A Survey of Serverless and Virtual Actor Frameworks

A Thesis

presented to

School of Applied Computing, Faculty of Applied Science and Technology

of

Sheridan College, Institute of Technology and Advanced Learning

by

Alec DiVito

in partial fulfilment of the requirements

for the degree of

Honours Bachelor of Computer Science (Mobile Computing)

December 2021

A Survey of Serverless and Virtual Actor Frameworks

by

Alec DiVito

Submitted to the School of Applied Computing, Faculty of Applied Science and Technology
on December 13, 2021, in partial fulfillment of the
requirements for the degree of
Honours Bachelor of Computer Science (Mobile Computing)

Abstract

In the past decade, we've seen increase popularity in developing systems that are able to easily and quickly become a distributed system. With this popularity we've seen the development of applications such as serverless frameworks, distributed key value stores, distributed stream processing frameworks and distributed actor frameworks. From these developments, it's clear that developers want to have access to frameworks which abstract away the distributed system which would allow them to write applications without the overhead of needing to understand how the distributed system works. There is also a want for application state to be as close to compute power as possible because moving data is expensive. With these requirements, stateful serverless applications and distributed actor models are the answer to what developers wants.

In this paper, we reviewed stateful distributed computing frameworks and services. We then complete a survey on the performance of different popular virtual actor frameworks. In our research we focus on the performance of 2 popular actor model frameworks running on top of the popular container orchestration system, Kubernetes. The frameworks we reviewed are Microsoft Orleans and Proto.Actor. From our 3 distributed and virtual microbenchmarks we evolved, we found that Proto.Actor preforms the best while Orleans has more robust documentation and community around it.

Keywords: actor model, serverless computing, orchestration, distributed computing

Thesis Supervisor: Dr. Ed Sykes
Title: Director, Centre for Mobile Innovation

Acknowledgments

Thanks Ed for all your patience and giving me time to learn how too research and write.

Contents

List of Figures

Listings

Chapter 1

Introduction

Developers have been searching for easier ways to create applications that effortlessly scale when ran in a distributed environment. One of the areas where we have seen large investment by cloud providers is in the serverless computing space. The advancement in serverless has been noticed by developers and companies as the adoption rate for serverless has been growing year over year [38, 26]. Serverless promises to handle all scaling infrastructure to meet demand, on demand, which is a appealing characteristic for the technology. Developers can then focus on development which result in less time to deploy business or research use cases and reduces the chances of costly mistakes [16]. Popular providers who are offering these services are AWS with Lambda, Azure with Azure Functions and Cloudflare with Cloudflare Workers.

It's been shown that serverless functions are scalable and work best when a workload is stateless and embarrassingly parallel, but when they are not it can become problematic [40]. One of the issues is that each function does not have unique addresses so messaging between functions is impossible. Another issue is maintaining state, as functions only live for a short amount of time (15 minutes) [32] which mean that state must be kept inside of a storage system and shipped to the function on start, or on request. Finally, it's hard to measure serverless performance as functions are random assigned to random machines. Luckily, there have been recent interest in developing serverless systems which address some of these issues stated.

Cloudburst [42] was one of these; it is a research project that looks into providing statefulness to serverless functions. They we're able to develop it by creating a fast distributed key value store (Anna [44]) and a distributed compute layer which would cache keys and values for fast access to the values. Shredder [45] was a project that would ship Javascript code to data instead of serverless's data to code paradigm. However they did not have the time to develop it into a distributed system. Cruical [20] provides a thread abstraction which maps threads to invocations of serverless functions. To provide state to functions, a distributed shared object layer was built on-top of a low-latency in-memory data store. SAND [24] took a note out of the actor model playbook and push all network messages to a central bus, then each node would pull down and keep a local cached bus of all events in the system. Functions would watch the bus and begin invocation if they saw a message that was meant for them. FAASM [41] used webassembly for developers to program their functions and allowed functions to share memory locally through a buffer. Data could also be saved to a local and distributed key value store. Finally, Apache Flink [2] is a project which runs stateful serverless functions in well known locations that periodically checkpoint their state to a RocksDB key-value store.

Major cloud providers AWS [5] and Azure [8] have also built on top of their own serverless platforms to offer more statefulness characteristics in some of their new products. This can be seen in both AWS Step Functions and Azure Durable Functions [7, 9]. These services focus on the ability to chain multiple serverless functions together to create a workflow where each function can share a context of the workflow. This is especially true for Azure Durable Function which we will give a deeper review in our literature review.

In our research, we found that many serverless solutions exist and range widely from production products to research based applications. During our time researching these technologies, we found most, if not all, were hard to run on our own hardware. Most research projects had a lack of documentation or had been left alone for a year or more which made rebuilding them difficult. Although serverless is promising, it seems as though the only supported projects are from major cloud providers. However, it's

at this point we begun to realize that the actor model and virtual actor frameworks share many characteristics solve some of the issues which serverless is now facing.

The actor model [33] is a conceptual model for concurrent computation which originated in 1973. An actor is a unit of computation which can only do 3 things: 1) create a finite amount of actors, 2) send a finite amount of messages, 3) determine how to processes when a message is received. Each actor contains it's own private state and can only interact with other actors through message passing. Because each actor own it's own state, there is no need for lock-based synchronization. When actors send messages, they are put into the receiving actors mailbox (or queue). Once there, the actor can processes one message at a time in a FIFO (first-in-first-out) queue. All messages contain the address of the sender, so actors are able to reply to messages they have received. The actor model was popularized by telecom companies using Erlang [18].

Although actor languages and libraries such as Erlang were able to simplify distributed system programming, it still burden the developer with many complexities because of it's low level of abstraction. Developers would need to solve key challenges in life cycle management of actors, distributed race conditions, handling failures and recovery of actors, placing actors and overall managing distributed resources. With these challenges led to the development of Microsoft Orleans [21] where the framework would try and manage all of those complexities so the developer would not need to be a distributed systems expert to develop applications in it. Around this time, most of the serverless characteristics except on demand scale could be found in Orleans.

Orleans provides a paradigm they named the Virtual Actor Model which raises the level of actor abstraction. In Orleans, actors are virtual entities that always exist in the virtual environment. Thus, actors created by Orleans can not be created or destroyed. If a message is sent to an actor that does not have an in-memory instance of it, Orleans will create one on an available server that knows the actor type. If an actor has not been used for some time, the runtime will reclaim the resources of the actor so that they could be used by something else. Actors running in Orleans don't fail as once the server running the actor is known to be dead, the actor is re-

instantiated on another which eliminates the need for a supervisor actor to re-created failed actors. Finally, if an actor is marked as stateless, Orleans will run just like a serverless function where multiple types of the same actor will be created and be given access to pull from the same mailbox thus allowing hot stateless actors scale out.

Another Virtual Actor Model framework which came out around 2016 was called Proto.Actor [35]. It released and marketed itself as the "next generation actor model framework". It was heavily inspired by Orleans. Proto.Actor system was created to use existing technologies and build off of them to provide it with the same characteristics as stated from Orleans above. On top of this, it allows for outside systems to communicate with an actor directly without the need to go through a frontend into the system. The project was spawned from the difficulties of using Akka.NET which required users to create custom core components such as Thread pools, network layers, serialization and more. In contrast, Proto.Actor focus is on concurrency and distributed programming. Protoactor adopts the Orleans model for fault tolerance. It uses industry standard technology gRPC for communicating between systems. When deployed as a cluster, it must be setup with with a clustering technology such as Kubernetes or Consul. Finally, the framework has been created for 3 different languages: C#, Kotlin and Golang.

Overall, in our research, we found that a stateful distributed framework should contain the following characteristics: Scalability, Portability, Statefulness, Coordination and Messaging, Predicatable Performance and short initialization times. Of the 6 characteristics, we've found that serverless has 1 (scalability), stateful serverless have 2 (scalability, statefulness) and actor frameworks have all but scalability and portability. Luckily, other technologies exist that can give our software these characteristics.

Kubernetes [14] is a cluster orchestration system that allows developers to load multiple containers on to a cluster of nodes. It is an open-source project and with all major cloud platforms supporting managed versions of it but also creating a self hosted cluster is just as easy. By loading virtual actor frameworks into an orchestration

system such as Kubernetes, we can achieve portability between cloud providers and automatic node scaling when a cluster is under load. Therefore, using Kubernetes can provide similar scaling characteristics to virtual actor frameworks that serverless technology already benefit from.

Overall, from our research, we've found that even though novel techniques for providing state to serverless workloads are being created, it is still far away from being feasible. It would be better to use existing and proven technologies such as the actor model to develop distributed applications. This thesis will review the performance of popular distributed actor model services by recording their performance in several microbenchmarks in best case and worst case scenarios. Throughout this thesis, we will review past research in serverless and distributed actor model solutions to expose their similarities and differences. Finally we will conclude our research by reviewing which platform is the best from results of our experiments, and will present what could be done on each platform to improve the performance of stateful computing in distributed applications.

1.1 Thesis Statement

This thesis provides an objective analysis on the performance of research and open-source based virtual actor frameworks using well-defined performance microbenchmarks. We believe our measurements will show a how far virtual actor model frameworks have come and their performance in throughput and latency in different distributed and stateful microbenchmarks. We believe that virtual actor frameworks improve upon current stateless serverless frameworks that rely on distributed state such as key-value stores and databases to provide state between invocations.

1.2 Main Contributions

Through this research open-source and researched distributed/virtual actor model frameworks were selected to find the current state and performance of virtual actor

model solutions that developers have access to today. From our tests we found that Proto.Actor performed at least 2 times better then Orleans while also being more complex to learn. Tests for open-source and research projects we're performed by setting up a real Kubernetes cluster and running a variety of test on it ranging from distributed operations like sorting a list to latency between virtual actors. From the results of all the conducted tests, a discussion of future research opportunities is presented highlighting the development of virtual actor model frameworks.

The main contributions of this work are 1) Measuring latency of communicating with other actors in stateless and stateful workloads 2) Measuring performance in worst case and average case scenarios 3) Introduce Microbenchmarks which have been extended to work in distributed settings.

1.3 Outline

This thesis is organized in the following way; Chapter 2 presents the Literature Review where we present serverless, stateful serverless and distributed/virtual actor model frameworks; Chapter 3 presents the Methodology which includes our tests, tools, metrics and microbenchmarks we will use to measure performance of our chosen frameworks; Chapter 4 reports our findings from the methodology; Chapter 5 and 6 will conclude our thesis with a review of what works well and what we hope to see with new developments in the research area of serverless and distributed actor model computing.

Chapter 2

Literature Review

In this section we will review serverless, stateful serverless and actor model projects created by company's, researchers and open source developers. In the first section, we will give an overview of current production ready stateful serverless solutions. Following that, we will review stateful serverless solutions that have been developed by research and open-source communities. After that we will review distributed actor model frameworks and it will be followed up with a review of virtual actor model frameworks that can be used today. Finally, we will summarize our stance on why we believe the virtual actor model is the future for serverless.

2.1 Serverless Overview

Serverless computing has begun to gain popularity in recent years with multiple cloud providers offering support for their serverless services [6, 9, 12, 13] and other companies developing their own serverless platforms [43]. These innovations have given developers a powerful abstraction that allow them to upload code into the cloud which are than accessible by a handle such as a URL. If the handle is ever access or triggered, the cloud provider will run the code. This switches the responsibility of running the code's resources to the cloud provider instead of the developer.

Although stateless serverless has been used successfully with developers creating novel solutions to specific problems that could be solved using embarrassing parallel

tasks [28, 36], they have overlooked other important areas needed for other forms of computing. One of those areas that were not targeted by stateless serverless was the poor handling of distributed state generated by distributed programs [32]. However, for serverless solutions to keep state, developers relay on cloud specific distributed databases or file systems that make it extremely difficult to move software onto other providers [42].

With cloud developers not trying to innovate to develop a stateful serverless solution, other developers began working on the issue. In recent years there has been an explosion of new research and open-source software such as Kuberentes [14] that lowered the barrier of entry into building cloud native software. With more developers having access to powerful platforms for developing complex systems it has lowered the requirements needed to develop a cloud native application.

2.2 Stateless Serverless

In this section, we will review stateless serverless platforms that are currently developed by cloud companies and open source contributes. Many stateful serverless technologies are built on top of their stateless counterparts. All of the solutions that are presented in this section work well for stateless tasks.

This section will analyze current serverless solutions that are popular today such as AWS Lambda [6] and Azure functions [10]. Then, we will also touch on Cloudflare workers [43] and explain why their solution is different from other cloud providers. Finally, we will give an overview of OpenFaaS [27] which is an open-source FaaS framework.

