact.

[14] docker docs: https://github.com/docker

[15] linux docs: https://github.com/torvalds/linux

[16] openstack docs: https://docs.openstack.org/ussuri/

[17] artifact source code: https://github.com/saurabh7517/thesis

Figure 11: Different states of the container before they are actually running

6 Evaluation

The artifact is run as a server that accepts a text file, It is responsible for decomposing the text file into a configurable size, if the size is small the number of blocks created is large, we will be conduction experiments with different sizes and check the time taken. This time evaluated using these experiments will be compared with other scatter-gather techniques. It is assumed that Kubernetes platform is pre-installed on a cluster of 3 nodes each having 2GB of RAM, 10 GB hard-disk space, 2.5 Ghz processor and Ubuntu-server as the operating system.

For the purpose of our experiments, the file size will be kept constant and the block sizes will be changing. The file size for the purpose of the experiments will be 1654 bytes. The number of functions launched are dependent on the block size. The algorithm for number of blocks has been stated in algorithm 1.
numberOfBlocks = fileSize/blockSize
So, it is seen that the number of functions spawned are dependent on the file size and block size. Below are some console outputs of the services that are running before and after application run on the kubernetes orchestrator.

Figure 12: Response from different workers with their result sent to master function

Figure 12 contains the response of word count calculated by the workers launched by master function. Each worker after calculation of the word count sends the result to the

master function.

6.1 Experiment / Case Study 1

For the first experiment, the following parameters are taken for file size and block size.

- file size = 1,654 bytes
- block size = 300
- pods created = 6

```
Changing isPodRunning status to true
Elapsed time for scatter-gather communication is 4.579534217s
```

Figure 13: Time calculation given the above parameters

Figure 13 shows the time taken for the master function to distribute tasks to worker function and get back results from each worker function.

6.2 Experiment / Case Study 2

For the second experiment, the following parameters are taken for file size and block size.

- file size = 1,654 bytes
- block size = 150
- pods created = 11

```
Changing isPodRunning status to true
Elapsed time for scatter-gather communication is 5.874750345s
```

Figure 14: Time calculation given the above parameters

Figure 14 shows the time taken for the master function to distribute tasks to worker function and get back results from each worker function.

6.3 Experiment / Case Study 3

For the third experiment, the following parameters are taken for file size and block size.

- file size = 1,654 bytes
- block size = 100
- pods created = 17

Figure 15 shows the time taken for the master function to distribute tasks to worker function and get back results from each worker function.

```
Changing isPodRunning status to true
Elapsed time for scatter-gather communication is  10.099787701s
```

Figure 15: Time calculation given the above parameters

6.4 Experiment / Case Study 4

For the fourth experiment, the following parameters are taken for file size and block size.

- file size = 1,654 bytes
- block size = 50
- pods created = 35

```
Changing isPodRunning status to true
Elapsed time for scatter-gather communication is  18.767616321s
```

Figure 16: Time calculation given the above parameters

Figure 16 shows the time taken for the master function to distribute tasks to worker function and get back results from each worker function.

6.5 Discussion

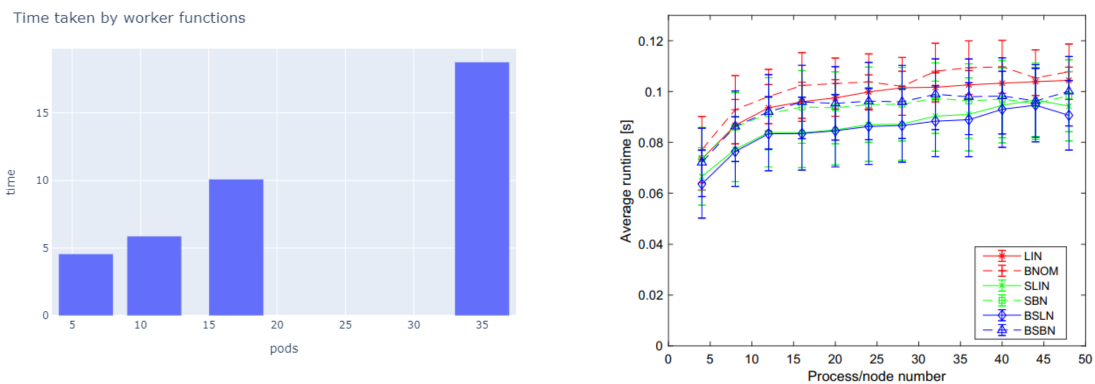

Figure 17: Left Image : time vs worker functions launched Right Image : time vs process nodes of intranode scatter-gather communication in MPI[24]

