VRIJE
UNIVERSITEIT
BRUSSEL

Bachelor thesis submitted in partial fulfillment of the requirements for the degree of bachelor of science: Computer Science

DELIVERING ACTORS WITH STORK

A distributed computing deployment tool

Gérard Lichtert

June 3, 2024

Promotors: Prof. Dr. Joeri de Koster and Prof. Dr. Wolfgang de Meuter.
Advisor: Mathijs Saey

Sciences and Bio-Engineering sciences

Contents

1 Introduction

In a world of electronics and machines where power consumption is always increasing, optimizing power consumption is becoming increasingly important. While electricity is expensive, the main reason lies in climate change. This is because the majority of our electricity production still comes from non-renewable sources, such as coal, oil, and gas [1]. These sources produce a lot of CO^2, and other greenhouse gases, which are the main contributors to climate change. While there is research being done to optimize energy generation, optimizing power consumption is becoming increasingly important. This means that we as programmers can also play a role in optimizing our programs to consume less energy.

In the world of computing, cloud computing is responsible for 1% of the worldwide energy consumption [2]. To make use of the cloud, we do not only need electricity to power the devices that provide cloud computing but also electricity to power the network, which also plays a significant role in energy consumption. This is because the network is the backbone of most if not all, communication between devices and applications. The amount of connected devices and applications keeps growing, which leads to higher network usage. Consequently, higher network usage also leads to higher energy consumption, which leads to a higher carbon footprint [3]. Higher network usage also requires better infrastructure to handle the increased data volume, which in turn also requires more energy [4]. To reduce the network load we need to look at the data that is being sent through the network and if we can reduce it.

Data is usually transported through the network for a few reasons. Sometimes it is to send data to a device to update local data, like a chat message that needs to be added to the chat, or a new email that needs to be downloaded. While often compressed, the data is used as-is, and thus cannot be further reduced. Other times, however, data is sent to be processed. This can potentially be optimized by applying the edge-computing principle. This means that (part of) the data that is originally meant for processing is processed locally first, potentially reducing the amount of data that needs to be sent after preprocessing it. Logically, the network load, and consequently the energy consumption, should be reduced. However, for a certain set of devices or applications, it could be optimal to pre-process the complete set of data prior to sending it over the network, while for another set of devices or applications, it could be optimal to send the data directly to the server. Sometimes, however, the optimal configuration could be a combination of the two. We want to be able to experiment with these different configurations. In this work, we focus on the Python programming language, as it is a popular language often used for data science or AI applications running in the cloud. For this purpose, we introduce Stork.

Stork is a distributed computing deployment tool written in Python that makes it possible to deploy distributed systems and try out different configurations regarding these systems. More concretely, it allows us to initialise the deployment of a distributed system, change the configuration of where each part of the distributed system is deployed in a declarative way, and reverse the deployment.

In this thesis, we discuss Stork, how a user can use it and how it is implemented. We start by discussing the background of Stork. Followed by how a user can use Stork. We then discuss how Stork is implemented and finalize the thesis with a conclusion.

2 Background

Before we discuss Stork, we discuss the goals that Stork should achieve and how. As mentioned in the introduction, we require a tool that allows us to declare where which data gets processed. This means that we need to be able to deploy parts of our program to different devices. Consequently, this means that our tool needs to work in the context of distributed systems. For this a distributed computing paradigm is required that allows us to deploy our system in a distributed way, as well as change where parts of our programs reside. Next, we discuss the available technologies and libraries in Python and choose the one(s) that best fit our needs.

2.1 Distributed computing paradigm

To start with distributed computing, a suitable distributed computing paradigm that allows our system to be deployed to the cloud without much manual intervention, yet helps us with experimenting different configurations, is required. There are several options, such as:

- Message Passing Interface (MPI)[5]
- Remote Procedure Call (RPC)[6]
- Shared Memory Model[7]
- The Actor Model[8]

An Actor is a computational unit that encapsulates its state and behaviour, interacting with each other through asynchronous message passing. Each Actor has a mailbox in which it receives these messages and processes them sequentially. The encapsulation property of the Actor Model allows Actors to be very modular, as we can see each Actor as a separate unit that processes part of the data. More concretely, each actor will be responsible for processing a single part of the data. This allows us to easily move parts of the data processing pipeline from one device to another. This is important because we want to be able to change where data gets processed without redefining how it gets processed.

While it is technically possible to achieve modularity in MPI, RPC, and the Shared Memory Model, due to the tight coupling of those models, it is harder to achieve. This is because the state and behaviour of the system are not encapsulated in a single entity but rather spread across the entire system. This makes it harder to move parts of the system around. Consequently, we choose the Actor Model as the distributed computing paradigm for Stork.