2.2.1 AWS Lambda

Lambda [6] is AWS serverless offering and it provides users compute services without needing to provision or manage any of the servers. It supports running code using preconfigured runtimes or custom created environments using containers.

Running code on AWS Lambda can be done by uploading a zip file containing the code and dependencies of the application or by creating a container that is compatible with the Open Container Initiative (OCI). If the function is created using a container, then it must include the operating system and runtime the lambda would use to execute the function. However, building a container is not needed if the function is developed in Node.js, Python, Ruby, Java, Go or .Net as AWS Lambda already has preconfigured runtimes for those programming languages.

To invoke a function, it must be registered to an event to watch for. An event can be a HTTP request, a timer that triggers on a schedule or consuming an event on a queue. When the function has been invoked by an event, it is sent a JSON document containing the information that triggered it. Once triggered, the code is attached to an *execution environment* which is a secure and isolated runtime environment for the code to be executed inside. Right now, execution is done inside of AWS own firecracker container which has fast startup times [17]. The runtimes purpose is to manage the processes and resources that are required to run the function. This includes relaying invocation events, context information and responses between AWS Lambda and a function.

When a function is triggered multiple times in quick succession, it's possible that it could be busy already handling a previous event. When this happens, a new instance of the function is created to handle the new event. This is because function instances can only handle one event at a time. However, this concurrency comes at a cost therefore it is possible to limit the amount of concurrency a function can receive so it does not scale out of control.

2.2.2 Azure Functions

Azure Functions [10] is the serverless offering from Azure that have native support for C#, F#, Java, Javascript, PowerShell and Python. However, they also include a way to create a custom handler that enabled support for any programming language. A Custom handler is a lightweight web servers that runs the developers code and is forwarded events from Azure Functions. A visualization of this can be seen in

Figure 2-1. Azure Functions are deployed inside of a *Function app* which is a logical way of grouping one or more Azure Functions together. All functions included inside of a *Function app* share a pricing plan, deployment method and runtime (programming language) as well as are managed, deployed and scaled together. Sharing state between functions must be done using an external persistent storage.

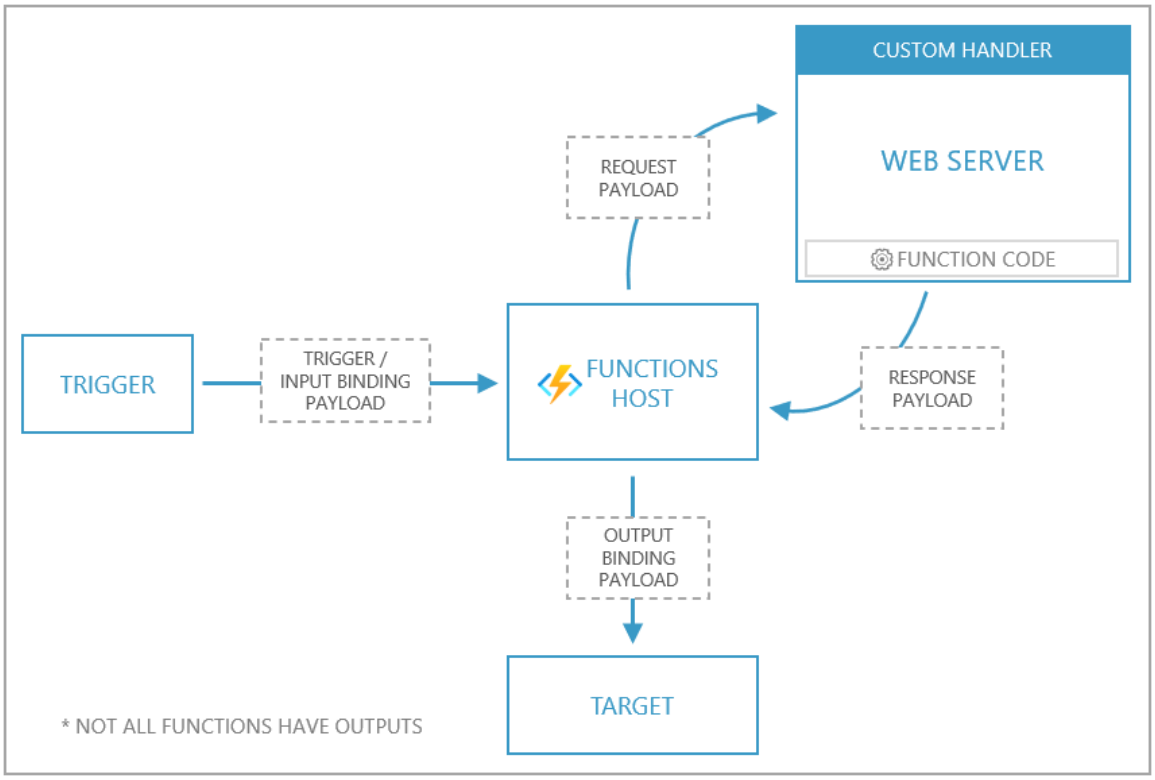

Figure 2-1: Custom handlers used in Azure Functions [10]

To deploy a *Function app*, each function must include a *functions.json* file with their deployment. This can be automatically generated depending on the language and tools that are being used for the project. Inside of each configuration file contains the functions trigger of which a function can only have 1 possible trigger. The configuration is mainly used to determine the types of events to monitor and how to pass data into and out of the function. When the functions have been deployed, they will attempt to handle as many event triggers as possible inside of a single-threaded runtime and will create more instances of the same function if it can not.

Azure Functions offers two relevant types of hosting plans on their platform: Consumption plan and Premium plan. The Consumption plan is the fully serverless hosting option for Azure Functions where you only pay when functions are running. After awhile, if events stop invoking the function, the function will be de-allocated. Multiple function apps can use the same consumption plan as long as they are located in the same region. All functions running on instances under the Consumption plan is limited to 1.5GB of memory and one core. The Premium plan brings a lot of benefits over the Consumption plan. Using the Premium plan, users can expect to avoid cold starts, guarantee functions have 60 minutes of execution time with the possibility of unlimited execution duration, better predictable pricing and virtual network connectivity. Other benefits include picking instances with one, two or four cores. The Premium plan is billed by memory allocated across all instances and the number of cores executing per second. At least one instance is always running a function to avoid cold start times with a max amount of instances always running set at 20.

2.2.3 Cloudflare Workers

Cloudflare Workers [43] is an interesting alternative to other cloud provided serverless solutions because they only support executing Javascript or Webassembly. Other languages are supported by using third party tools which compile or transpile the original language to Javascript or Webassembly. Workers use Googles V8 Javascript engine [15] as a runtime to support execution. By only supporting executing javascript workers, Cloudflare is able to host thousands of functions on the same machine.

The only triggered that is supported by Workers are requests sent over HTTPS. Each function when executed is inside of a sandbox meaning that a function will not know about other functions running on the same machine. Because Workers are not starting virtual machines they consume less memory and reduce cold start times with it taking on average 5 milliseconds to start a worker. However, cold start times are further reduced by starting to warm functions during the TLS negation phase so that when the HTTP body has been received by the worker, the function is already warm and ready to execute. This is presented inside of Figure 2-2.

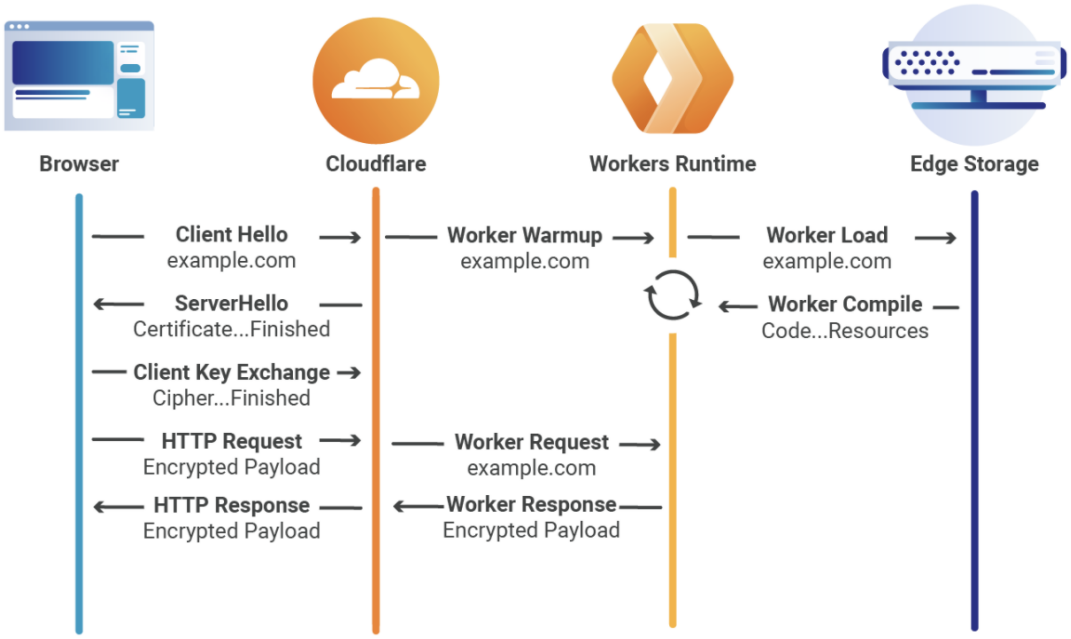

Figure 2-2: Cloudflare cold start architecture
[43]

2.2.4 OpenFaaS

OpenFaaS [27] is a open-source serverless framework that supports writing functions in Node.js, Python, Go, Bash and Java. Any language without support can be supported by developing a custom image using a container generated from a Dockerfile. OpenFaaS can be deployed onto different platforms such as OpenShift and Docker Swarm but the recommended one is to deploy it on is Kubernetes. However, if deploying to a low powered system such as a raspberry pi or a lower spec computer, than they provide *faasd* which is a static binary that works on linux systems.

OpenFaaS depends on a handful of cloud native which they call **PLONK stack**. This can be visualized in Figure 2-3. A PLONK stack includes Prometheus, Linkerd, OpenFaaS, NATS and Kubernetes. Prometheus is a open-source systems monitoring and altering tool which is used get metrics from nodes inside of a cluster. It's used inside of OpenFaaS to provide node autoscaling automatically. Linkerd is a service mesh for Kubernetes that makes it easy for applications running on the same cluster

to communicate with eachother. NATS is a messaging system that sends events to functions inside of OpenFaaS which come from the API gateway of the system. Finally, Kubernetes which is the platform of choice to run OpenFaaS on.

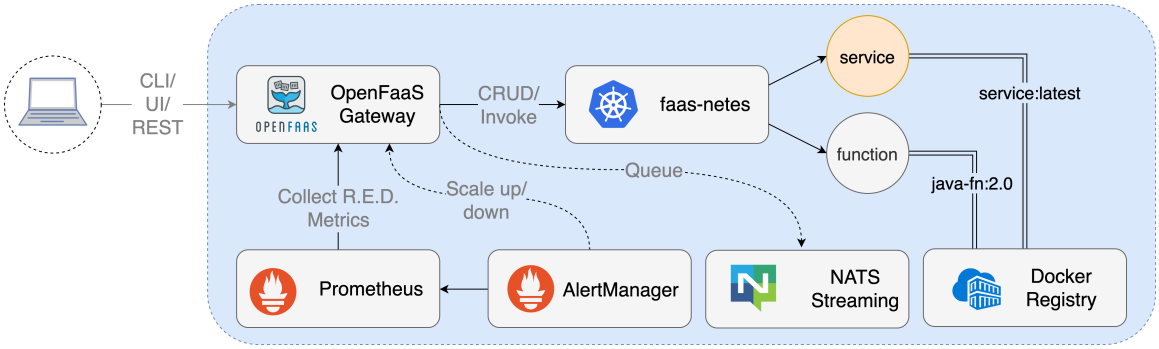

Figure 2-3: Architecture of OpenFaaS

2.3 Stateful Serverless Overview

Inside of this section, we will review serverless technologies that have been developed with state in mind as a requirement for their systems. Using these systems, developers can access state without needing access to a distributed database or block storage.

Overall, in the world of serverless there are three types of overarching categories of systems that can be used to run stateful serverless applications. The first category are Orchestration Serverless frameworks such as AWS Step functions [7] and Azure Durable functions [9]. The second category are big data frameworks which includes Hadoop [3], Apache Spark [4] and Apache Flink [2]. The last category includes researched serverless solutions. The research community has been putting a lot of work into coming up with effective serverless solutions, the most promising of which is Cloudburst [42] which comes from MIT. Other researched solutions include FAASM [41].