Figure 17 shows two graphs depicting scatter-gather communication patterns, the left side graph shows a plot of the functions spawned vs the time taken by the artifact. The graph on the right is about intranode scatter-gather communication, the plot is between number of processes and time taken using MPI library[24].Figure 18 suggest that the scatter-gather latency on the artifact is much more than in using the MPI library [25].

The above experiments suggest that the time is exponentially increasing as the number of worker nodes are increased linearly. The exponential increase in time can be attributed to the time taken by the Kubernetes engine to request docker to create an image of the worker functions. It is also ensured here that the image created is of minimal size and with bare minimum dependencies, the image of the worker function is 5MB in size.

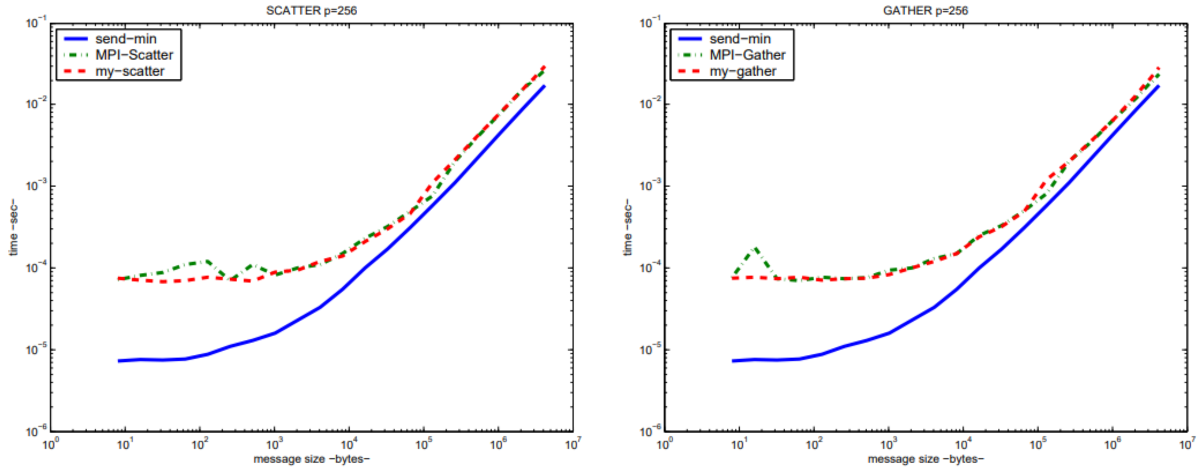

Figure 18: Time taken in scatter-gather communication in MPI[25]

The reduced size is taken so that it has minimum memory footprint while running on the docker containers. After the images are created, they are pulled by the Kubernetes engine through a private repository and started on the Kubernetes cluster. The Kubernetes cluster is instrumental in sending regular heartbeats to the master function with the addresses of the worker functions, this communication makes the master aware of the worker functions failed or newly spawned. Though the artifact solves the scatter-gather communication problem, but the evaluation statistics shows more feasibility tests to be conducted.

7 Conclusion and Future Work

The objective of the thesis enables the communication between serverless function by developing a novel artifact. The artifact developed solves the scatter-gather communication pattern between a master and worker functions. To show the scatter-gather communication, a file is divided into chunks of configurable size called blocks, the number of blocks created is equal to the number of worker functions spawned by the master. The block of data is distributed to the worker functions. This phase is the scatter phase. The worker function processes this data and send the results back to the master. This is the gather phase. The implementation shows the development of the artifact and the components that have been used in its development.

Though the artifact solves the scatter-gather communication problem on the serverless platform, it requires the availability of a orchestrator engine like Kubernetes, mesos or docker compose in order to be aware of the functions spawned. Another dependency of the artifact is the NATS server which is used for scatter-gather type of communication between master and worker functions.

