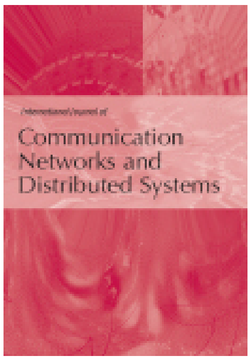


International Journal of Communication Networks and Distributed Systems

ISSN online: 1754-3924 - ISSN print: 1754-3916
https://www.inderscience.com/ijcnds

An objective comparison of two prominent virtual actor frameworks: Proto.Actor and Orleans

Edward R. Sykes, Alec DiVito

DOI: 10.1504/IJCNDS.2024.10057792

Article History:

Received:	14 October 2022
Last revised:	18 April 2023
Accepted:	21 April 2023
Published online:	30 April 2024

An objective comparison of two prominent virtual actor frameworks: Proto.Actor and Orleans

Edward R. Sykes* and Alec DiVito

Centre for Mobile Innovation,
Sheridan College,
Oakville, Ontario, Canada
Email: ed.sykes@sheridancollege.ca
Email: divitoa@sheridancollege.ca
*Corresponding author

Abstract: Recently there has been a significant increase in developing distributed systems easily and rapidly. Driven by the demand of software communities, developers seek tools and frameworks that abstract away low-level details of the underlying distributed system and the need to understand complex details on how the system works. Researchers have explored serverless frameworks, distributed key value stores, distributed stream processing frameworks and distributed actor frameworks. Currently, stateful serverless applications and distributed actor models may be the answer to what developers need. In this paper, we present a review of stateful distributed computing frameworks, and the results of experiments that compare Orleans and Proto.Actor – two popular actor model frameworks – running on Kubernetes. We discovered that the Proto.Actor performs at least two times faster than Orleans, but is more complex to learn. We present the results of these tests, and provide a discussion of future research opportunities highlighting virtual actor model frameworks.

Keywords: actor model; distributed computing; microbenchmark tests; serverless computing; virtual actor frameworks.

Reference to this paper should be made as follows: Sykes, E.R. and DiVito, A. (2024) ‘An objective comparison of two prominent virtual actor frameworks: Proto.Actor and Orleans’, *Int. J. Communication Networks and Distributed Systems*, Vol. 30, No. 3, pp.313–342.

Biographical notes: Edward R. Sykes has an Honours BSc (1992) in Mathematics and Computer Science (Summa Cum Laude) from McMaster University, a MSc (2005) in Computer Science from McMaster University and a PhD (2012) in Computer Science from the University of Guelph. He has a BEd (1993) and Ontario Teaching Certificate (OTC) from the University of Western Ontario, a MEd (1998) and PhD (2006) in Education (Cognition and Learning) from Brock University. He is the Director of Sheridan College’s Centre for Mobile Innovation (CMI), a research centre that focuses on focuses on mobile health (mHealth) and creates innovative solutions using mobile technologies, artificial intelligence/machine learning, wearable computing, and augmented/virtual reality. He is also an Adjunct

Associate Professor in the Computing and Software Department at McMaster University and an Affiliate Scientist at the University Health Network, University of Toronto.

Alec DiVito is a Senior Cloud Engineer in Toronto helping bring the power of the cloud to RBC employees using tools like Terraform and Kubernetes. He graduated from Sheridan College with a degree in Mobile Computing with a specific interest in virtual actor model frameworks. He also helped to run the Hackville, a Sheridan College-based hackathon, and raised $12,000 for its participants. In his personal time he enjoys learning about databases and distributed system and use the newly found knowledge to write his own. He also enjoys riding his bike and spending time outdoors.

1 Introduction

Developers have been searching for easier ways to create applications that effortlessly scale when run in distributed environments. Recently, cloud providers have invested heavily in the serverless computing area. The advancement in serverless has been noticed by developers and companies as the adoption rate for serverless has been growing year over year (Lin et al., 2020; Datadog, 2020). Serverless promises to handle all scaling infrastructure to meet demand, on demand, which is an appealing characteristic. It enables developers to focus on creating solutions and less time on configuring environments and reducing the chance of costly mistakes (Adzic and Chatley, 2017). Popular providers who are offering these services are AWS with Lambda, Azure with Azure Functions and CloudFlare with CloudFlare workers.

Current reviews show that a stateful distributed framework should consist of the following characteristics: scalability, portability, statefulness, coordination and messaging, predictable performance and short initialisation times (Hewitt, 2015; Agache et al., 2020; Shahrad et al., 2020; Sreekanti et al., 2020). Of these six characteristics, we discovered that serverless has one characteristic (scalability), stateful serverless has two characteristics (scalability and statefulness), and actor frameworks have all 6 except scalability and portability. Fortunately, other technologies exist (e.g., Kubernetes), that satisfy these two characteristics collectively providing a full system solution for developers.

The actor model is a conceptual model for concurrent computation which originated in 1973 (Hewitt, 2015). An *actor* is a unit of computation which can only do the following:

1 create a finite number of actors

2 send a finite number of messages

3 determine how to process the message once it is received.

Each actor contains its own private state and can only interact with other actors through message passing. Since each actor own its own state, there is no need for lock-based synchronisation. When actors send messages, they are put into the receiving actors mailbox (queue). Once there, the actor can process one message at a time in a first-in-first-out (FIFO) fashion. All messages contain the address of the sender, so actors are able to reply to messages they have received. The actor model was popularised by telecom companies using Erlang (Armstrong, 2003).

Although actor languages and libraries such as Erlang aim to simplify distributed system programming, it still is a burden on developers because of its low level of abstraction and inherent complexity. Developers 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 (Armstrong, 2003). These challenges led to the development of Microsoft Orleans (Bernstein et al., 2014). Orleans is a framework that aims to simplify and manage all these complexities so the developer would not need to be a distributed systems expert to develop distributed applications.

Orleans provides a paradigm referred to as the *virtual actor model* which raises the level of actor abstraction. In Orleans, *actors* are virtual entities that always exist in the virtual environment. Actors 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 for other system purposes. Finally, if an actor is marked as stateless, Orleans will run like a serverless function where multiple types of the same actor will be created and be granted access to pull from the same mailbox, thus allowing hot stateless actor scaling.

Proto.Actor is another virtual actor model framework that was created in 2016 (Johansson, 2016). It was released and marketed as the ‘next generation actor model framework’. It was heavily inspired by Orleans (Johansson, 2016). *Proto.Actor* was created using popular open source technologies and satisfies the same characteristics as Orleans plus one unique addition; Proto.Actor also provides outside systems a way to communicate directly with actors inside the system without a client connection. 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, serialisation and more (Johansson, 2016). In contrast, Proto.Actor focused on concurrency and distributed programming. Proto.Actor adopts the Orleans model for fault tolerance. It uses gRPC for communicating between systems. When deployed as a cluster, it must be setup with a clustering technology such as Kubernetes or Consul. The framework provides wide support for the developer community by offering choice of the following popular programming languages: C#, Kotlin and Golang (i.e., the *Go Programming Language*).

This paper is organised as follows, Section 2 presents the background where we present an overview of the current serverless, stateless serverless and actor model frameworks. Section 3 presents the methodology by which we tested and evaluated the selected frameworks using well-defined microbenchmarks. Section 4 reports our findings, Section 5 presents a discussion, and Section 6 provides a conclusion.

2 Background

This background section present a review of serverless, stateful serverless, actor model frameworks and virtual actor model frameworks created by companies, researchers and open source developers.

2.1 Serverless

Serverless computing has gained popularity in recent years with multiple cloud providers offering support for their serverless services (AWS Lambda, 2023; Durable Functions, 2023; Google Cloud Functions, 2023; IBM Serverless Functions, 2023) and other companies developing their own serverless platforms (Varda, 2017). These innovations have given developers a powerful abstraction that allow them to upload code into the cloud which is then accessible by a handle such as a URL. If the handle is ever access or triggered, the cloud provider will run the code.

Serverless functions are scalable and work best when a workload is stateless and embarrassingly parallel, but when they are not, it can become problematic (Naumenko and Petrenko, 2021). One of the issues is that each function does not have an unique address so messaging between functions is impossible. Another issue is maintaining state; as functions only live for a short amount of time (approximately 15 minutes or less) that state must be kept inside a storage system and shipped to the function when requested (Hellerstein et al., 2018). Current serverless solutions that are popular today are AWS Lambda (2023), Azure Functions (2023), and CloudFlare workers (Varda, 2017).

2.2 Actor model frameworks

New actor model systems incorporate distribution strategies by default into their frameworks. This alleviates developers from needing to understand how the program is executing behind the scenes and instead enables them to focus on developing the logic for an actor (Charousset et al., 2014). These frameworks guarantee that an actor will receive at most one message. They also provide the concept of a virtual actor which Orleans has popularised as a *grain* which is an actor that always exists (virtually). In this section, we review two virtual actor frameworks: *Orleans* and *Proto.Actor*.