Current data-intensive serverless applications have address the statelessness problem a bit differently though with their own custom solutions to provide statefulness to it. Some of the solutions such as ExCamera [28] and Cirrus [23] tried to mitigate data movement cost by using long lived serverless VM. This introduced non-serverless

parts to the application. There has also been a focus on moving away from container based serverless functions. Terrarium and Cloudflare Workers [43] use googles V8 JavaScript engine to isolate applications from each other.

This review focuses on general purpose stateful serverless solutions. It will include an overview of different cloud platforms and frameworks with details on some of the technologies and projects mentioned above.

2.4 Serverless Orchestration

Serverless Orchestration was the first type of stateful serverless solution that provided developers a paradigm to access state inside of their code without having the need to access a distributed database.

General purpose stateful serverless solutions were first developed as serverless orchestration. They provide users with a way to develop a directed acyclic graph where each node is a serverless functions. The cloud provider, such as AWS or Azure [5, 8] are the main players and have developed promising solutions to these problems with their serverless orchestration technology.

Orchestration systems are generally programmed around an idea of state machines which use a shared context and transfer the output of one serverless function to another. Both AWS Step Functions and Azure Durable Functions have been built on top of their own cloud platforms using their own serverless offerings.

2.4.1 AWS Step Functions

AWS Step Functions are a stateful orchestration layer on top of AWS Lambda. They work as state machines where each Lambda function is a new state. States can perform work, make choices pass parameters and create more Lambda functions that execute in parallel. When combining multiple states together, it creates a workflow which are designed using a JSON file or a graphical user interface. An example of this can be viewed in Figure 2-4

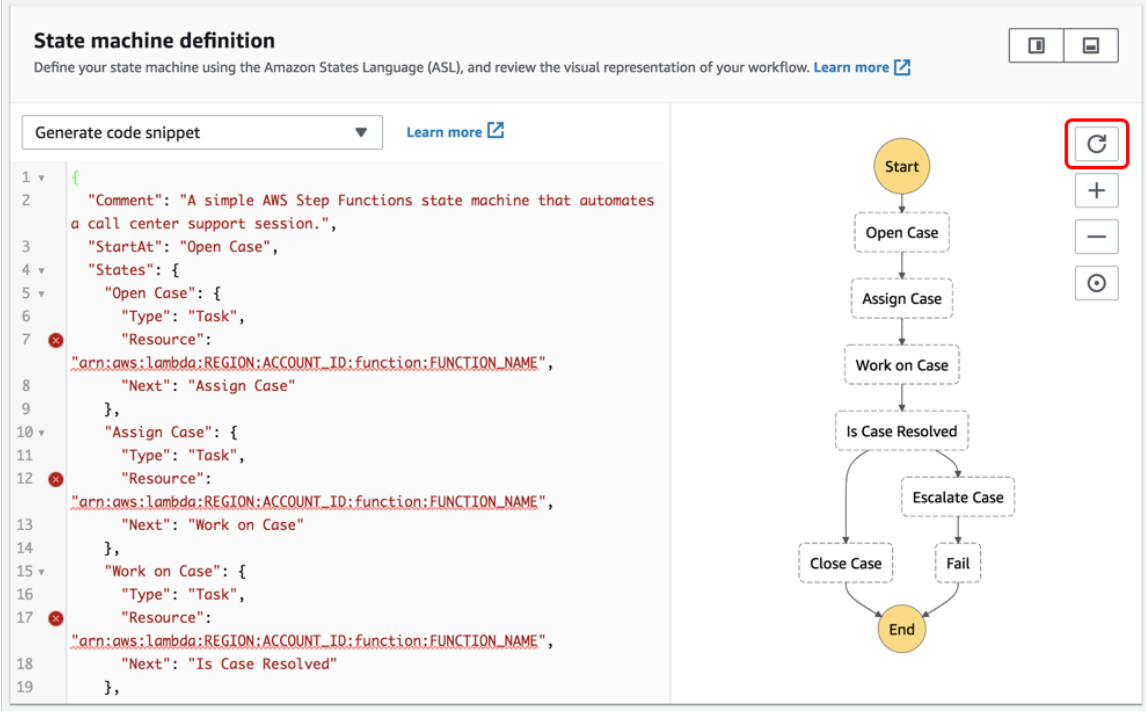

Figure 2-4: Example of coding states for an AWS Step Function

Workflows have two modes of execution which help with very different workloads. One of the workloads they have is a Standard workflow that is ideal for long running workflow. In this workflow states are only triggered once, unless otherwise stated. Workflows can run up to be 1 year long and are billed for the number of state transitions that are processed. The second workflow which is much more interesting for general purpose serverless computing is the Express Workflow which are ideal for high volume event processing workloads, transforming data, and saving it.

2.4.2 Azure Durable Functions

Azure Durable Functions are an extension of Azure functions. Durable functions are the orchestration system that handles state, checkpoints, and restarts for the user. They do this through an orchestration function which defines a workflow in code which call multiple other Azure functions that are meant to execute in the process of the workflow. An example of a Durable function is presented in Figure 2-5 where it chains together multiple Azure functions. Every time a function inside of the orchestration

function needs to wait or yield a response, the execution stops and Azure triggers the Lambda function to run. When it completes and returns the result, it is cached in the orchestration function. The orchestration function starts executing from the top again and continues until it completes or needs to run another serverless function that it does not have the cached result for.

```
[FunctionName("Chaining")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1", null);
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return  await context.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // Error handling or compensation goes here.
    }
}
```

Figure 2-5: Example of *chaining* Azure functions inside of a Azure Durable function using C#

Durable Functions overall have no time limit and can run forever which is better than AWS Step functions which only hold state for 1 year. A downside is that Durable functions only support a handful of languages because of the constraints of the platform's development. Currently supported languages are C#, JavaScript, Python, F#, and PowerShell. All Azure Functions and Durable Functions are open source.

Durable Functions can be defined in the current *function app* or in other projects. When executing an Azure Function from an Durable Function, the input will be pushed onto a queue and a virtual machine inside of Azure datacenters will pop the message from the queue and execute the message inside of a Azure function. When orchestrating Azure Functions, the Durable Function will be put to sleep which will prevent users from getting "double billed" for both the Azure function and Durable

function being executed at the same time. The queues and block storage needed to execute Durable Functions are billed to the users account.

2.5 Kubernetes

Kubernetes is a platform built by Google that was developed from years of experience running containers in datacenters [22]. It is a open-source container management orchestration platform for running many applications on a cluster of computers. Using containers, Kubernetes can deploy, manage and scale application.

Kubernetes needs many component installed on a cluster of machines to work. Every cluster is made up of at least one master node which handles running Kubernetes and one or more worker nodes that runs the user containers. At the core of the master node, there is an API server which developers can communicate with to issue commands to the cluster though HTTP or using tools list *Kubectl*. Data about the state of the cluster is saved to a key-value store running locally called etcd. Along side the API server runs the scheduler and controllers. The scheduler works to maintain the user defined state of the cluster while controllers are used to extend Kubernetes base functionality. The master node then can issue commands to the worker nodes through a daemon program called *Kubelet* running on all of the worker nodes. All nodes on the cluster must run a Container Runtime such as Docker so that they are able to execute container programs. The lowest level of abstraction in Kubernetes is a Pod. Clients of the system will normally access the system through worker nodes that are exposed through services. This is a general overview of Kubernetes and can be visualized in Figure 2-6.

Kubernetes provides concepts to abstract the way that developers can think about running workloads inside of a cluster of computers. One of those concepts is *Pods* which are one or more container that runs an application. If a pod were to run more than one container inside of a pod, the other containers added to it are considered to be running as a *sidecar*. All of the containers inside of a pod share resources and are co-located on the same machine although multiple pods can be running the same

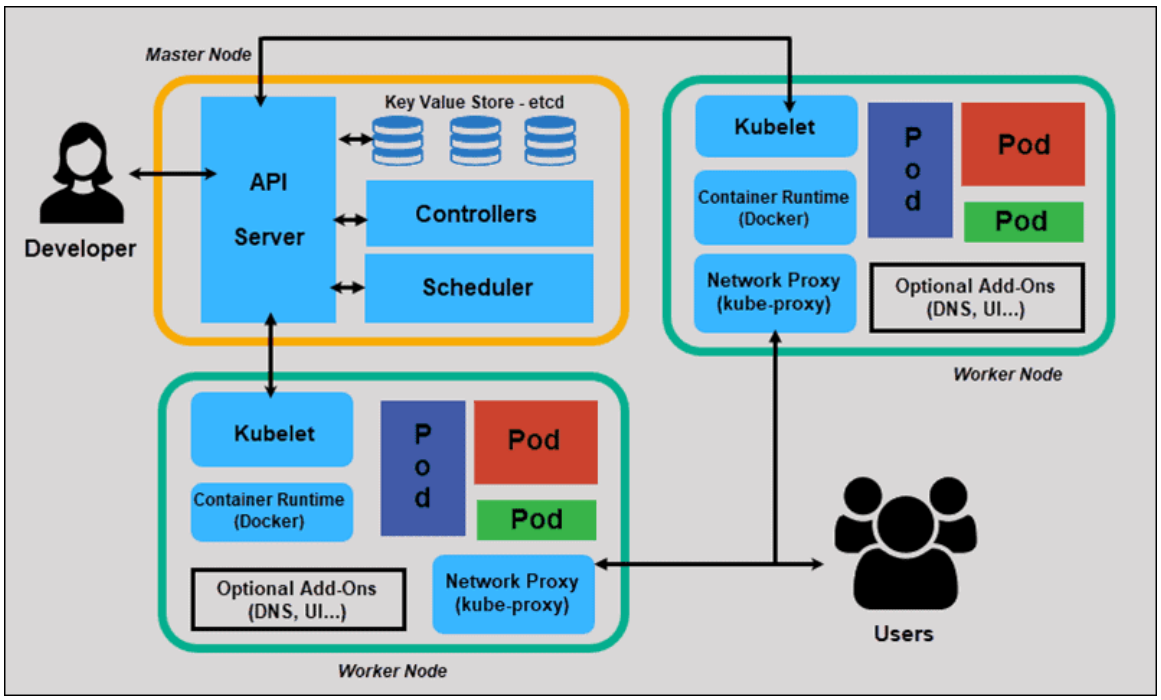

Figure 2-6: General visualization of a Kubernetes Cluster

application. Every pod in the cluster is given a unique IP address for use inside of the cluster. Pods can be accessed through *services* which abstracts the access to a collection of pods IP's through a single entry point. This entry point can be a public IP address along with an open port or a DNS name that resolves pointing to the public IP of the cluster.

Kubernetes is able to watch deployed Pods resource usages or custom metrics and provides developers tools to scale pods depending on the load of a Pod. Kubernetes can be configured to scale Pods virtually such as adding more CPU or RAM, or horizontally such as deploying more Pods to distribute the load.

2.6 Big Data Serverless

Big data has seen a lot of interest in the past several years as data has become more important since the resurgence of machine learning and statistics. Serverless has been playing a big roll in big data from the beginning with google first creating Hadoop which distributed data over many nodes to handle distributed calculations. Following

Hadoop was Spark and now there is Flink, which is the newest member to handle distributed computing on a cluster of computers using the serverless paradigm. There has also been research solutions such as FAASM [41] which is mixing webassembly runtimes and running operations on large datasets.

2.6.1 Apache Flink

Apache Flink is the most recent big data serverless application created to date. It integrates well with industry standard cluster technology such as Kubernetes but can also be ran as a standalone cluster. It was developed in Java and Scala and contains 4 components that it requires to run: a job manager, resource manager, task manager and dispatcher. Apache Flink works on bounded and unbounded streams of data from different data sources, however, normally data is supplied from Apache Kafka. The key feature of apache Flink is being able to operate on these streams of data in a stateful fashion without needing to worry that the state will be correct. User have reported being able to hit impressive scalability performance such as handling trillion of events a day, processing terabytes of state and running on thousands of cores.

Flink's focuses on streaming use cases such as event-driven applications, stream/-batch analytics and data pipelines. It handles all the data and statefulness of each function so that developers can focus on creating business logic. The reason why Apache Flink can operate at scale is that instead of running functions which request state, the state invocates the serverless function meaning once the function has started to execute it already has the input parameters and local state needed to process a request. This makes Apache Flink a great choice for companies that need to process massive amount of data and store large amounts of state.

2.6.2 FAASM

FAASM is a recent big data research project that takes a different approach. The project sets out to solve 2 issues: 1) inefficient state sharing between distributed functions and 2) reducing isolation overhead of functions.