The experiments in the evaluation shows that as the number of workers increase linearly the time taken for spawning them increases exponentially, which puts questions on the feasibility of the framework. Further research and experiments needs to be conducted to reduce cold start latency. The reduction in latency will provide a huge leverage to this framework to be used commercially by scientific computation and distributed computation which have sub operations of scatter-gather.

References

[1] 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," 2019.

[2] H. Shafiei and A. Khonsari, "Serverless computing: Opportunities and challenges," *ArXiv*, vol. abs/1911.01296, 2019.

[3] Z. Wei-guo, M. Xi-lin, and Z. Jin-zhong, "Research on kubernetes' resource scheduling scheme," in *Proceedings of the 8th International Conference on Communication and Network Security*, ser. ICCNS 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 144–148. [Online]. Available: https://doi.org/10.1145/3290480.3290507

[4] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. Santa Clara, California, USA: Association for Computing Machinery, 2017, p. 445–451, impactfactor : 5.550. [Online]. Available: https://doi.org/10.1145/3127479.3128601

[5] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA, USA: USENIX Association, Mar. 2017, pp. 363–376, ranking : B. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[6] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 475–488.

[7] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: A universal execution engine for distributed data-flow computing," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, p. 113–126.

[8] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless mapreduce on aws lambda," *Future Generation Computer*

Systems, vol. 97, pp. 259 – 274, 2019, impactfactor : 4.639. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18325172

[9] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *CoRR*, vol. abs/1812.03651, 2018, citedby:13. [Online]. Available: http://arxiv.org/abs/1812.03651

[10] D. Barcelona-Pons, Á. Ruiz, D. Arroyo-Pinto, and P. García-López, "Studying the feasibility of serverless actors," in *ESSCA'18*, ser. CEUR Workshop Proceedings, vol. 2330. CEUR-WS.org, 2018, pp. 25–29. [Online]. Available: http://ceur-ws.org/Vol-2330/short1.pdf

[11] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, p. 263–274, citedby : 10. [Online]. Available: https://doi.org/10.1145/3267809.3267815

[12] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proc. VLDB Endow.*, vol. 8, no. 13, p. 2110–2121, Sep. 2015. [Online]. Available: https://doi.org/10.14778/2831360.2831365

[13] A. Aytekin and M. Johansson, "Harnessing the power of serverless runtimes for large-scale optimization," *CoRR*, vol. abs/1901.03161, 2019. [Online]. Available: http://arxiv.org/abs/1901.03161

[14] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Sock: Rapid task provisioning with serverless-optimized containers," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 57–69, ranking : A.

[15] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 218–233. [Online]. Available: https://doi.org/10.1145/3132747.3132763

[16] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 133–145.

[17] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Future Generation Computer Systems*, vol. 83, pp. 50 – 59, 2018, impactfactor : 4.639. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17316485

[18] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in *Proceedings of*

the 5th International Workshop on Serverless Computing, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 37–42. [Online]. Available: https://doi.org/10.1145/3366623.3368139

[19] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, Colorado, USA: USENIX Association, Jun 20-21 2016, ranking : A. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[20] A. Kuntsevich, P. Nasirifard, and H.-A. Jacobsen, "A distributed analysis and benchmarking framework for apache openwhisk serverless platform," in *Proceedings of the 19th International Middleware Conference (Posters)*, ser. Middleware '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–4, ranking : A. [Online]. Available: https://doi.org/10.1145/3284014.3284016

[21] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 13.

[22] P. Matthews, J. Rosenberg, and R. Mahy, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," RFC 5766, Apr. 2010. [Online]. Available: https://rfc-editor.org/rfc/rfc5766.txt

[23] P. Matthews, J. Rosenberg, D. Wing, and R. Mahy, "Session Traversal Utilities for NAT (STUN)," RFC 5389, Oct. 2008. [Online]. Available: https://rfc-editor.org/rfc/rfc5389.txt

[24] J. Proficz, "Process arrival pattern aware algorithms for acceleration of scatter and gather operations," *Cluster Computing*, Jan 2020. [Online]. Available: https://doi.org/10.1007/s10586-019-03040-x

[25] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1206