2.2.1 Orleans

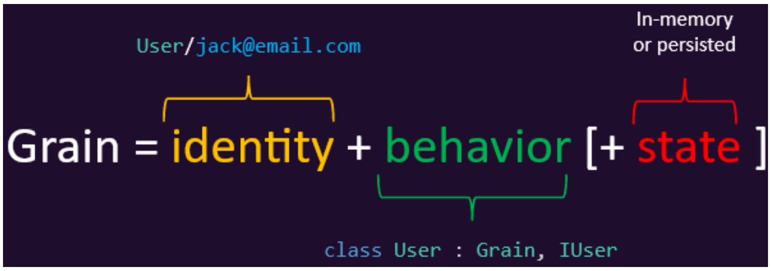
Orleans is a framework that was developed by Microsoft Research in 2011 and was released as an open source project in 2015 (Nelson, 2022; Astbury, 2022). It has been used in production with the most notable example being used as the backend for Halo 3 (an extremely popular first person role playing game) (Astbury, 2022). Orleans introduced and popularised the concept of a *virtual grain*, which is an actor that virtually lives forever in a system even if it does not physically exist. *Grains* must implement an interface which defines the actor's behavior and must be declared and run inside of a *Silo* – a server that runs *grains* for the Orlean's framework.

A visual overview of a grain can be seen in Figure 1. A *grain* consists of an identity, a behaviour (defined as an interface) and optionally the state (Nelson, 2022; Astbury, 2022). *Grains* are created at runtime when needed and contain all their state information 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. Since 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.

Grains run in a distributed setting and are given primary keys that serve as the grain's identity. These identities can be a globally unique identifier (GUID), a long integer, or a string. An example of this identity field is shown in Figure 1. Grains are implemented using the singleton design pattern, so, empty GUID values are typically used (Chard et al., 2019; Gamma et al., 1994). When creating a new grain the placement of it is by default random, however, this can be configured by the developer.

Grains can be configured to acquire state from persistent storage upon activation. In Orleans, these types of objects are called *data objects*. Grains can interact with data objects inside of its state if it is wrapped in a persistent state interface. Although reading the state is automatic on the initialisation of a grain, the developer decides when state must be saved back to persistent storage. When saving the updated state, Orleans may create a deep copy of the data object state which it will use to save the data to storage.

Figure 1 Components of a *grain* in Orleans *grain* (see online version for colours)


User/jack@email.com
In-memory or persisted
Grain = identity + behavior [+ state]
class User : Grain, IUser

Source: Agache et al. (2020)

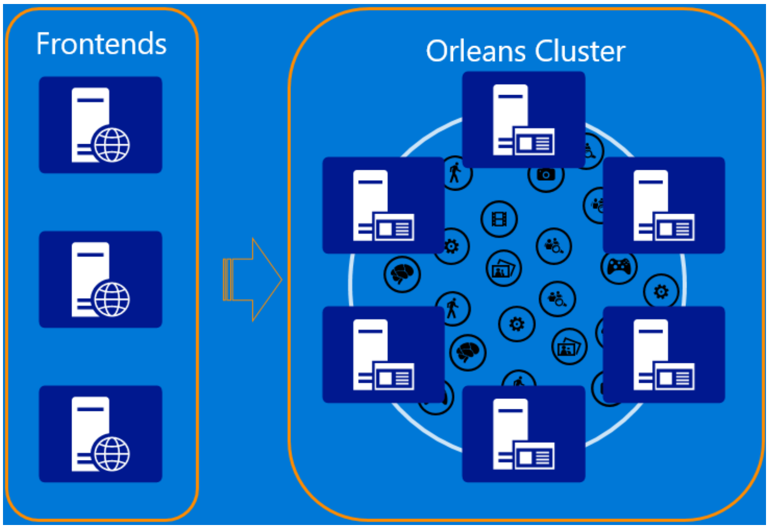
Silos are used to host grains (Nelson, 2022; Astbury, 2022). Multiple Silos can be run together to form a *cluster*. Typically, Silos are run as a cluster for scalability and fault-tolerance. When run as a cluster, Silos work together to distribute work, and detect and recover from failures. Third party systems can send messages to a cluster by creating a client which connect to a cluster from the outside. Clients can co-host a machine with a Silo. However, typically, clients and Silos are run separately on different 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 manage all the available grains.

A Silo can learn about which Silos support different grains through their Silo membership protocol. Every Silo that joins the cluster must add themselves to the membership table. This table is stored in a database enabling access for every node (e.g., SQL Server and DynamoDB) and all in-memory virtual grains. Failures are detected using a peer-to-peer algorithm such as, *Chord* (Stoica et al., 2003) that helps the cluster reach agreement on the active (alive) nodes in the cluster. The membership table helps nodes find each other and coordinate agreement of the membership view.

Silos store grain identities within a data structure called the *grain directory* which is a global registry for registered grains that is built on a peer-to-peer system like Chord (Nelson, 2022; Astbury, 2022). 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 on other Silos. The *grain directory* is responsible for mapping a grain to its correct Silo. By default, it is a consistent in-memory distributed directory which is partitioned across a cluster of Silos. The directory maps grain keys to the location of the grain on a particular Silo in the cluster. It is implemented as a distributed hash table where each Silo holds a partition of the directory (Nelson, 2022; Astbury, 2022).

Accessing an Orleans cluster is normally done through a frontend as Orleans is meant to be run as a 'backend' system. Frontends can be any type of server through which a user can interact (Nelson, 2022; Astbury, 2022). These frontend servers can create clients into the Orleans cluster where they can then invoke requests. This is depicted in Figure 2.

Figure 2 Frontend servers connecting to Orleans cluster (see online version for colours)



Source: Nelson (2022)

2.2.2 Proto.Actor

Proto.Actor is another virtual actor model framework that has been implemented in several popular programming languages including C#, Kotlin and Go (Johansson, 2016). The design of Proto.Actor originated from Orleans (Nelson, 2022). *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 libraries to handle service discovery and message serialisation. In this section, a review of how Proto.Actor works from a single machine to a cluster is presented.

Actors in *Proto.Actor* are containers for state, behaviour, a mailbox, child actors and a supervisor strategy. When an actor is created, it produces a process ID (PID) which is the actor's location in either a local or remote setting. Inside the actor is state information which may change based on incoming messages and the behaviour needed to handle new messages that are received. Other actors can send an actor a message which will be enqueued to a client's 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. Actors also include another mailbox which is only used for system messages for managing the actor's context. Actors are not required to typecheck incoming messages, instead, an actor can receive any type of message and determine the most effective way to perform it.

Actors can be in different states through the life cycle of the application, namely: *Started*, *Alive*, *Stopping*, *Restarting*, and *Stopped*. Some of these states emit special events which the actor can hook onto. The *Started* event happens just before an actor processes its first message and can load initialisation state into the actor at this time. For *Restarting* and *Stopping*, 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.

Grains are virtual actors that are identified by cluster identities and are used to point to the location of the grain. Like Orleans, grains are spawned by the framework and are stopped when they are no longer used. Their style of messaging differs from actors by recommending to follow a request/response style of communication. Messages can be sent to the actor through its cluster identity (PID such as a URL). A cluster identity is made up of a Kind (an object) and an Identity (an objects ID). Grains implement a *Protobuf* interface which generates the methods for communicating with the virtual actor.

Messages in *Proto.Actor* are used to communicate with actors. Messages are immutable and sent through the system asynchronously. Actors eventually get the methods and process them later. If messages fail to be revived by a mailbox, they become dead letters. When this event happens, they are pushed onto an EventStream where such messages will be handled.

Actors can send methods in two ways. The first is by using a *Send* call which is a non-blocking, fire-and-forget operation. It is the preferred method of execution unless a request and response type of communication is required (Johansson, 2016). If a response is required, an actor can execute a *Request*. The difference between *Request* and *Send* is that *Request* includes the senders PID while *Send* does not.

Virtual actors in *Proto.Actor* are created by defining a *Protocol Buffers* or *Protobuf* contract. It is an interface definition language that was developed by Google which allows users to create a *contract* between services using a domain specific language. Virtual actors then implement the contract produced by the generated code. At startup, nodes declare all of the virtual actors that they support. At runtime when virtual actors are created, they have *location transparency* meaning that their deployed location is random but the cluster will be able to still route messages to them. If the node they are running on fails, they will be restarted on another node. The way that virtual actors can be spawn is configurable using round-robin, local affinity (spawn the actor close to the creator) or through a custom implementation. A requirement for virtual actors in Proto.Actor is that all communication using them must use request/response messaging.

Proto.Actor can be a normal actor model system with the ability to spawn actors when connecting two systems using peer-to-peer strategies. It becomes a virtual actor framework when it is configured as a cluster. *Proto.Actor* leverages existing cluster member management technology such as Consuel, ETCD and Kubernetes (Johansson, 2016) to learn about all of the nodes in the cluster and each of their available virtual actors. Nodes gossip with each other to share their state with other nodes in the cluster. The location of spawned virtual actors are kept inside of an actor cache on each node and inside of the identity lookup service. It can be configured to act as a distributed hash table where each nodes keep a slice of the entire state or through an external database.