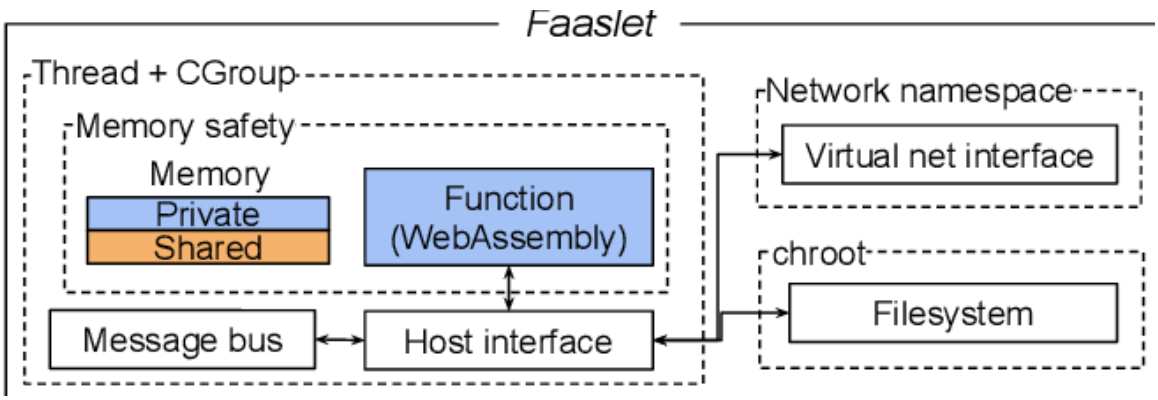

Figure 2-7: Faaslet abstraction with isolation

To solve these problems, FAASM distributed serverless runtime introduces *Faaslets* which are a lightweight isolation abstraction that have fast startup times and can be visualized in Figure 2-7. Faaslets have a 200KB memory footprint and start in less than 10ms. They achieve lightweight isolation by using webassembly and *software fault isolation* (SFI) which guarantee that functions only have access to their own memory. Similar groups of functions are also grouped together into Linux cgroups that have access to dedicated CPU thread and share a configured amount of CPU time. This allows the FAASM runtime to share local address spaces between functions so that shared memory regions can be shared.

Faaslets provide a two-tier shared state hierarchy which provides local in-memory state sharing and global distributed state sharing. Functions have access to both states through the Faaslet API provided by the FAASM runtime. When data is saved to the global tier, it must be assigned a key which is used when looking up the data.

2.7 Research

Thanks to invention of Kubernetes, we are seeing more researchers and companies creating their own autoscaling stateful serverless clusters that are cloud agnostic. In the following section we present CLoudBurst, the latest advancement in general purpose stateful serverless technology built by researchers.

2.7.1 CloudBurst

Cloudburst [42] is a research stateful serverless platform created by the RISELab at Berkeley. It archives statefulness by using the highly scalable key value store called *Anna* [44] as its storage layer which caches frequently used data on the same machines as the serverless functions. This helps lower the latency to access data between worker nodes.

To deal with consistency and fault-tolerance, Cloudburst intorduces their own system called *AFT* which is a layer that is between the serverless and storage layer. The *AFT* layer assures fault tolerance by enforcing that reading data only comes from data that has been confirmed to be committed.

Functions for Cloudburst are written in python and take objects or references to key value store keys. The values of the references to the key values store are retrieved at runtime and inserted into the program. Cloudburst API gives access to Anna for persistent storage and the ability to send a receive messages from other running functions. Cloudburst functions are executed inside long-lived python processes which connect python programs to Cloudburst.

2.8 Distributed Actor Model

In this section we will be focusing on distributed actor models that allow developers to create distributed systems for the actors to execute in. These Frameworks provide an abstract messaging layer to allow the developer to not need to understand how messages are being passed from one actor to another. However, the developer does need to code in a way which allows nodes to know about each other and connect to one another. Overall this section will review Erlang and C++ Actor Framework.

2.8.1 Erlang

Erlang is a declarative, concurrent, functional programming language that was created by telecom companies in the late 70's and released to the public in the 80's. Erlang

runs inside of a virtual machine and is used to build distributed, fault-tolerant, highly available systems that can have code hot swapped into the system.

Erlang programs are made up of a collection of lightweight processes that are ran as a single threaded task. Processes do not share memory with each other and are ran on a virtual machine that is able to handle executing many processes at once. Processes can communicate with each other using message passing. Messages that successfully reach their target process are placed inside of the target processes mailbox to be handled eventually. Programs are normally built out of hundreds of small Erlang processes. The idea of ErLang is to have every event being ingested into the system to relate back to a unique process.

In Erlang the idea of defensive programming where you try and handle ever error that a process might encounter is discouraged. Instead programs are built to fail. When a process fails, it terminates itself and sends a message to it's supervisor. An example of a simple process can be viewed in Code 2.1 where it attempts to process the factorial of a integer greater then or equal to zero. If for example, a negative value is passed, the program would fail and thus, terminate.

Listing 2.1: Example of factoral in ErLang

```
factorial(0) -> 1;
factorial(N) when N > 0 ->
  N * factorial(N-1).
```

When a process terminates, a message is sent to the process that spawned it which is named as a supervisor. Supervisors are able to monitor other processes and are able to capture them if they terminate. Based on the exit signal of the process, a supervisor can decide what to do. It can restart the process, or keep it terminated. Sometimes a supervisor decides that restarting a process won't solve a problem, if that is the case, it can destroy itself and all of it's child processes to send a message to it's supervisor. The top level supervisor can then decide what to do. If supervisors continue to not know how to solve the problem, they are terminated until the system stops or the issue is solved.

The language embeds distribution into the runtime and a cluster is characterized as a number of Erlang runtime systems communicating with each other. Once a cluster has been created, process can be ran on other nodes. If a process is started on another node, it must be declared so at runtime. Same with message passing, if a process wants to send a message to a process on another node, it must know the nodes name and the processes PID to successfully send the message.

2.8.2 C++ Actor Framework

C++ Actor Framework [25] (CAF) is a modern C++ library which gives a programmer the tool of creating actors inside of their programs. CAF was designed so that actors would implemented as light weight state machines that could be strongly or dynamically typed. Actors can be configured to run as event based actor or as a blocking actor. Event based actors allow for their execution to be interrupted after a handler has returned. Blocking actors on the other hand take control of the thread it is running on and does not allow any other actor to run on it until it has been destroyed.

Actors are defined by returning behaviors which are a collection of functions that have different types of arguments which are the message(s) they accept. When defining the handlers for an actor they are compared one-by-one as a list by the way they were defined in the source code. For actors that need the same arguments on multiple handlers, atoms can be used. Atoms are non-numerical constants introduced by the programming language Erlang. They are unambiguous, special purpose type and have no runtime overhead and in CAF you can use them to tag types (handlers).

Actors only execute when they are reacting to messages inside of their mailbox. In CAF, actors can only communicate between each other through messages which in turn normally will alter some of the actors internal state. For message to be sent, the data structure of a message must be serializable. If actors are statically typed, then the accepting handler will need to be programmed before the message can be sent as if it is not implemented at compile time, the program will fail to build. This problem is not present for dynamic actors as CAF checks to find if a message has a handler at

runtime. One feature of CAF is that message are able to implement priorities. The higher the priority, the soon it will get processed by the actor.

Mailboxes in CAF are implemented as double-ended queues. This design allows threads in CAF to attempt to steal work from other thread queues that maybe overloaded with messages. Threads with no work can attempt to steal work from the top of other queues. Overall, threads will go through 3 states of work stealing: 1) aggressively attempts to steal work 100 times with no sleeping in between, 2) moderately attempts 500 steals with 50 microsecond sleeps after every 2 requests, 3) relaxed runs forever and attempts to steal with with a 10 millisecond sleep every 2 requests.

One feature of CAF is the ability to group actors together. When groups are executing on the same system and immutable messages are received that the entire group needs to processes, each actor can copy the message by reference then by value. However, messages are copied if the actor hold a reference to the message and at least one argument of the message handler takes a mutable reference.

Actors are shared through pointers to the physical actor. To implement this, CAF created it's own smart points as they wanted to include more location data of the actor inside of the shared pointer and reduce the amount of indirection which happens in the standard library version of shared pointers. Actors are only destroyed if there are 0 strong references to it.

Middleman in CAF, are designated actors which handle communication with the operating system. They allow actors to seamlessly message other actors. In terms of networking CAF provides a broker that is an event based actor which is a middleman that multiplexes socket I/O.

2.9 Virtual Actor Model

New actor model systems build distribution by default into their frameworks. This allows developers to have no need to understand how the program is executing behind the scenes and instead focus on developing the logic for an actor. These frameworks guarantee that an actor will receive at most one message. They also provide the

concept of a virtual actor which Orleans has popularity named a *grain* which is an actor that always, virtually exists. In this section we review Orleans and Proto.Actor virtual actor model frameworks.

2.9.1 Orleans

Orleans is a framework that was developed by Microsoft research back in 2011 and was released as an open source project in 2015. It has been used in production with the most notable example is being used as the multiplayer backend for Halo 3. Orleans introduced and popularized the concept of a virtual grain, which is an actor that virtually lives forever in a system even if it physically does not exist. Grains must implement an interface which defines the actors behavior and must be declared and ran inside of a Silo, which is a name for a server that runs grains for the Orleans framework.

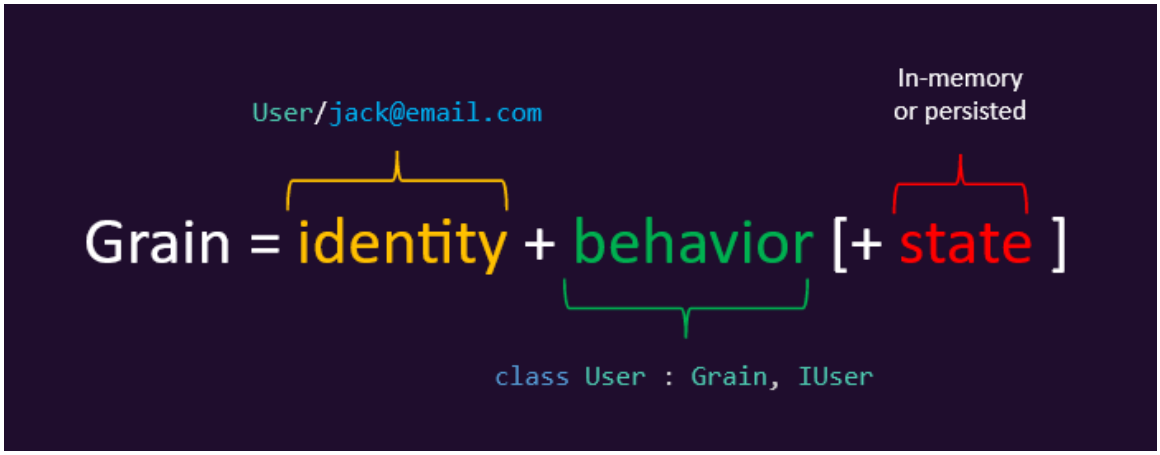

Figure 2-8: Example of what components make up a grain in Orleans

An visual overview of a grain can be seen in Figure 2-8. A grain is overall made up of an identity, a behavior (defined as an interface) and optionally the state kept inside of it. Grains are created at runtime when needed and contain all of their state within memory. If grains have been found to be idle for too long, they are deactivated and removed from the Silo to free up resources. Grains can communicate with each other through Orleans which handles delivery of a message. Because Orleans takes care of communication, the grain can be invoked if it does not exist and the message

can be delivered. Orleans handles the life cycle of the grain at runtime, which allows developers to program grains as if they were always in memory.

Because grains need to run in a distributed setting, they all need to be given primary keys which would act as the grains identity. These identities can be a Globally Unique Identifier (GUID), an long integer or a string. An example of this can be seen in the identity field in Figure 2-8. For grains which are to act as singletons, its good to set them up with empty GUID values. When creating a new grain, the placement of it is by default random, however this can be configured by the user.

Grains can be configured to grab state from persistent storage upon activation. In Orleans, these types of objects are called data objects. Grains can interact with these data object inside of it's state only if the data object is wrapped in a persistent state interface. Although reading the state is automatic on the initialization of the grain, it is up to the developer to decide when state needs to be persistented to storage. When saving the updated state, Orleans may or may not create a deep copy of the data object state which it will use to save the data to storage.

Silos are used to host grains. Multiple Silos can be ran together to form a cluster. Typically, Silos are ran as a cluster for scalability and fault-tolerance. When ran as a cluster, Silos work together to distribute work and detect and recover from failures. Third party systems can send messages to a cluster by creating a client which can connect to a cluster from outside of it. Clients can co-host a machine with a Silo however, it's normal for client and Silos to be ran separate machines. Silos can only create grains it understands which are the grains declared at build time, however, it can communicate with other Silos in a cluster to understand all of the available grains within the cluster.

Silos store grain identities within a data structure called the grain directory which is a global registry for registered grains. The purpose of this directory is to ensure that a message to a grain will be delivered and that no other instances of the grain are created. The Grain Directory is responsible for mapping a grain to it's correct Silo. By default, it is an eventually consistent in-memory distributed directory which is partitioned across a cluster of Silos.

When Silos are updated, this normally includes the grains inside of the Silos as well. For this reason, it is possible for two or more versions of a particular grain to exist within a cluster. By default, grains are required to be backwards compatible. This means that a V1 grain can message a V2 grain but a V2 grain may only message a V2 grain.

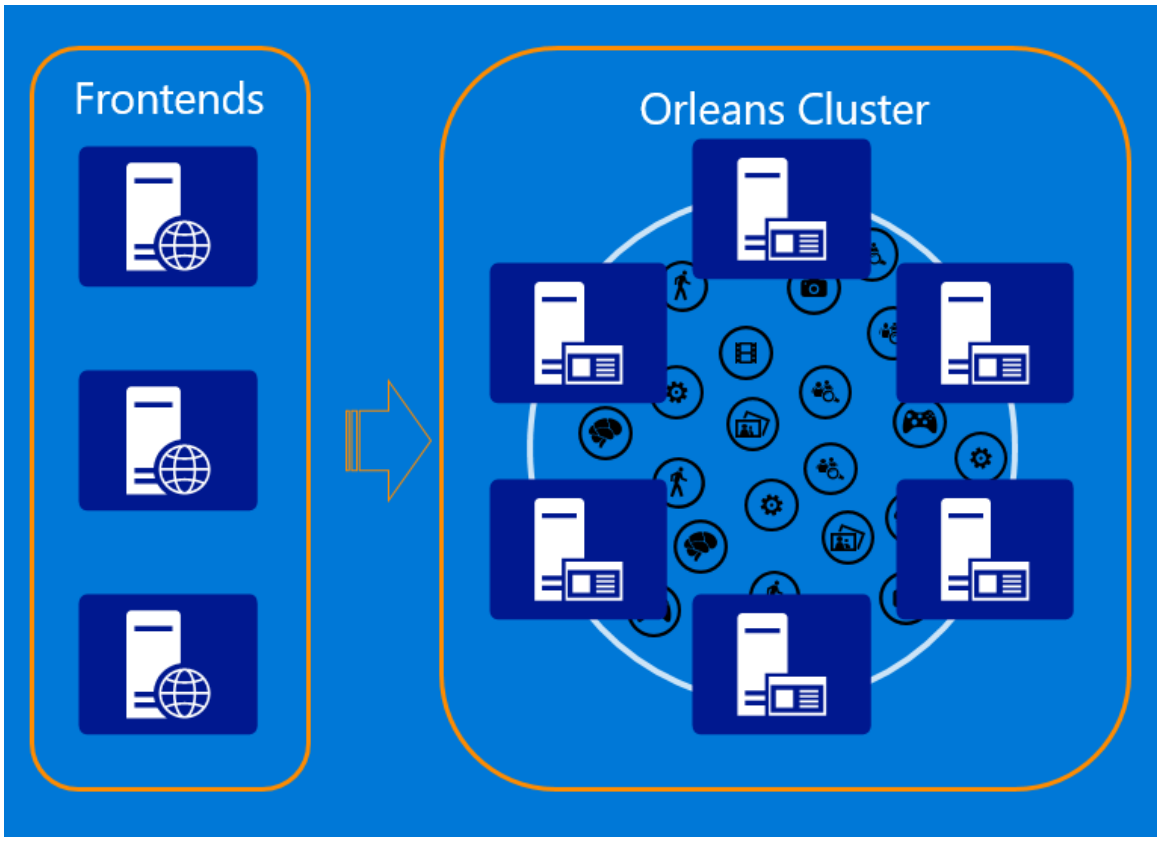

Figure 2-9: Frontend servers connecting to Orleans cluster

Accessing an Orleans cluster is normally done through frontends as Orleans is meant to be ran as a systems "Backend". Frontends can be any type of server which a user can interact with. These frontend servers can create clients into the Orleans cluster where they can then invoke requests. This is visualized in Figure 2-9

2.9.2 Proto.Actor

Proto.Actor is another virtual actor model framework which has been implemented in languages C#, Kotlin and Go. It has been heavily inspired by Orleans. It differs

from Orleans by being able to run on a single machine, on multi-machines with remote connections and like Orleans by running in a cluster. Proto.Actor was not developed by a company so the developers created it with the idea that to run it as a virtual actor framework, they would use popular, off the shelf library's to handle service discovery and message serialization. In this section, we review how Proto.Actor framework works from a single machine to a cluster of them.

Actors in Proto.Actor are containers for state, behavior, a mailbox, child actors and a supervisor Strategy. When an actor is created, it produces a Process ID (PID) which is the actors location in either a local or remote setting. Inside of the actor can be state which changes from incoming messages and the behavior needed to handle new messages that are received. Other actors can send an actor a message which will be enqueued to a clients mailbox by the time-order of a send operation. By default the mailbox is a simple FIFO stack however a mailbox can be configured with a priority queue. Actor include another mailbox which is only used for system messages which is only used by the actors context. A requirement for actors is being able to handle every message in it's mailbox as Proto.Actor doesn't require actors to implement every messaged received.

Actors can be in different state through it's life of the application. The states an actor can be in are the following: Started, Alive, Stopping, Restarting, and Stopped. Some of those state emit special events which the actor can hook onto. The Started event happens just before an actor processes it's first message and it recommended to load initialization state into the actor at this time. For Restarting and Stopping events, the actor should stop gracefully and persist any data that needs to be saved. Finally Stopped is triggered when the actor has been disconnected from the system and can no longer send or receive messages.

Messages in Proto.Actor are used to communicate with actors. Messages are immutable and inherently thread-safe as no thread is allowed to tamper with a messages contents. Messages passed within the system are eventually consistent, meaning that they will be processed asynchronously by the system when the actor gets to it. If messages fail to be revived by a mailbox, they become Dead Letters. When this event

happens, it is pushed into an EventStream where it can be handled if you have an implementation of the EventStream which handles Dead Letters.

Actors can send methods in two ways. The first is by using a Send call which is an non-blocking, fire-and-forget operation. It is preferred to execute in this way unless you require a request and response type of communication. If you do need to have a response with your request, an actor can execute a Request. The difference between Request and Send is that Request includes the senders PID while Send does not.

By default, messages that are passed in a remote setting use Protobuf [30]. For messages to be passed remotely however, there must be at least 2 separate system running Proto.Actor. Those system can be connected in different ways. The first is in a remote setting where clients connect to each others as peers. The other way is through clustering where systems learn about each other using.

Protocol Buffers or Protobuf for short, is an interface definition language that was developed by google which allows a user to use a domain specific language to create a contract between service. This can be paired up with gRPC [31] to handle remote connections with other hosts. gRPC can then communicate bi-bidirectionally between hosts using an HTTP/2 pipe. GRPC can then apply compression on the data sent over the network to optimize message size. Proto.Actor uses Protobuf contracts to generate a connection between the actor model system and the other system and gRPC to send and receive messages between two remote systems.

Up until now, our explanation of Proto.Actor was just a normal actor model system. It becomes a virtual actor framework once you configure it to be a cluster. Proto.Actor has been developed to allow for custom clustering systems to be used to manage a cluster member management such as Consuel, ETCD or Kubernetes. A requirement of using virtual actors in Proto.Actor is that messaging between actors must be performed using request-response methods. Grains can be auto generated for you by using Protobuf. After generation, the developer only needs to implement the base version of the actor which are all of the methods in the Protobuf service.

2.10 Why Actor Model is the future for Serverless

Serverless has been around for many years but it has been limited by the programming model that it has been supplied to it by cloud vendors. In recent times, we have been noticing an increase in the amount of third party solutions that try to solve one issue at a time. This thesis is meant to show that we are very close to obtaining stateful serverless solutions and through testing we hope to show some of the holes that still need to be fixed. Keeping data close to the function also positively correlates with the performance of a serverless platform [41, 42]. We are close to a solution and have promising technologies being developed to solve this issue right now.

2.11 Summary

This thesis will be focusing on the research and open-source solutions of virtual actor model as an alternative to hosted serverless offerings from large cloud providers. However, this thesis is also meant to show that stateful serverless has already been achieved through different technology and concurrency concepts. Our review purely focuses on general purpose computing of virtual actor model frameworks and conducting microbenchmarks to measure the best performing one.

Chapter 3

Methodology

In this chapter we will go over the metrics that will be used for testing distributed and virtual actor model frameworks. After which we will give an overview of the tools that were used to run the tests. Finally, we will present the tests that will be performed along with coded samples of the test.

3.1 Evaluation Criteria

To effectively test each distributed and virtual actor model framework, we must focus on measurable metrics. Other actor model frameworks have been tested before and have found great success in microbenchmarks [39, 34, 37]. Microbenchmarks are created to test a specific part of the working system. For our testing, we took inspiration from existing single node microbenchmarks and converted them to work in a distributed setting. The following metrics will be used for our testing to measure the performance of our microbenchmarks:

- ***Throughput***: One metric that will be collected is the throughput of the actor system. It is calculated by taking the number of requests the system completes and dividing it by the time it took to complete test. This metric will be used to understand how many messages can be processed by the actor system for the particular task. The number of messages sent is kept as a count inside of the

actor. When the test concludes, the count is collected and divided by the test time. This measurement will be used to measure performance per actor and the overall system. This is an important metric as it provides performance limits of the amount of messages a framework is able to process in a provided time frame such as a second.

- ***Latency***: Latency is used to measure request and response times between two actors. It is calculated by measuring how long it takes for a request to receive a response. This metric is used to determine the average time it takes to complete a request. It is also used to compare performance between virtual and distributed tests. This metric is important to us because it will show on average how long it takes to send data between actors.

- ***Actors***: Actors are the basic building blocks of an actor model system. All of our tests will include increasing the number of actors which the system will be required to handle as test difficulties increase. This metric is meant to measure the performance degradation of the system as more actors are added. It is an important metric because actors need to share CPU time and understanding the overhead of context switching could tell us how performant a given framework is as well as the best ratio of actors to a CPU core.

- ***Message Size***: Message size is a simple metric meant to be used to measure how the system preforms if we increase the amount of data we send between actors. By sending random data, we can get a better understanding how a workload on an framework may perform if it is required to be passing large pieces of data around between actors. It is an important metric which is used to give us a better understanding of the frameworks performance when there is a need to pass data between the actors.

3.2 Tools

To effectively test our microbenchmarks and give a fair playing field for all of our actor model frameworks, we are running our tests in a controlled environment. This means staying away from the cloud and using machines which we own that can give us consistent performance. Kubernetes is used to create a cluster for each framework. It provides a way to hook into its event system and collect information on pods that join the cluster. Using that information, the virtual actor frameworks can automatically connect to friendly nodes when they are initialized inside of Kubernetes. The cluster will be setup with Ansible [1] which makes redeploying the cluster repeatable and quick. For the rest of this section we will provide a brief overview of Kubernetes and Ansible, why we chose to use them and how they will help us create a fair test environment.

3.2.1 Kubernetes (K8s)

Kubernetes is a system that runs and coordinates containerized application on a cluster of computers. It takes the roll of managing the life-cycle of a deployed application and provide applications the ability to scale automatically and be highly available. In our tests, we have leveraged Kubernetes by creating a consistent framework for collecting our metrics and saving it to a file for later processing. On top of this, frameworks such as Orleans and Proto.Actor both make use of Kubernetes for it's clustering characteristics. It makes for deploying our applications easy which makes re-testing frameworks quick.

Overall, we found that using Kubernetes helps us create a fair test environment because it simplifies the setup process for the test. Once deployment files have been written and the test images deployed to a registry, tests can be repeated on any cluster without issues. Kubernetes acts as a flexible test environment that can run on any machine. This means that as long as the docker container can be built which contains the test, it can be performed on any set of machines running Kubernetes.

Kubernetes has begun to become a standardized inside of the development community and both projects have been developed with deployment to a Kubernetes cluster in mind [11, 42].