3 Methodology

This section presents the microbenchmark experiments that were used for testing the virtual actor model frameworks including the evaluation criteria, tools, hardware environment, benchmark test configuration, data collection and the evaluation techniques. Microbenchmark testing is a well-defined and accepted method to evaluate specific parts of a working system (Parizi et al., 2020; Imam and Sarkar, 2014; Khan, 2020).

3.1 Evaluation criteria

Microbenchmarking have been used successfully to evaluate the performance of other distributed systems (Parizi et al., 2020; Imam and Sarkar, 2014; Khan, 2020). In this study, we took inspiration from existing single node microbenchmarks and converted them to work in a distributed setting. The following metrics were used in the measuring the performance of *Proto.Actor* and *Orleans*.

- *Throughput:* This metric is calculated by taking the number of requests the system completes and dividing it by the time it takes to complete test:

  $$throughput = \frac{number\ of\ requests}{completion\ time} \tag{1}$$

  The *throughput* metric is used to understand how many messages can be processed by the actor system for a specific task. The number of messages sent is kept as a count inside the actor and when the test concludes, the count is collected and divided by the test time. This measurement is used to measure performance per actor and the overall system.

- *Latency:* Latency is used to measure the 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 required to complete a request. It is also used to compare performance between virtual and distributed tests.

- *Actors:* Actors are the basic building blocks for an actor model system. All tests include measuring the impact of increasing the number of actors in the system which implicitly places increasing load on system. Consequently, this metric measures the performance degradation of the system as more actors are added (Parizi et al., 2020). It is an important metric because actors need to share CPU time and understanding the overhead of context switching is informative on how performant a given framework is as well as the best ratio of actors to a CPU core (Khan, 2020).

- *Message size:* This metric is used to measure the system's capabilities based on the size of the messages sent between actors. By experimenting with sending different sized messages, we gain a more thorough understanding of how a workload on a framework may perform if it is required to be passing large pieces of data between actors. It is an important metric which is used to give insights into the framework's performance when there is a need to pass data between actors.

3.2 Tools

All microbenchmarks were conducted in a controlled environment to ensure a fair playing field for evaluating the actor model frameworks. This implies that no cloud computers or services were used as they may have altered the results. Kubernetes was used to create a cluster for each framework. Kubernetes 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 pods when they are initialised inside of Kubernetes. The cluster was set up with *Ansible* (Ansible, 2023) which makes redeploying the cluster reliable, repeatable and efficient. The motivation for why Kubernetes and Ansible were chosen for this study is described in the following section.

3.2.1 Kubernetes

Kubernetes is a system that runs and coordinates containerised application on a cluster of computers (Kubernetes, 2023). Kubernetes is quickly becoming the standard across the cluster development community (Cloudstate, 2023; Sreekanti et al., 2020). It takes the role of managing the lifecycle of a deployed application and provide applications with high availability and the ability to scale automatically. In our tests, we used Kubernetes by creating a consistent framework for collecting the metrics and saving the results to a file for later processing. Furthermore, frameworks such as *Orleans* and *Proto.Actor* both make use of Kubernetes for its clustering characteristics. Kubernetes enables easy deployment of applications which also supports efficient framework testing.

Kubernetes enabled the creation of a fair test environment because it simplified the setup process for the microbenchmarks. Once the deployment files are 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 (Kubernetes, 2023). As a result, as long as the Docker container is built to contain the test, it can be performed on any set of machines running Kubernetes (Boettiger, 2015; Miell and Sayers, 2019).

3.2.2 Ansible

Ansible is a tool that automates executing scripts on remote or local systems (Ansible, 2023). In order to effectively test distributed actor model frameworks, it is essential to consistently setup the test environment in a consistent, and repeatable manner. Ansible is a tool that is developed by Red Hat that helps automate managing systems from the command line and YAML scripts (Ansible, 2023). Developers use Ansible to develop playbooks which contain YAML markup that contain instructions to be executed on a remote computer(s) (Ansible, 2023). These instructions could install, start, stop, uninstall services or even run shell scripts. Developing YAML scripts for our environment reduced the chances for changes to happen between tests and thus increased reliability, and internal and external validity of our study.

3.2.3 Hardware environment

All tests were deployed onto a Kubernetes cluster consisting of one Raspberry Pi 4B, 4 GB edition which was configured to be the cluster *master node* and three Raspberry Pi 4B, 8 GB editions which were the *worker nodes*. All tests were run on the worker nodes; the master node only ran Kubernetes-specific workloads such as, deciding the placement of new pods and managing Kubernetes API server which developers have access to. The cluster was initialised using the tool *kubeadm* which helps initialise Kubernetes clusters (Kubernetes, 2023). To setup the Kubernetes environment on the nodes, we used an Ansible script which automated the cluster setup process. All tests were 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 were programmed into their own respective frameworks. After each test, metrics were collected from all actors and saved to a CSV file. All tests collected information on latency between requests and responses, the number of actors used, the message sizes and throughput of the system. Overall, all tests measured the total time taken to accomplish specific tasks. All Raspberry Pi's were equipped with SSDs. This was done to increase the stability and reliability of the cluster and to improve the consistency of the performance of the cluster and the metrics collected (Geerling, 2020).

3.3 Benchmark test configuration

A collection of tests were conducted on *Orleans* and *Proto.Actor*. Each test was executed in the exact same fashion. A Dockerfile(s) was built which contained code for the server, and client (Boettiger, 2015). A YAML script was created to deploy the container. Every server instance was deployed using a Pod. Each test client then got executed as a Job.

In this section, we present the tests that were performed to collect data on the previously described metrics:

1 throughput

2 latency

3 actors

4 message size.

Each test encapsulated at least one metric previously defined.

3.3.1 Distributed ping pong

The distributed ping pong test involves initialising two pods inside the Kubernetes cluster. One pod knows of the ping actors while the other pod knows of the pong actors. To complete the test, each ping actor contacts a pong actor for a defined amount of times. To start the tests, a client pod remotely creates many ping and pong actors. After they have been created and initialised with the message size in bytes and the number of messages to be sent and received to complete the test, the client tells each ping actor to start. An example of the actors used in this test is shown in Listing 1 and an example of the entire life cycle of the test is presented in Figure 3. While the ping actors run, they collect data on the latency of each ping pong message and provide a list to the client. From this test, the analysis of the latency can be conducted between two actors that are run on separate pods as well as the throughput of how many messages that can be sent every second.

Listing 1 Example ping pong microbenchmark (see online version for colours)

```
*** LISTING 1 ***
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);
   }
}
```

3.3.2 Virtual ping pong

Based on the distributed ping pong test, the *virtual ping pong test* is different in that both pods know of the ping and pong actors. The framework manages the placement of actors in the cluster. This test enables comparisons between the virtual and distributed ping pong tests to determine performance differences of the framework when it manages the placement strategy of the actors (Parizi et al., 2020). Figure 4 presents an example highlighting the differences between the two tests.

3.3.3 Distributed divide and conquer

The *distributed divide and conquer* test, based on the MergeSort, involves three pods deployed to a cluster where each pod has a unique type of divide and conquer actor that relates to the type of pod they are running on. For example, ‘*sort A*’ specifies the sort() function to be executed on ‘*pod A*’. The client initialises one of the actors on a pod with a list of numbers that the actor needs to sort by dividing the list into two and then sending it to two other actors. This process continues until the list size is less than or equal to two. Once the list has completed dividing, it returns the completed sorted list of numbers. A generalised example of this test is shown in Listing 2. This test measures the time a framework takes to create actors that work together to complete a task.

Listing 2 Example of a divide and conquor microbenchmark (see online version for colours)

```
*** LISTING 2 ***
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()
      }
   }
}
```

Figure 3 Complete sequence diagram of distributed ping pong test (see online version for colours)

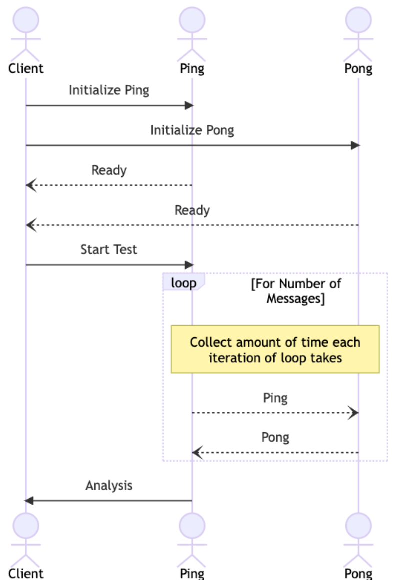


Figure 4 Difference between distributed ping pong test and virtual ping pong test (see online version for colours)