3.2.2 Ansible

To effectively test distributed actor model frameworks, we need the confidence that we can consistently setup our test environment in a consistent manner. To achieve this characteristic we used Ansible which is a tool that automates executing scripts on remote or local systems.

Ansible is a tool that is developed by red hat that helps automate managing systems from the command line and YAML scripts. Developers can use Ansible to develop playbooks which contain YAML markup which contain instructions to be executed on a remote computer(s). These instructions could install, start, stop, uninstall services or even run shell scripts. By developing YAML scripts for our environment, reduces the chances for changes to happen between tests. We used Ansible in this thesis to automate the deployment of our Kubernetes cluster.

3.3 Hardware Environment

All tests will be deployed onto a Kubernetes cluster of one Raspberry Pi 4B, 4 GB edition which will be the cluster master node and three Raspberry Pi 4B, 8 GB editions which are the worker nodes. Differences between worker and master nodes is that our workloads such as our tests will not be deployed on the master node. The master node only runs Kubernetes specific workloads such as deciding the placement of new pods and managing Kubernetes API server which developers have access to. The cluster will be initialized using the tool kubeadm which helps initialize Kubernetes clusters. To setup the Kubernetes environment on the nodes, we will use an Ansible script which will automate the cluster setup process. All tests will be performed with a clean Kubernetes cluster with each node inside of the cluster running a fresh install of Ubuntu 20.04 64-bit ARM edition. Tests for each language

are programmed into their own respective frameworks. After each test has been completed, metrics are collected from all of the actors and saved to a CSV file. All tests will collect information on latency between requests and responses, number of actors used, message sizes and throughput of the system. Overall, all tests will measure the total time it takes to accomplish certain tasks. All of the Raspberry Pi's have been equipped with SSD. This was done to increase reliability of the cluster as well as because of their performance increase over SD cards [29].

3.4 Tests

To collect our required data, we will be performing a collection of tests that will be executed with each framework (i.e., Orleans and Proto.Actor). Every test will be connected with one of the previously stated objective metrics in the previous section. It will be connected to show the performance of a particular feature that the framework provides.

Each test will be executed the same way. There will be dockerfile(s) that are built which contain code for a server, client or both. Then a YAML script needed for deployment of the container will be generated. Every server instance will be deployed using a Pod. Each test client will then get executed as a Job.

In this section, we will cover all of the tests that will be performed to collect data on the following defined metrics. The metrics are: 1) Throughput, 2) Latency, 3) Actors, and 4) Message size. Each test will encapsulate at least one metric that has been defined.

3.4.1 Distributed Ping Pong

To test distributed ping pong, we will initialize two nodes inside of the Kubernetnes cluster. One node will know of the ping actors while the other node will know of the pong actors. The test is simple, send a set of amount of pings from the ping actors to a pong actor. To start the tests, we create a client node which remotely creates a set amount of ping and pong actors. After they have been created and given

initialization data containing the message size in bytes and number of messages to send and receive to complete the test, the client tells each ping actor to start. An example of the actors used in this test can be seen in Listing 3.1. While the ping actors were running, they collect data on the latency of each ping pong message and provide a list of it to the client. From this test, we will be able to analyze the latency between two actor that are ran on separate nodes as well as the throughput of how many messages can be sent every second.

Listing 3.1: Example Ping Pong Microbenchmark

```
class Ping {
   Pong pong;
   int pings = 0;
   int repeats = 0;
   void Init(Pong pong, int repeats) {
      pong = pong;
      repeats = repeats;
   }
   void Run() {
      this.Ping(pong, new Message());
   }
   void Ping(Pong pong, Message msg) {
      if (ping < repeats) {
         ping++;
         sender.Pong(this, msg);
      }
   }
}
class Pong {
   pongs = 0;
   void Pong(Ping ping, Message msg) {
      pongs++;
      ping.Ping(msg);
```

```
    }
}
```

3.4.2 Virtual Ping Pong

Similar to the distributed ping pong test, however, this time both nodes will know of the ping and pong actors. The difference being that the framework will manage placing them in the cluster. We hope to use the results of this test and compare them to the results of distributed ping pong to compute the performance difference of the framework when it manages the placement strategy of the actors.

3.4.3 Distributed Divide and Conquer

To test distributed divide and conquer, 3 nodes will be deployed to the cluster. Each node will have a unique type of divide and conquer actor that relate to the type of node they are running on. An example of this would be a "Sort A" actor will run on the "A node" and so on. The client will initialize one actor on a node with a list of numbers that the actor will need to sort by dividing the list in two and sending it to two other actors which continue to reside the list until the size is less then or equal to two. Once the list has completed dividing, it will return the complete list of numbers have been sorted. An generalized example of this test can be seen in the Listing 3.2. From this test, we are measuring the time it takes a framework to create actors that will work together to complete a task.

Listing 3.2: Example of a Divide and Conquor Microbenchmark

```
class Sort {
    Task<List<int>> Divide(msg: List<int>) {
        if (msg.Count > 2) {
            var mid = msg.Count / 2;
            Sort right = Context.GetSortGrain($"{Guid.NewGuid()}");
            Sort left = Context.GetSortGrain($"{Guid.NewGuid()}");
```

```
            var r = await right.Divide(msg[..mid]);
            var l = await left.Divide(msg[mid..]);
            var sorted = r.Concat(l).Sort();
            return sorted;
        } else {
            return msg.sort()
        }
    }
}
```

3.4.4 Virtual Divide and Conquer

Similar to the distributed divide and conquer test, expect each node will know of every type of actor that can be created inside of the system. Every time an actor creates another actor, it will be the frameworks responsibility to assign it a location inside of the cluster on one of the nodes. From this test, we will measure how long it takes to create many new actors and divide the list of numbers. Using the results from this test, we will also analyze the difference in speed from our distributed test compared to this test.

3.4.5 Distributed Big

For testing mailbox contention, we can use the microbenchmark Big [19]. The test will be ran on 3 nodes that will be deployed to the cluster. Similar to distributed divide and conquer, there will be 3 nodes with each node having the same actor but with a different tag so that they are unique. A client will create and initialize N actors across all 3 nodes. During the initialization, an actor is told about all existing actors in the system except for their self. When the test is ran, each actor sends a message to all of the other actors in a many-to-many communication. After the first round of messages have been sent, actors wait for a response and continue to ping and pong

eachother until the test has completed. An example of this test can be seen in the Listing 3.3. From this test, we are measuring the increased latency many-to-many messages caused the test. We are also measuring the throughput of the system when overloaded with internal messaging.

Listing 3.3: Example of a Big actor for the Big Microbenchmark

```
class Big {
   List<BigClient> clients;
   Message msg;
   int ping = 0;
   int pong = 0;
   int repeats = 0

   Task Init(List<BigClient> clients, int repeats) {
      repeats = repeats;
      clients = clients;
   }

   Task Run() {
      foreach (var client in clients) {
         client.Ping(this, msg);
      }
   }

   Task Ping(Big sender, Message msg) {
      if (ping < repeats) {
         ping++;
         sender.Pong(this, msg);
      }
   }

   Task Pong(Big sender, Message msg) {
```

```
      pong++;
      sender.Ping(this, msg)
   }
}
```

3.4.6 Virtual Big

Similar to distributed big, this test will send many-to-many messages across a collection of actors. The big difference between the tests is that all nodes in the cluster will know of the different actors in the system, so new actors can be spawned on any machine. The results of this test will be compared to it's counter part distributed test and the difference will be measured of how much of a performance increase there is when communication is not forced over the network.

3.5 Analysis

The analysis will involve reviewing the results of the data collected by measuring the total throughput of a variety of tests, and mean, median and distribution of latency. The data will be collected from completing the following tests: 1) Distributed Ping Pong, 2) Virtual Ping Pong, 3) Distributed Divide and Conqour, 4) Virtual Divide and Conqour, 5) Distributed Big, and 6) Virtual Big.

Chapter 4

Findings (Analysis and Evaluation)

From our tests, we have collected data which we will present now. This data was collected through running tests found in the methodology section. In this section we will review each tests findings, present graphs for each test, and explain the results we've received.

4.1 Test Settings

All tests were completed on a Kubernetes cluster. For each test, we repeated the following steps: 1) Create the servers that would run the applications (i.e. Orleans and Proto.Actor), 2) execute a client that would connect to the servers and start the test, 3) Collect and save metrics once the test completed, 4) Destroy the test environment (client and servers). All servers were configured with 3 CPU cores and 6 Gigabytes of memory. All servers were deployed as Pods while all clients were deployed as Jobs.

4.2 Ping Pong

In the ping pong test we were able to collect a variety of relevant metrics. First we were able to determine throughput of the frameworks from a variety of settings by increasing the actor count and data size of a message that was sent between pings

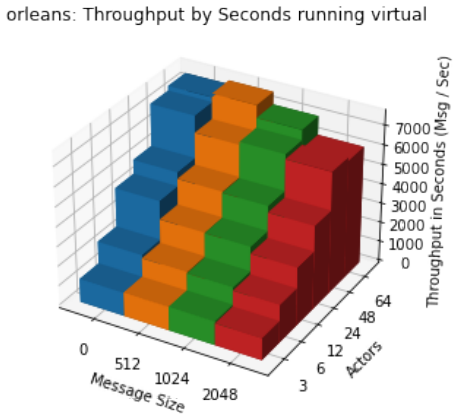

Figure 4-1: Throughput of distributed actors in Orleans

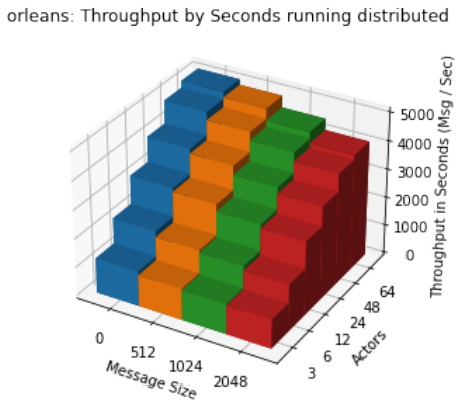

Figure 4-2: Throughput of virtual actors in Orleans

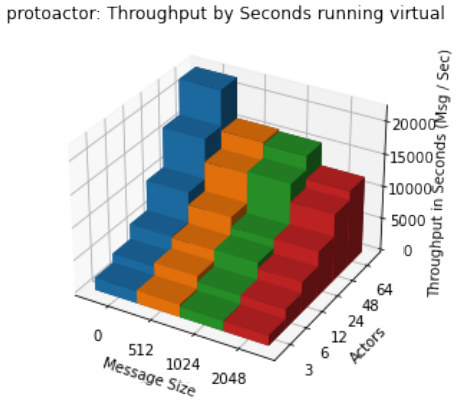

Figure 4-3: Throughput of virtual actors in Proto.Actor

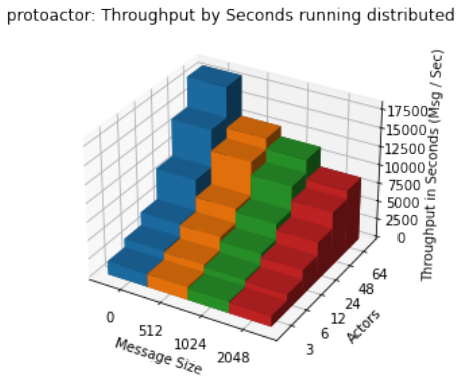

Figure 4-4: Throughput of distributed actors in Proto.Actor

and pongs. In all of the tests, we sent 7500 ping and 7500 pong messages. This test was repeated with 3, 6, 12, 24, 48 and 64 actors along with 0, 512, 1024, 2048 byte message sizes. The results can be visualized in Figures 4-2, 4-1, 4-3 and 4-4. Orleans and Proto.Actor both had their performance improve when ran in using a virtual setting by spreading out the load. Overall, Proto.Actor was able to achieve between 1.7-2.5x more throughput then Orleans.

Switching from distributed to virtual actors in Orleans saw throughput increase on average of 1.3x. As actor counts went up, we saw throughput increase to a high of 1.5x times when using 64 actors. For Proto.Actor, there was an average increase of 1.25. However, it differed from Orleans by most actor achieving a 1.25 increase when

using virtual actors. Figure 4-5 shows the performance difference between distributed and virtual actor throughput's as a percentage.

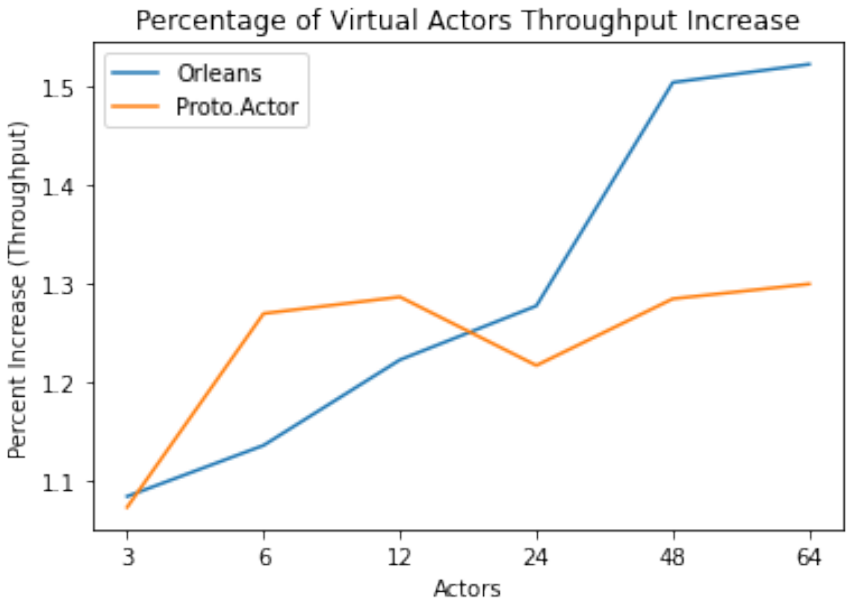

Figure 4-5: Virtual Actor throughput increase as a percentage compared to Distributed Actors

Along with the throughput, we were also able to calculate average latency times by message size which is visualized in Figure 4-6 and 4-7. When virtual actors are deployed, common latency times between messages are between 1 and 9 seconds for Orleans and 1 and 4.5 seconds for Proto.Actor. Distributed actors stay consistent with values virtual actors with latency times doubling in value. Virtual latencies do not have a pattern with most message sizes taking about the same time. Compare this to distributed latencies and we see that communicating over the network scales latencies linearly as the size of the message increases. Outliers can be viewed by comparing the max and 99.9% quartile latencies as seen in Table 4.1. The table also shows that distributed actors scale normally while virtual actors may have spikes in their latencies. It is notable that Proto.Actor performs better at 99.9% quartile consistently in virtual and distributed tests.

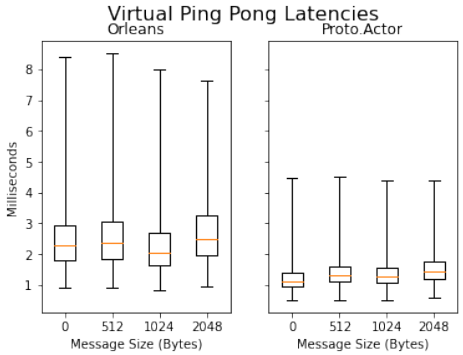

Figure 4-6: Latency of ping and pong for virtual actors

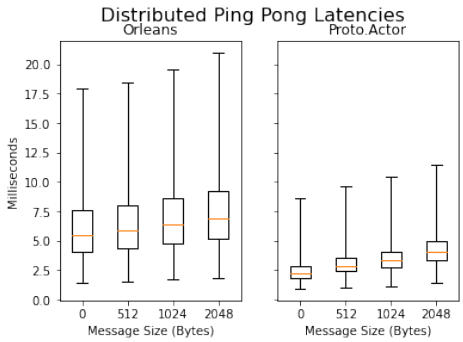

Figure 4-7: Latency of ping and pong for distributed actors

Table 4.1: Ping Pong Worst Case Latencies In Milliseconds

Framework	Setting	Min	Mean	Quartile(95)	Quartile(99)	Quartile(99.9)	Max	Max STD
orleans	Distributed	1.606851	6.949848	15.760030	19.219958	32.918106	86.992790	4.459547
protoactor	Distributed	1.089667	3.484110	7.619781	10.021473	22.976022	61.356861	10.311075
orleans	Virtual	0.896293	2.849023	5.631562	8.126904	39.277333	800.322975	195.259017
protoactor	Virtual	0.534368	1.461046	3.253128	4.446191	11.939781	73.485777	5.630817

4.3 Sort

For this test, we compared the performance of Orleans and Proto.Actor in sorting a list of random integers using actors. This test primary tests the cost of spawning new virtual actors (grains) inside of the cluster. This test was performed with lists sizes of 1000, 2000, 4000, 8000, 12000.

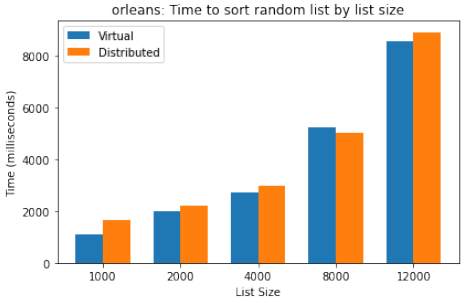

Figure 4-8: Performance of virtual and distributed compared actors sorting list of integers using Orleans

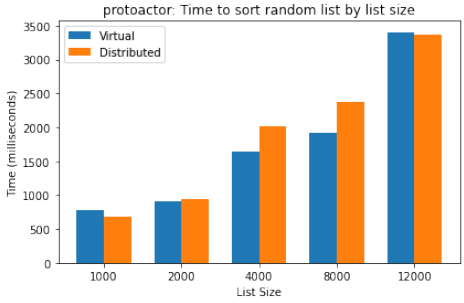

Figure 4-9: Performance of virtual and distributed compared actors sorting list of integers using Proto.Actor

Figure 4-8 shows the difference in performance for orleans for distributed and virtual workloads. The results show that on average virtual grains performance better

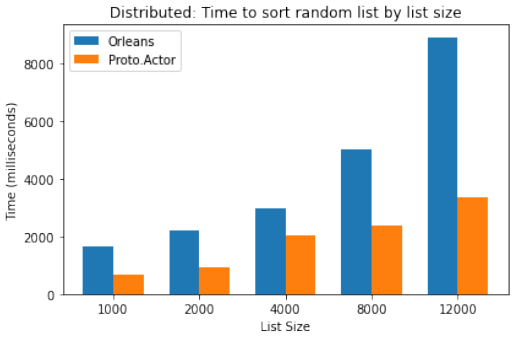

Figure 4-10: Time needed to sort random list of numbers using distributed actors

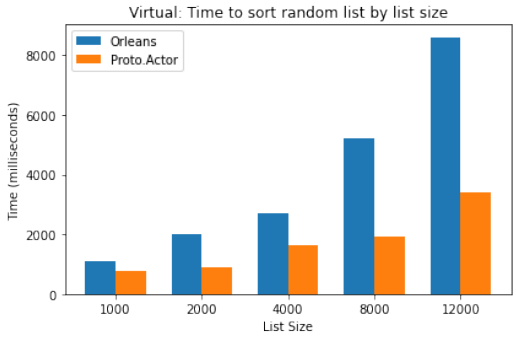

Figure 4-11: Time needed to sort random list of numbers using virtual actors

then their distributed counter parts. This is probably because of Orleans default placement algorithm which is random, thus grains are still placed all over the cluster. For that reason we see when sorting a list of 8000 items, virtual sorting was slower. From these results we can conclude that grain creation for Orleans has consistent performance in a virtual and distributed settings.

Figure 4-9 is showing the results for Proto.Actor. The performance difference between sorting a list with virtual and distributed is not consistent. However the results do conclude that most of the time, sorting a list with virtual actors is a equal, if not faster then when sorting with only distributed actors.

Performance results between the two frameworks can be viewed through Figure 4-10 and Figure 4-11. In both graphs, Orleans takes considerably more time to complete the sort test when compared to Proto.Actor. On average Orleans requires more then 2 times more time to complete a sort. Possible reasons for this is that Orleans placement strategy is 2 times more expensive then Protoactors thus it takes longer to complete the test.

4.4 Big

In this test, we compare the performance of mailbox contention by having every grain messaging each other in a many-to-many style. In all of the tests for Big, we sent 5000 ping and 5000 pong messages. This test was repeated with 3, 6, 12, 24 actors

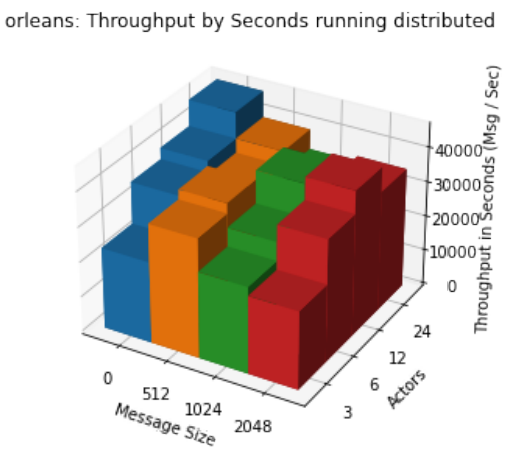

Figure 4-12: Orleans many-to-many messaging using distributed actors

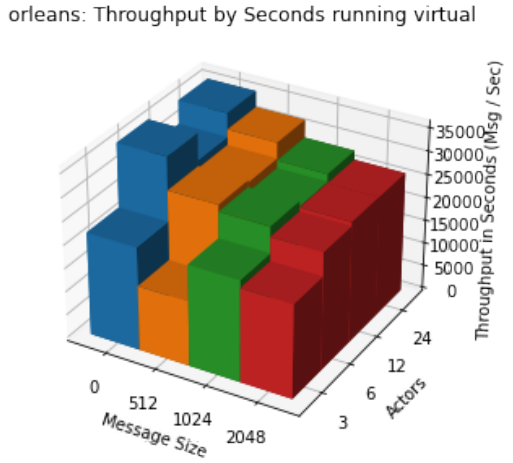

Figure 4-13: Orleans many-to-many messaging using virtual actors

along with 0, 512, 1024, 2048 byte message sizes. We attempted to also include 48 and 64 actors, however, both Orleans and Proto.Actor failed to complete the tests due to our systems not being able to handle the load. General results for throughput can be viewed in Figures 4-12 and 4-13 for Orleans and Figures 4-14 and 4-15 for Proto.Actor.

Figures 4-12 and 4-13 reflect these results. Overall, distributed grains were able to preform better then their virtual counterparts. On average virtual grains topped out around 30,000 messages a second while surprisingly distributed grains was able to achieve an average of around 37,500 message a second. We believe this is because the grains are evenly distributed across all nodes while in a virtual environment grains are placed randomly.

Proto.Actor out performs Orleans by almost 11 times over Orleans performance. The highest throughput that was achieved using virtual actors was with 24 actors and a message size of zero. Overall it was able to score 540,000 messages a second. Distributed performance was close with 510,000 messages a second with 24 actors and a message size of 2048. Performance growth for virtual actors is linear while for distributed actors it may have performance regressions. These results can be visualized in Figure 4-14 and 4-15. To show this performance difference clearly, we can compare the average overall performance by actor group. As seen in Figures 4-16

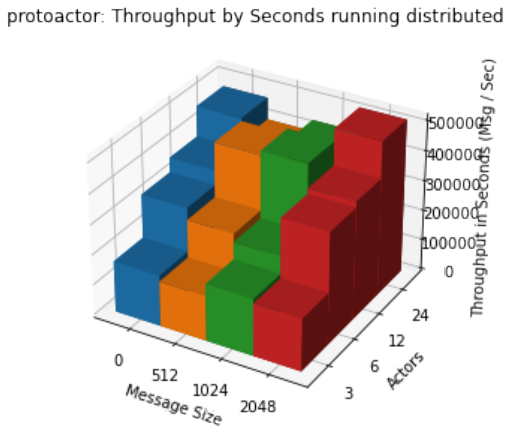

Figure 4-14: Proto.Actor many-to-many messaging using distributed actors

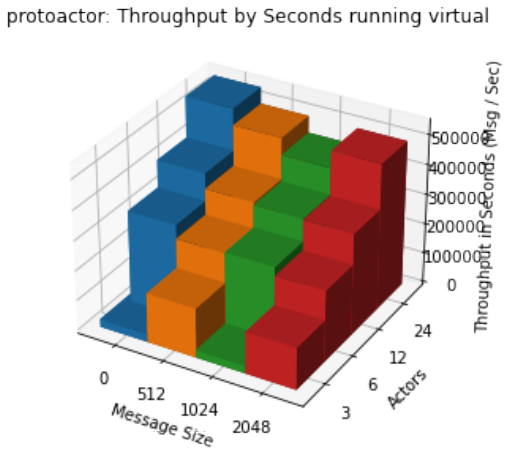

Figure 4-15: Proto.Actor many-to-many messaging using virtual actors

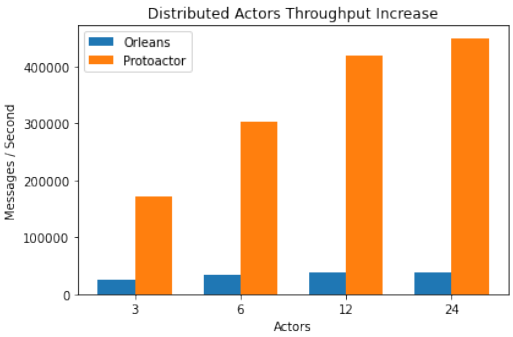

Figure 4-16: Performance different in Big test using distributed actors

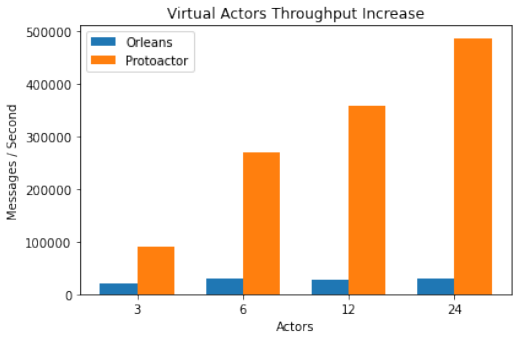

Figure 4-17: Performance different in Big test using virtual actors

and 4-17 where for both distributed and virtual workloads, Orleans can not achieve any where near the performance of Proto.Actor.