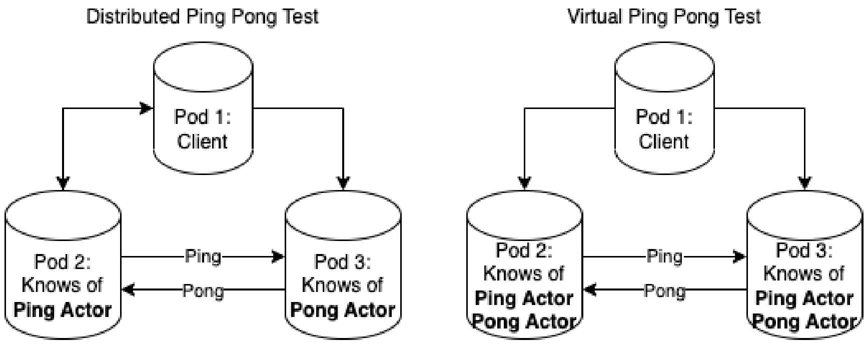
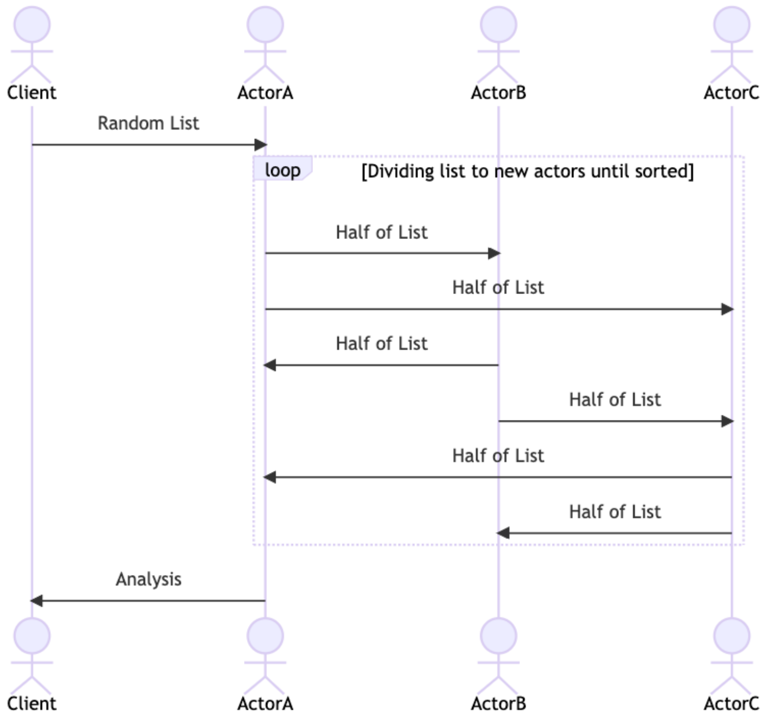

Distributed Ping Pong Test
Pod 1: Client
Pod 2: Knows of **Ping Actor**
Pod 3: Knows of **Pong Actor**
Ping
Pong
Virtual Ping Pong Test
Pod 1: Client
Pod 2: Knows of **Ping Actor** **Pong Actor**
Pod 3: Knows of **Ping Actor** **Pong Actor**
Ping
Pong

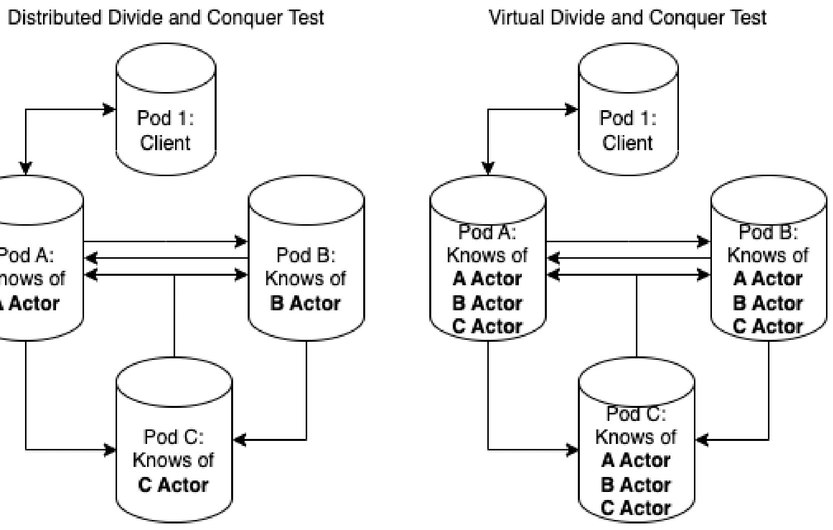
Figure 5 Sequence diagram of how actors message each other and divide a list



3.3.4 Virtual divide and conquer

Similar to the distributed divide and conquer test, the *virtual divide and conquer test* expects each pod to know of every type of actor that can be created inside the system. Every time an actor creates another actor, it is the framework's responsibility to assign it a location inside the cluster on one of the pods. This test measures how long it takes to create many new actors and divide the list of numbers. Figure 5 presents a sequence diagram visualising this test and interactions. Using the results from this test, we also analysed the difference in speed from our distributed test. The differences between the distributed and virtual divide and conquer tests are shown in Figure 6.

Figure 6 Differences between *distributed* and *virtual divide and conquer tests*


Distributed Divide and Conquer Test
Pod 1: Client
Pod A: Knows of A Actor
Pod B: Knows of B Actor
Pod C: Knows of C Actor
Virtual Divide and Conquer Test
Pod 1: Client
Pod A: Knows of A Actor B Actor C Actor
Pod B: Knows of A Actor B Actor C Actor
Pod C: Knows of A Actor B Actor C Actor

3.3.5 Distributed big

The *distributed big* microbenchmark test is commonly used for testing mailbox contention (Aronis et al., 2012). The test was run on three pods that were deployed to the cluster. Similar to *distributed divide and conquer*, there were three pods with each pod having the same actor but with a different name so that they were unique to that pod. A client creates and initialises N actors across all three pods. During the initialisation, an actor is told about all existing actors in the system. When the test is run, each actor sends a message to all 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 each other until the test is complete. This sequence is visualised in Figure 7 and Listing 3 presents a code example of this test. This test measures the increased latency due to the many-to-many messages induced by the test. It also facilitates measurements on the throughput of the system when overloaded with internal messaging.

Listing 3 Example of a big actor for the big microbenchmark (see online version for colours)

```
*** LISTING 3 ***
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)
    }
}
```

3.3.6 Virtual big

Similar to *distributed big*, this test sends many-to-many messages across a collection of actors. The difference is that, in *virtual big*, all pods in the cluster know all the actors in the system, so a new actor can be spawned on any machine. This is visualised in Figure 8. The results of this test are compared to its counterpart distributed test. It facilitates the measurement of how much of a performance difference exists when communication is not forced over the network.

3.4 Test settings

All tests were completed on a Kubernetes cluster. For each test, we repeated the following steps:

1 create the servers that support Orleans and Proto.Actor

2 execute a client that connects to the servers and start the test

3 collect and save metrics once the test completed

4 destroy and cleanup 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.

Figure 7 Sequence diagram of big test (see online version for colours)

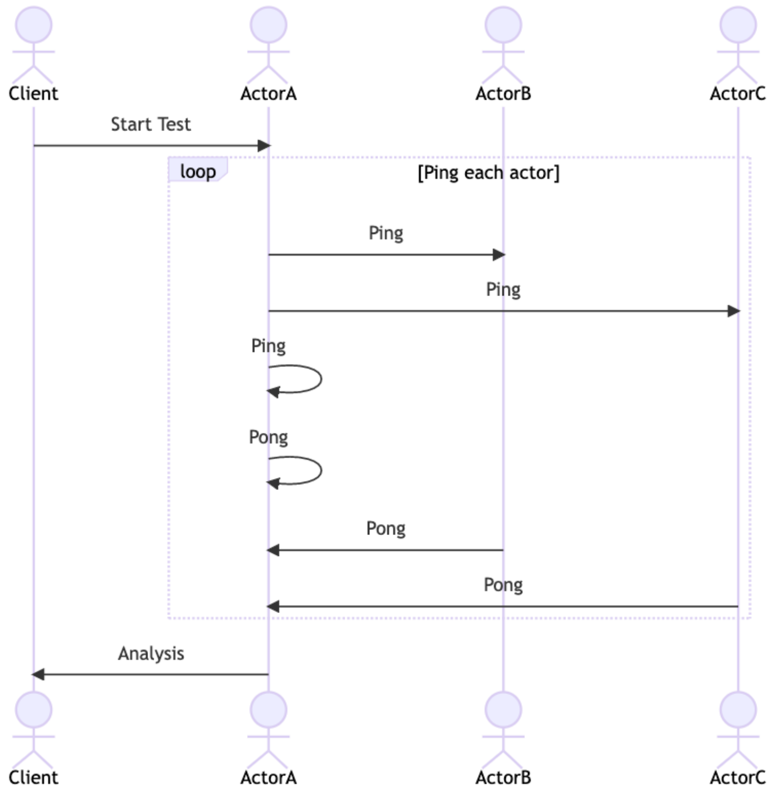
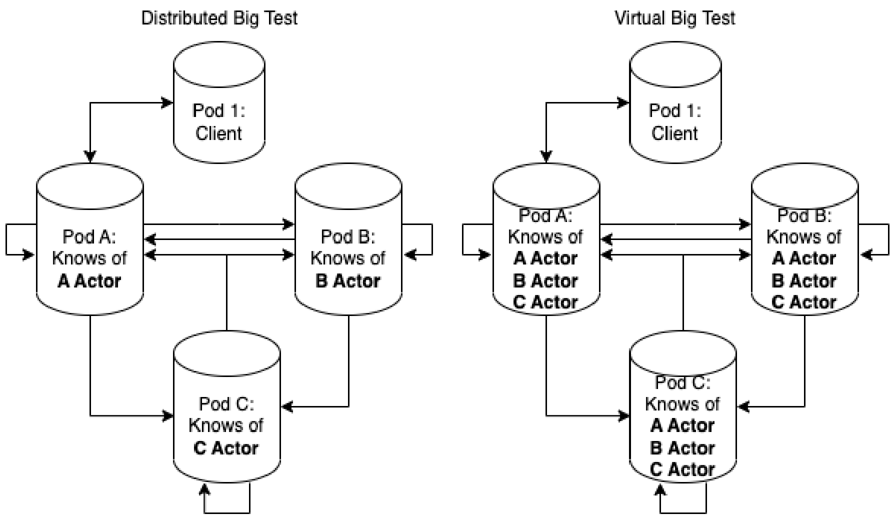


Figure 8 Difference between distributed big test and virtual big test



3.5 Analysis and evaluation methodology

The evaluation and analysis methodology involved reviewing the results of the data collected by measuring the total throughput of a variety of tests; computing the standard descriptive statistics (i.e., mean, median, mode, skewness, standard deviation, etc.); and the distribution of the latency. The data was collected from completing the following tests:

1. distributed ping pong
2. virtual ping pong
3. distributed divide and conquer
4. virtual divide and conquer
5. distributed big
6. virtual big.

4 Findings (analysis and evaluation)

In this section we present our findings, analysis and evaluation of the tests conducted.

4.1 Ping pong

The ping pong test facilitated the collection of 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 and pongs. In all the tests, we sent 7,500 ping and 7,500 pong messages. This test was repeated with 3, 6, 12, 24, 48 and 64 actors along with 0, 512, 1,024, 2,048 byte size messages. The results are presented in Figures 9, 10, 11 and 12. Both *Orleans* and *Proto.Actor*'s performance improved when the test was run in a virtual setting by spreading out the load. Overall, *Proto.Actor* achieved between 1.7 to 2.5× increased throughput over *Orleans*.

Switching from distributed to virtual actors in Orleans saw throughput increase on average of 1.3×. As actor counts went up, we saw throughput increase to a high of 1.5× times when using 64 actors. For Proto.Actor, there was an average increase factor of 1.25. However, it differed from Orleans by most actors achieving a 1.25× increase when using virtual actors. Figure 13 shows the performance difference between distributed and virtual actor's throughput as a percentage.

Along with the throughput, we were also able to calculate average latency times by message size which is visualised in Figures 14 and 15. When virtual actors are deployed, the latency times between messages were between 1.0 and 9.0 seconds for Orleans, and 1.0 and 4.5 seconds for Proto.Actor. Results from the distributed actors were consistent with values achieved from the virtual actors, however, the latency times doubled. Virtual latencies did not exhibit this pattern; most message sizes took approximately the same amount of time. In contrast with the distributed mode, the latencies observed for communications 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 shown in Table 1. The table also shows that distributed actors scale linearly and consistently while virtual actors may have spikes in their latencies. It should be noted that Proto.Actor performs better at the 99.9% quartile consistently in virtual and distributed tests.

Figure 9 Throughput of virtual actors in Orleans (see online version for colours)

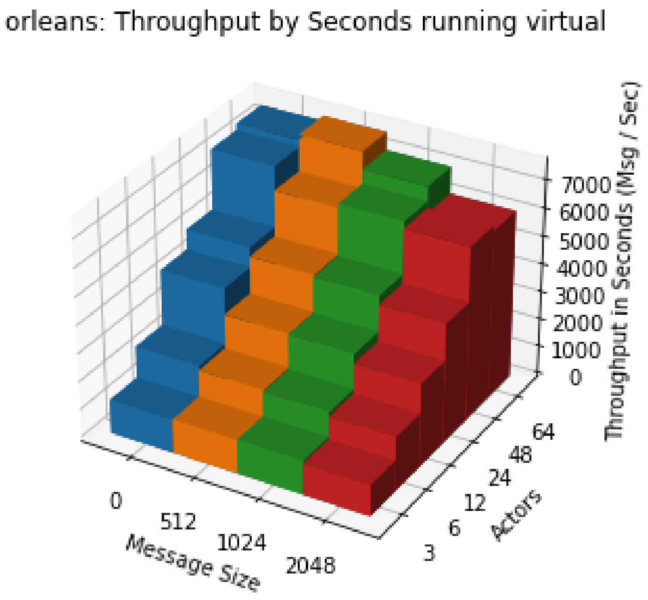


Figure 10 Throughput of distributed actors in Orleans (see online version for colours)

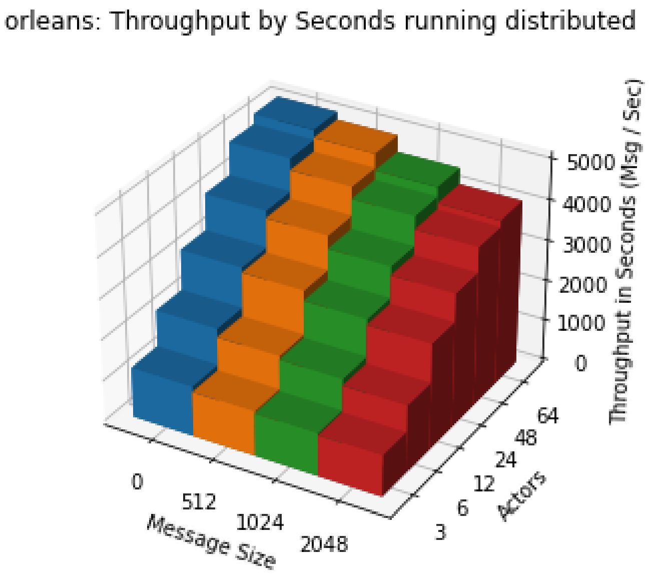


Figure 11 Throughput of virtual actors in Proto.Actor (see online version for colours)

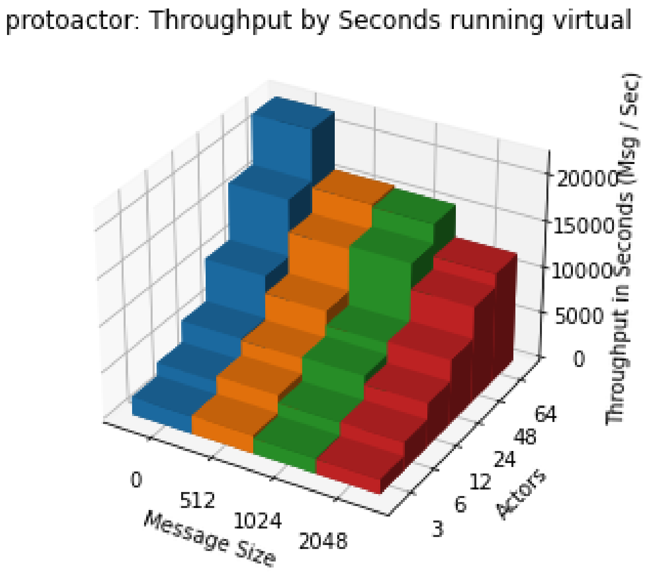
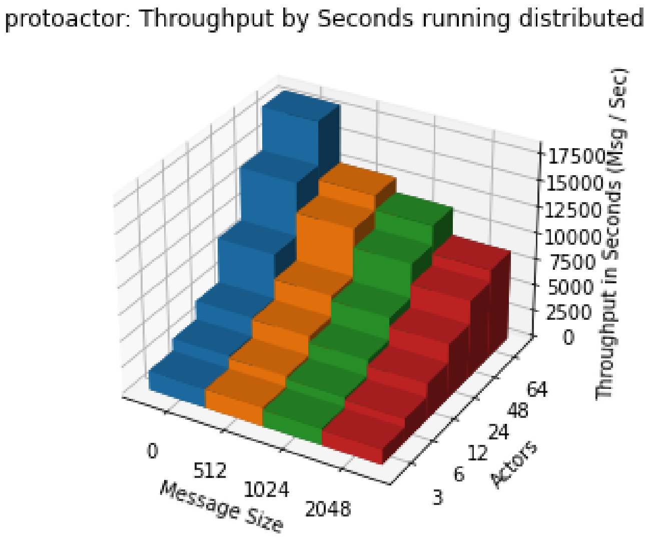


Figure 12 Throughput of distributed actors in Proto.Actor (see online version for colours)



4.2 Sort

In the *Sort* test, the performance of Orleans and Proto.Actor was assessed by sorting a list of random integers using actors. This test primary evaluates the cost of spawning new virtual actors (grains) inside of the cluster. This test was performed with lists sizes of 1,000, 2,000, 4,000, 8,000, and 12,000.