4.5 Discussion

In this paper, we set out to find the current state of virtual actor frameworks that are available today. Taking our two chosen frameworks: Orleans and Proto.Actor, we reviewed, tested and collected metrics that indicate Proto.Actor is faster then Orleans.

From our review, we found that although both frameworks accomplish similar tasks, they were built with different needs in mind. Orleans was developed with the

ideology of an actor always existing in a distributed system thus using the name virtual actor or grain. They built on top of this system to allow for normal use of the actor model in a automated distributed setting. Compare this to Proto.Actor which was developed with the idea that a developer should choose when to use a virtual and normal actors.

We also found in our review that when pairing virtual actor framework with a cluster technology such as Kubernetes, we can create systems that are alike to serverless services. Although running our own servers is different then using serverless, we gain better fine grain control over how our actors can hold state and when to save it. We can also develop in such a way as one actor holds the source-of-truth for a users state thus making the overall system easier to understand as an actor lives anywhere and can be quickly located and forwarded messages.

Through our tests, we found that overall, Orleans was easier to work with when developing our tests. We did run into trouble however when running our benchmarks. This is because Orleans scans the binaries included in the container on start up and loads any libraries that are meant to be used as grains in the system. This made running tests with Orleans harder as each actor now needed to be built inside of a custom image with a unique grain inside of it. This increased build and development times.

Proto.Actor had different difficulties. One of the larger problems we had was the difficulty of troubleshooting problems without contacting someone from the project. Documentation is lacking however it is an open-source project so it can be understandable. Currently they have bug bounty programs posted on Github to improve documentation for some monitory value. Because of that lack of documentation, we ended up developing Proto.Actors benchmarks twice. Once using auto generating code that requires virtual actors follow a request- response style of communication. This resulted in some of our tests dead locking or taking longer then normal compared to Orleans. With Proto.Actor, there are multiple ways to do the same thing which can make it harder for new users of their framework.

Overall, This thesis mainly focused on the performance of virtual actors in both frameworks. We found that between point-to-point communication, Proto.Actor is up to 2 times faster then Orleans and provides linear performance growth as the load on the system increases. Proto.Actor is also faster in spawning new actors with the framework being able to sort a list using actors more then 2 times faster then Orleans. Finally, our results show that when communicating many-to-many, Proto.Actor is over 10 times faster then Orleans. From our results, we can confidently say that Proto.Actor is more performant then Orleans.

From our findings, we believe that future virtual actor model frameworks should include ways for generating code that support send-and-receive and one-shot messaging. Systems should also have the ability to create virtual and non-virtual actors as seen in Proto.Actor. For more adoption, they should include a way of allowing messages to be sent to actors directly without a frontend. Finally, included in each framework should be a way for automatically adding new grains without shutting down or restarting a server as seen with serverless technologies.

Chapter 5

Conclusion

In this section, we conclude the thesis. First we present our summary where we quickly review results. Following that are our limitations where we talk about some of the difficulties we had when developing our thesis. Finally, we present our future work and discuss what can be done to build on our work.

5.1 Summary

In this thesis, we set out to find the best performing virtual actor framework from the two frameworks we chose. Of those, we set out to analyze their performance by measuring latency when sending and receiving messages and throughput of the system when under load through microbenchmarks.

From our results we can see that overall, Proto.Actor allows for faster communication between actors and fast when creating new actors. Proto.Actor has always performed at least 2 times as well as Orleans most likely thanks to it's low latency in inter-actor communication. We believe this shows that using proven technology is better to use then creating custom components, especially for systems that require heavy use of serialization.

To summarize, we believe that both frameworks performed well and that both are good for different workloads in different settings. Orleans is great when used for a new project or as a service that will only run in the backend as a micro service. Orleans

is able to get started quickly and can is by default a distributed program. Orleans documentation is very good and the library has a large community around it adding nice features to the framework. Proto.Actor is a better choice for integrating into an existing system with it's reliance on protobuf and gRPC thus not requiring the need for a frontend into the system. The framework has been fully implemented in C#, Go and Kotlin which allows for more adoption if a certain language is a requirement. As we reviewed in our findings, Proto.Actor is also very fast. The learning curve is higher then Orleans as it allows the developer multiple ways to implement virtual and normal actors. On top of that, Proto.Actor also currently lacks enough documentation however the developers are easily accessible through their communication channels.

5.2 Limitations

During the creation of this thesis, there were many issues we encountered. The first was in our original idea of testing stateful serverless frameworks from other research projects. When trying to build the software needed for these research projects, we found that was difficult as most research projects have been left for at least a year if not more and most are barely held together. After spending over 3 weeks trying to build some projects, we were led to other solutions which brought us to virtual actor model. Virtual actors had much stronger open-source projects that meet all of the goals of stateful serverless set out to accomplish as aside from 'limitless scaling" and no server maintenance.

We felt the most pain when implementing Proto.Actors grains. We found that it is lacking documentation giving a good overview of different ways to create and message actors. We had implemented Proto.Actors benchmarks twice, once with auto-generating code that required request-response style of communication and after using process ID's which allow for messages to be sent to grains mailboxes. When auto-generating code, we found that Visual Studio would not find the reloaded generated file after we made a change to our protobuf file. This required us to restart Visual Studio multiple times multiple times as we developed our benchmarks. An-

other issue we had with the generated code was running into dead locks when running some of our benchmarks such as big.

Other limitations include the time it took to execute the tests and collect the findings. Some tests such as pingpong need to execute 40 tests to collect metrics on each situation. Each test can take anywhere from 1 minute to 5 minutes depending on the settings. On top of that, all tests have 2 versions, a distributed and virtual version. This multiplies the test time in 2 making full retests for just one test (pingpong) take up to 3 hours!

Finally, to test the cluster on the raspberry pis, I had to compile the benchmarks into arm64 docker images. This took a lot of time, with each release build of an image taking at least 5 minutes. It was not bad for Proto.Actor as one image could run a test in both a distributed and virtual setting. However, for Orleans, a unique image for each node was needed to test distributed workloads. This was because if a image was compiled with all of the included grains, but told to only use one of those grains, it would not respect what you programmed and search the included binaries for grains compiled in the project. Thus, for a test like ping pong, 4 images was needed to be built. Tests such as Big required 5 images. This limitation could be elevated by developing on an computer with an arm chip or using a stronger computer.

5.3 Future Work

There are a number of suggestions for future work such as comparing throughput performance of virtual actors to distributed actor model frameworks such as CAF and Erlang. Adding more benchmarks would also help strengthen the results we see here and would help in understanding the best performing framework. One such benchmark could compare how virtual actor frameworks scale out during times of high load. It would be interesting to see how this compares to other serverless solutions as well. Increasing the specs of the machines would also be an interesting addition to our work and would hopefully show performance scales linearly.

Bibliography

[1] Ansible. URL https://www.ansible.com/. Accessed: 2021-03-22.

[2] Apache flink, . URL https://flink.apache.org/. Accessed: 2021-03-05.

[3] Apache hadoop, . URL https://hadoop.apache.org/. Accessed: 2021-03-05.

[4] Apache spark, . URL https://spark.apache.org/. Accessed: 2021-03-05.

[5] Amazon web services, . URL https://aws.amazon.com/. Accessed: 2021-03-05.

[6] Aws lambda, . URL https://aws.amazon.com/lambda/. Accessed: 2021-03-05.

[7] Step functions, . URL https://aws.amazon.com/step-functions/. Accessed: 2021-03-05.

[8] Azure, . URL https://azure.microsoft.com/. Accessed: 2021-03-05.

[9] Durable functions, . URL https://docs.microsoft.com/en-us/azure/azure-functions/durable/. Accessed: 2021-03-05.

[10] Azure functions, . URL https://azure.microsoft.com/en-us/services/functions/. Accessed: 2021-03-05.

[11] Cloudstate. URL https://cloudstate.io/. Accessed: 2021-03-05.

[12] Google cloud functions. URL https://cloud.google.com/functions. Accessed: 2021-03-05.

[13] Aws lambda. URL https://aws.amazon.com/lambda/. Accessed: 2021-03-05.

[14] kubernetes. URL https://kubernetes.io/. Accessed: 2021-03-28.

[15] V8: Google's open source high-performance javascript and webassembly engine. URL https://v8.dev. Accessed: 2021-03-05.

[16] G. Adzic and R. Chatley. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 884–889, 2017.

[17] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, Feb. 2020. USENIX Association. ISBN 978-1-939133-13-7. URL `https://www.usenix.org/conference/nsdi20/presentation/agache`.

[18] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, 2003.

[19] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A scalability benchmark suite for erlang/otp. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 33–42, 2012.

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

[21] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41*, 2014.

[22] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, Jan. 2016. ISSN 1542-7730. doi: 10.1145/2898442.2898444. URL `https://doi.org/10.1145/2898442.2898444`.

[23] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[24] R. Chard, T. J. Skluzacek, Z. Li, Y. Babuji, A. Woodard, B. Blaiszik, S. Tuecke, I. Foster, and K. Chard. Serverless supercomputing: High performance function as a service for science. *arXiv preprint arXiv:1908.04907*, 2019.

[25] D. Charousset, R. Hiesgen, and T. C. Schmidt. Caf-the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 15–28, 2014.

[26] Datadog. The state of serverless, 2020. URL `https://www.datadoghq.com/state-of-serverless/`.

[27] A. Ellis. Openfaas. `https://github.com/openfaas/faas`, 2021.

[28] 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), pages 363–376, Boston, MA, Mar. 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[29] J. Geerling. The fastest usb storage options for raspberry pi, Aug 2020. URL https://www.jeffgeerling.com/blog/2020/fastest-usb-storage-options-raspberry-pi.

[30] Google. Protobuf. https://github.com/protocolbuffers/protobuf, 2008.

[31] Google. grpc. https://github.com/grpc/grpc.io, 2016.

[32] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[33] C. Hewitt. Actor model of computation: Scalable robust information systems, 2015.

[34] S. M. Imam and V. Sarkar. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, pages 67–80, 2014.

[35] R. Johansson. Protoactor. https://github.com/asynkron/protoactor-dotnet, 2016.

[36] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. *CoRR*, abs/1702.04024, 2017. URL http://arxiv.org/abs/1702.04024.

[37] M. N. A. Khan. React++: A lightweight actor framework in c++. Master's thesis, University of Waterloo, 2020.

[38] X. C. Lin, J. E. Gonzalez, and J. M. Hellerstein. Serverless boom or bust? an analysis of economic incentives. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020. URL https://www.usenix.org/conference/hotcloud20/presentation/lin.

[39] M. Mohajeri Parizi, G. Sileno, T. van Engers, and S. Klous. Run, agent, run! architecture and benchmarking of actor-based agents. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 11–20, 2020.

[40] T. Naumenko and A. Petrenko. Analysis of problems of storage and processing of data in serverless technologies. *Technology Audit and Production Reserves*, 2 (2):58, 2021.

[41] S. Shillaker and P. Pietzuch. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 419–433, 2020.

[42] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[43] K. Varda. Introducing cloudflare workers: Run javascript service workers at the edge. *Cloudflare Blog*, 2017.

[44] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[45] T. Zhang, D. Xie, F. Li, and R. Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.