Figure 13 Virtual actor throughput increase as a percentage compared to distributed actors (see online version for colours)

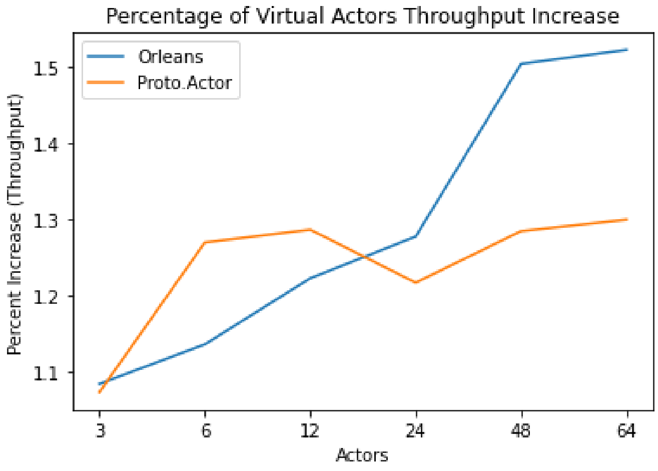


Figure 14 Latency of ping and pong for virtual actors (see online version for colours)

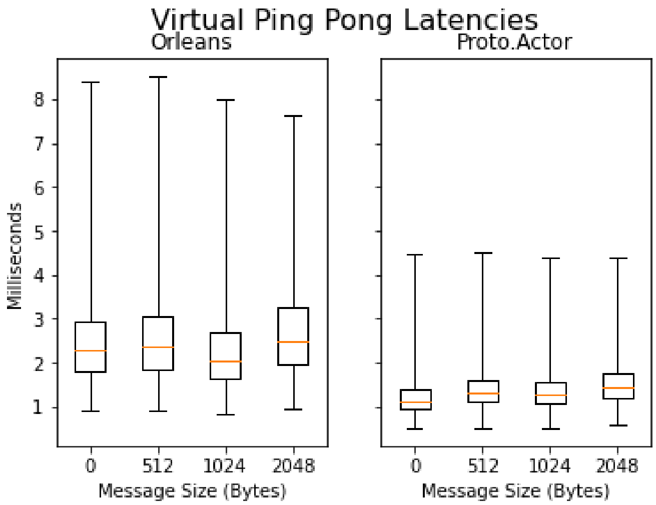


Table 1 Ping pong worst case latencies in milliseconds

Framework	*Setting*	*Min*	*Mean*	*Quartile (95)*	*Quartile (99)*	*Quartile (99.9)*	*Max*	*Max STD*
Orleans	Distributed	1.607	6.950	15.760	19.220	32.918	86.993	4.460
Proto.Actor	Distributed	1.089	3.484	7.620	10.022	22.976	61.357	10.311
Orleans	Virtual	0.896	2.849	5.632	8.127	39.277	800.323	195.259
Proto.Actor	Virtual	0.534	1.461	3.253	4.446	11.940	73.486	5.631

Figure 16 shows the difference in performance for Orleans for distributed and virtual workloads. The results show that, on average, virtual grains perform better than their distributed counterparts. This is likely due to the fact that Orlean's default placement algorithm is random, thus grains are placed all over the cluster. For that reason we see when sorting a list of 8,000 items, virtual sorting was slower. From these results we can conclude that grain creation for Orleans has consistent performance in both virtual and distributed settings.

Figure 15 Latency of ping and pong for distributed actors (see online version for colours)

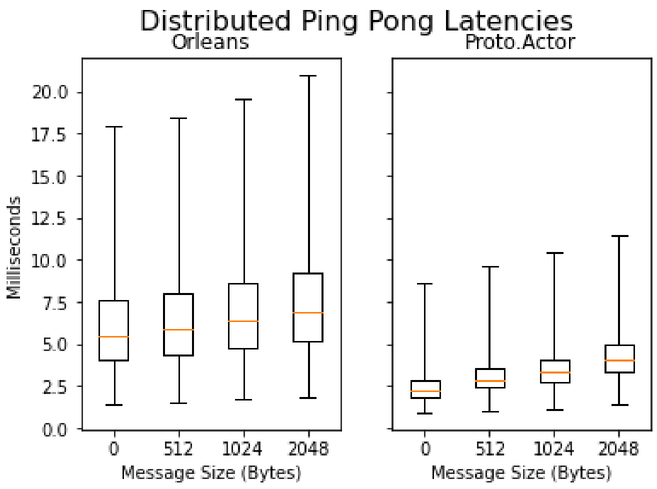


Figure 16 Performance of virtual and distributed compared actors sorting list of integers using Orleans (see online version for colours)

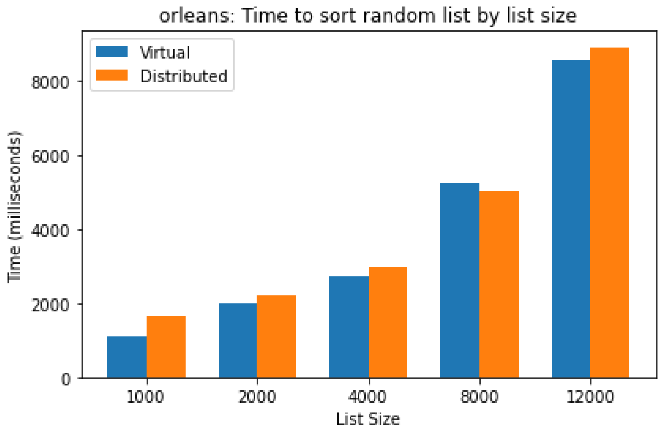


Figure 17 shows the results for Proto.Actor for the *sort* microbenchmark test. The performance difference between sorting a list with virtual and distributed is not consistent. However, the results indicate that sorting a list with virtual actors is equal, if not faster than sorting with only distributed actors.

Figure 17 Performance of virtual and distributed compared actors sorting list of integers using Proto.Actor (see online version for colours)

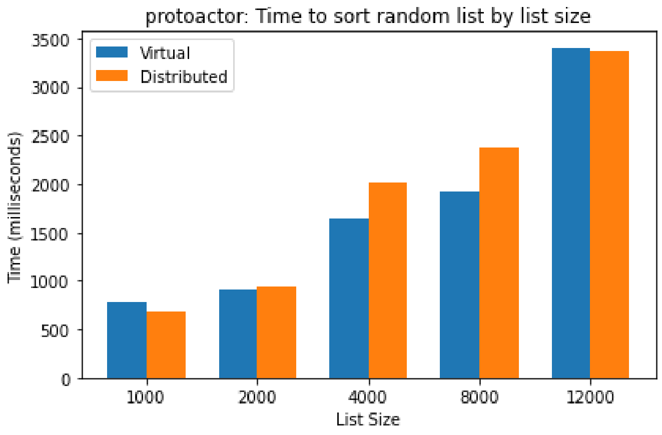
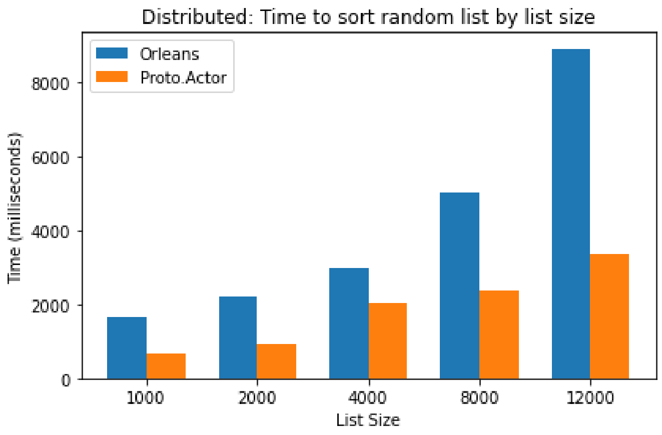


Figure 18 Time needed to sort random list of numbers using distributed actors (see online version for colours)



The performance results between the two frameworks can be viewed through Figure 18 and Figure 19. In both graphs, Orleans takes considerably more time to complete the *sort* test when compared to Proto.Actor. On average, Orleans requires more than two times more time to complete a sort. The possible reasons for this is that Orlean's placement strategy is two times more expensive than Proto.Actor's, thus, it takes longer to complete the test.

4.3 Big

In the *big* test, the performance of mailbox contention was tested by having every grain messaging each other in a many-to-many fashion. In all the tests, 5,000 ping and 5,000 pong messages were sent and received. This test was repeated with 3, 6, 12, 24 actors along with 0, 512, 1,024, 2,048 byte message sizes. We attempted to also include 48 and 64 actors, however, both Orleans and Proto.Actor failed to complete the tests because our system was not able to handle the load. General results for throughput can be viewed in Figures 20 and 21 for Orleans and Figures 22 and 23 for Proto.Actor.

Figure 19 Time needed to sort random list of numbers using virtual actors (see online version for colours)

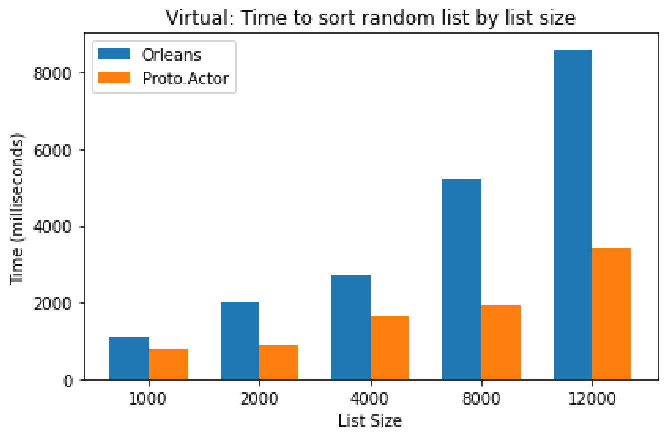
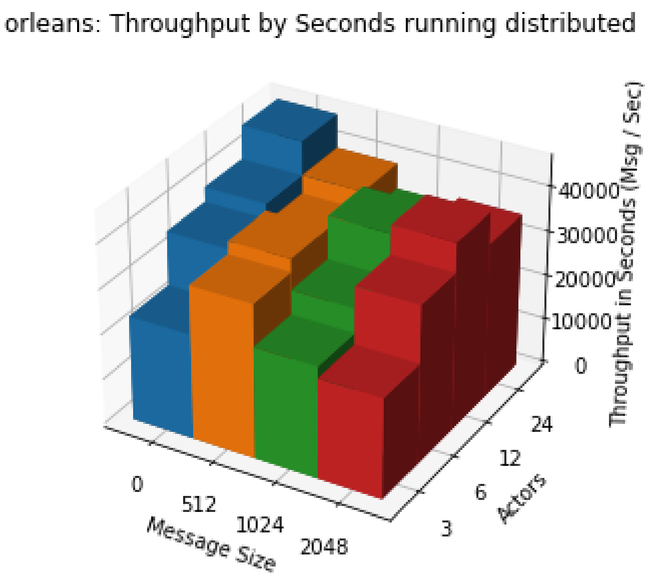


Figure 20 Orleans many-to-many messaging using distributed actors (see online version for colours)



Overall, distributed grains were able to perform better than their virtual counterparts (see Figures 20 and 21). 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 per second. We believe this is because the grains were evenly distributed across all nodes while in a virtual environment grains are placed randomly.

Figure 21 Orleans many-to-many messaging using virtual actors (see online version for colours)

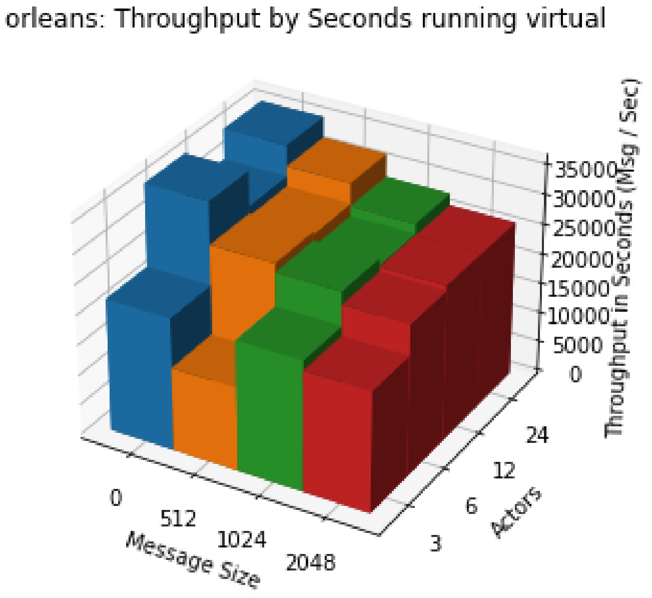


Figure 22 Proto.Actor many-to-many messaging using distributed actors (see online version for colours)

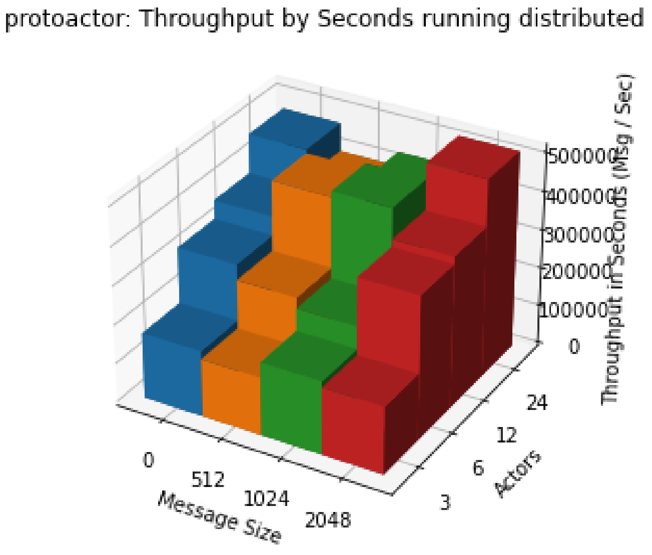


Figure 23 Proto.Actor many-to-many messaging using virtual actors (see online version for colours)

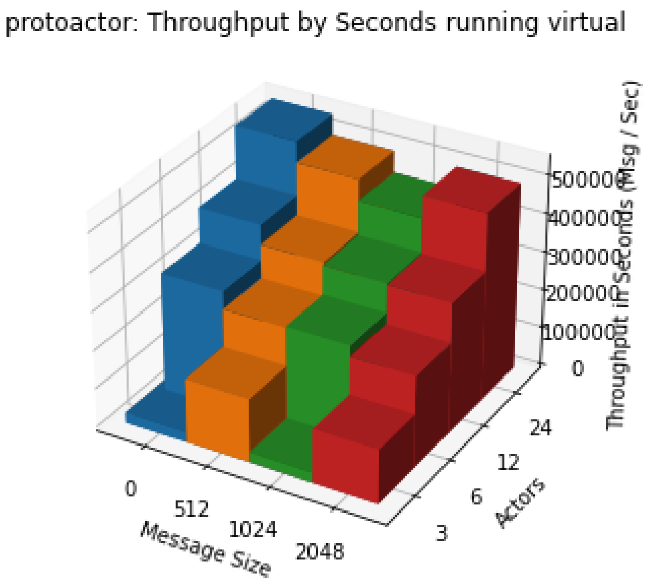
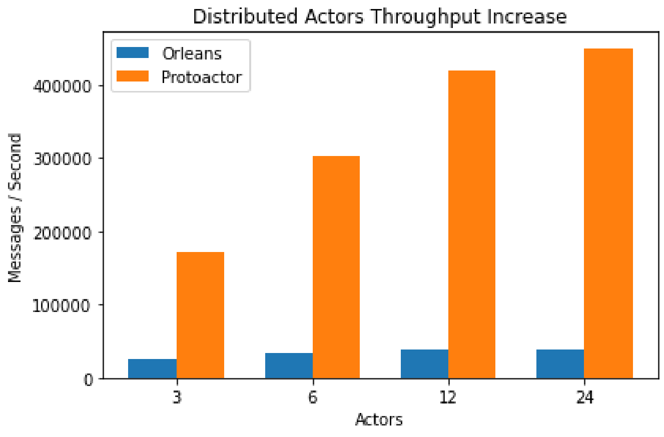
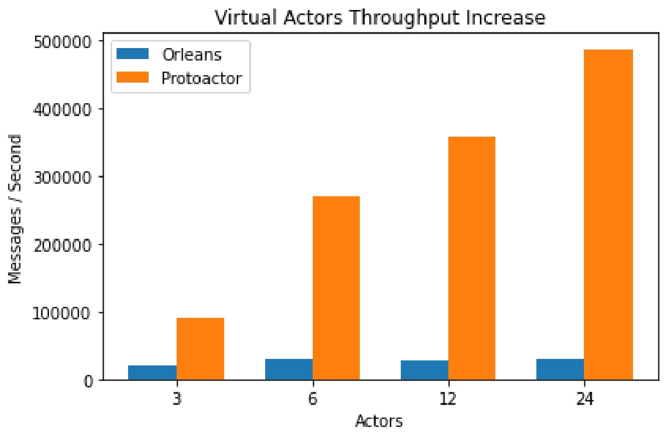


Figure 24 Performance different in big test using distributed actors (see online version for colours)



Proto.Actor outperformed Orleans by almost a factor of 11×. 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 per 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 are visualised in Figures 22 and 23. To show this performance difference clearly, we compared the average overall performance by actor group. These results are shown in Figures 24 and 25. These findings indicate that for both distributed and virtual workloads, Proto.Actor significantly outperforms Orleans.

Figure 25 Performance different in Big test using virtual actors (see online version for colours)



5 Discussion

5.1 Actor models: the future of serverless

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 than 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*. Orleans was built to allow for normal use of the actor model in an automated distributed setting. *Proto.Actor*, on the other hand, was developed from the idea that a developer should choose when to use a virtual or normal actor.

We also discovered that when pairing virtual actor frameworks with a cluster technology such as Kubernetes, we can create systems that are similar to serverless services. Although running our own servers is different from using serverless, we gain 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 user's 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 in designing, programming and developing our tests. However, we did run into some issues when running our benchmarks because Orleans scans the binaries included in the docker 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 node needed to be packaged with only one actor compiled inside the container. This inevitably increased build and development time.

When developing with Proto.Actor, we had different difficulties. One of the larger problems was troubleshooting issues without being able to contact someone for assistance. Documentation is lacking and although it is an open source project, there was not a lot of activity or community support available. Currently, Proto.Actor has bug bounty programs posted on Github to improve documentation for some monitory value. Because of the lack of documentation, we ended up developing Proto.Actors benchmarks twice. Once using auto generated code that required virtual actors to 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 to become proficient with this framework.

From a performance perspective, this paper focused on researching and evaluating the performance of virtual actors of both frameworks. We found that between point-to-point communication, Proto.Actor is up to two times faster than 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 than two times faster than Orleans. Finally, our results show that when communicating many-to-many, Proto.Actor is over ten times faster than Orleans. From our results, we can confidently say that Proto.Actor is more performant than Orleans.

Based on our findings, future virtual actor model frameworks should include ways of 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 increased adoption, systems should also include a way of allowing messages to be sent to actors directly without a front-end. Finally, frameworks should also include a way to automatically add new grains without shutting down or restarting the server as this is commonplace in serverless technologies.

6 Conclusions

In this section, we provide a summary of main findings of this work, the limitations and suggestions for future work.

6.1 Summary

In this study, we set out to find the best performing virtual actor framework from the two popular frameworks – *Orleans* and *Proto.Actor*. We conducted a variety of microbenchmark tests to analyse their performance by measuring latency when sending and receiving messages, and system throughputs when under load.

Our results showed that overall, Proto.Actor allows for faster communication between actors and faster instantiation of new actors. Proto.Actor consistently performed at least two times faster than Orleans in the tests, most likely due to its low latency in efficient inter-actor communication.

Both frameworks performed well overall, however, each one has specific advantages over the other based on different workloads in different settings. Orleans offers an advantage over Proto.Actor when used for a new project or as a service that will only run in the backend as a micro-service. Orleans has a good, comprehensive documentation set and the library has a large community following that provides support and are routinely adding features to the framework. This makes Orleans easier for developers to quickly learn and troubleshoot issues effectively.

Proto.Actor is a better choice for integrating into an existing system with its reliance on *protobuf* and *gRPC* thus not requiring the need for a frontend to the system. The framework has been fully implemented in C#, Go and Kotlin which allows for greater adoption, if a certain programming language is a requirement. As presented in the findings, Proto.Actor is also very fast. However, the learning curve is higher than Orleans as it allows the developer multiple ways to implement virtual and normal actors. In comparison to Orleans, Proto.Actor currently lacks comprehensive documentation which may make setup, and initial orientation a challenge.

In the spirit of furthering science and this work, the following resources are provided:

- Microbenchmark test scripts: https://bitbucket.org/ed_sykes_team/orleans_vs_proto.actor/src/master/scripts/.
- Data collected from the tests: https://bitbucket.org/ed_sykes_team/orleans_vs_proto.actor/src/master/data/.

We hope this will encourage other researchers to explore and extend our work.

6.2 Limitations

During this study, several limitations were identified. The first is with implementing Proto.Actors grains. We found that Proto.Actor's lack of documentation made it difficult to gain a thorough understanding of how to create and message actors. We implemented Proto.Actors benchmarks twice, once with auto-generated code that required request-response style of communication and after using process IDs which allow for messages to be sent to grain's 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 as we developed our benchmarks. Another issue we had with the generated code was running into deadlocks while running some of our benchmarks such as *big*.

Other limitations included 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 run. Each test took between 1 and 5 minutes, depending on the settings. Furthermore, all tests have two versions, a *distributed* and a *virtual* version. This multiplied the test time by two making full retests for just one test (e.g., PingPong), take up to three hours.

To run the tests on the cluster on the Raspberry PIs, we compiled the benchmarks into arm64 Docker images. This took a considerable amount of time, with each release build of an image taking at least five minutes. It was not too onerous 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 an image was compiled with all the included grains, but used only one of those grains, it would not respect what was programmed and search the included binaries for grains compiled in the project instead. Thus, for a test like PingPong, four images were needed to be built. Tests such as big required five images. This limitation could be mitigated by developing on a computer with an arm chip or using a more powerful computer.

6.3 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 presented in this study and increase our understanding the relative strengths and limitations of each 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. Lastly, increasing the specifications of the machines would also be an area of future work to explore the impact on performance scaling.

References

Adzic, G. and Chatley, R. (2017) 'Serverless computing: economic and architectural impact', in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp.884–889.

Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P. and Popa, D-M. (2020) 'Firecracker: lightweight virtualization for serverless applications', in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pp.419–434.

Ansible (2023) [online] https://www.ansible.com/ (accessed 22 March 2023).

Armstrong, J. (2003) *Making Reliable Distributed Systems in the Presence of Software Errors*, PhD thesis.

Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y. and Venetis, I.E. (2012) 'A scalability benchmark suite for Erlang/OTP', in *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, pp.33–42.

Astbury, R. (2022) *Orleans Dashboard*, Apress, Berkeley, CA, pp.37–44.

AWS Lambda (2023) [online] https://aws.amazon.com/lambda/ (accessed 5 March 2023).

Azure Functions (2023) [online] https://azure.microsoft.com/en-us/services/functions/ (accessed 5 March 2023).

Bernstein, P., Bykov, S., Geller, A., Kliot, G. and Thelin, J. (2014) 'Orleans: distributed virtual actors for programmability and scalability', *MSR-TR-2014–41*.

Boettiger, C. (2015) 'An introduction to docker for reproducible research', *SIGOPS Oper. Syst. Rev.*, January, Vol. 49, No. 1, pp.71–79.

Chard, R., Skluzacek, T.J., Li, Z., Babuji, Y., Woodard, A., Blaiszik, B., Tuecke, S., Foster, I. and Chard, K. (2019) *Serverless Supercomputing: High Performance Function as a Service for Science*, arXiv preprint arXiv:1908.04907.

Charousset, D., Hiesgen, R. and Schmidt, T.C. (2014) '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*, pp.15–28.

Cloudstate (2023) [online] https://cloudstate.io/ (accessed 5 March 2023).

Datadog (2020) *The State of Serverless* [online] https://www.datadoghq.com/state-of-serverless/ (accessed 20 May 2023).

Durable Functions (2023) [online] https://docs.microsoft.com/en-us/azure/azure-functions/durable/ (accessed 5 March 2023).

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Pearson Education.

Geerling, J. (2020) *The Fastest USB Storage Options for Raspberry Pi*, August.

Google Cloud Functions (2023) [online] https://cloud.google.com/functions (accessed 5 March 2023).

Hellerstein, J.M., Faleiro, J., Gonzalez, J.E., Schleier-Smith, J., Sreekanti, V., Tumanov, A. and Wu, C. (2018) *Serverless Computing: One Step Forward, Two Steps Back*, arXiv preprint arXiv:1812.03651.

Hewitt, C. (2015) 'Actor model of computation: scalable robust information systems: one IoT is no IoT1', *Symposium on Logic and Collaboration for Intelligent Applications*, Stanford, USA, hal-01163534v7.

IBM Serverless Functions (2023) [online] https://www.ibm.com/topics/serverless (accessed 5 March 2023).

Imam, S.M. and Sarkar, V. (2014) '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*, pp.67–80.

Johansson, R. (2016) *Proto.Actor* [online] https://github.com/asynkron/protoactor-dotnet (accessed 10 February 2023).

Khan, A.N. (2020) *React++: A Lightweight Actor Framework in C++*, Master's thesis, University of Waterloo.

Kubernetes (2023) [online] https://kubernetes.io/ (accessed 28 March 2023).

Lin, X.C., Gonzalez, J.E. and Hellerstein, J.M. (2020) 'Serverless boom or bust? An analysis of economic incentives', in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, USENIX Association, July.

Miell, I. and Sayers, A. (2019) *Docker in Practice*, Simon and Schuster.

Naumenko, T. and Petrenko, A. (2021) 'Analysis of problems of storage and processing of data in serverless technologies', *Technology Audit and Production Reserves*, Vol. 2, No. 2, p.58.

Nelson, T. (2022) *A Primer on Microsoft Orleans and the Actor Model*, Apress, Berkeley, CA, pp.1–15.

Parizi, M.M., Sileno, G., van Engers, T. and Klous, S. (2020) '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*, pp.11–20.

Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M. and Bianchini, R. (2020) 'Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider', in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, July, pp.205–218.

Sreekanti, V., Wu, C., Lin, X.C., Schleier-Smith, J., Faleiro, J.M., Gonzalez, J.E., Hellerstein, J.M. and Tumanov, A. (2020) *Cloudburst: Stateful Functions-as-a-Service*, arXiv preprint arXiv:2001.04592.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F. and Balakrishnan, H. (2003) 'Chord: a scalable peer-to-peer lookup protocol for internet applications', *IEEE/ACM Transactions on Networking*, Vol. 11, No. 1, pp.17–32.

Varda, K. (2017) *Introducing CloudFlare Workers: Run Javascript Service Workers at the Edge*, CloudFlare Blog.