Using an IoT system as a running example because IoT devices account for 50% of networked devices[9]. Say the system exists out of three devices with sensors and a cloud, where the devices feed the sensor data to the cloud. Once the data is in the cloud, it gets processed and stored in a database. It is possible to encapsulate the behaviour of each sensor in an Actor and connect this with another Actor responsible for sending the data to a server, where the data gets sent through a series of Actors that are each in turn responsible for processing a single part of the data, just like in Figure 1.

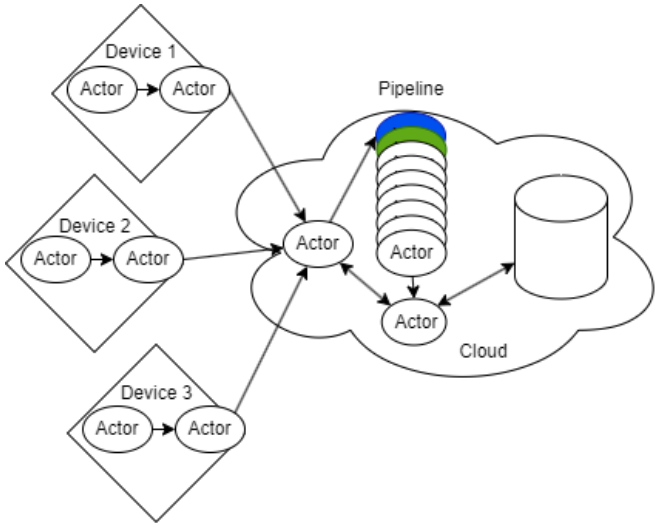

Figure 1: An example of an IoT system using the Actor Model

Conceptually, we should be able to spawn some of the Actors that process data from the pipeline to the IoT devices and place them before the Actor that sends the data to the Server. These Actors will pre-process data prior to sending it to the cloud, where the data will go through the rest of the pipeline. The preprocessing Actors that have been moved are coloured blue and green. Achieving one of the possible configurations without having redefined how the data gets processed.

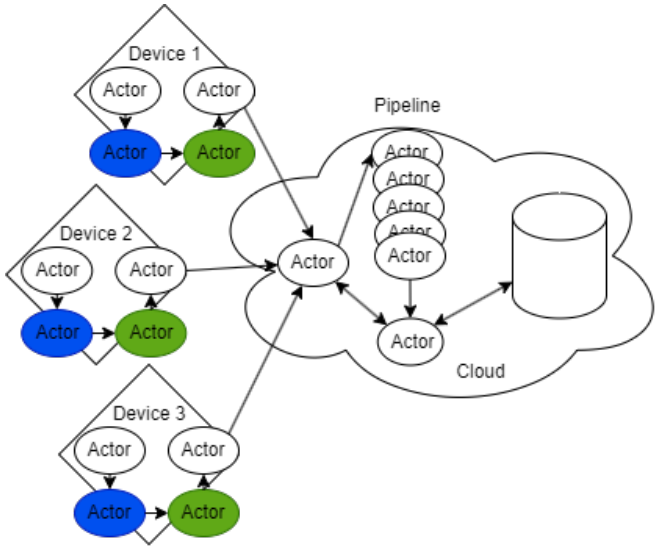

Figure 2: An example of an IoT system where some of the Actors that process data are spawned on the IoT devices instead of the server.

2.2 Implementing Actors in Python

When using Stork to deploy distributed systems and try out different configurations, Actors have to be created. However, before we can discuss how to use Stork, we discuss how to implement Actors in Python. In Python, there are two libraries that allow us to implement the Actor Model: Pykka[1] and Thespian[2]. Thespian is the only library that supports distribution and distributed communication out of the box, while Pykka does not. Because of this, Thespian is the library of choice for Stork, and consequently, Stork only works with Thespian Actors.

To create an Actor, a class must inherit from a Thespian Actor class. Furthermore, this Actor must at least implement the `receiveMessage` method or equivalent, depending on the Actor class that is extended. As mentioned in subsection 2.1., Actors communicate through asynchronous messaging. This means that when the Actor receives a message, it calls the `receiveMessage` method. An example of an Actor implementation is given below. This Actor will respond with "Hello World! back" when it receives the message "are you there?".

```
1 from thespian.actors import Actor
2
3 class HelloWorld(Actor):
4
5     def receiveMessage(self, message, sender):
6         if message == "are you there?":
7             self.send(sender, "Hello World! back")
```

Listing 1: Actor example

3 Stork

Stork is a distributed computing deployment tool, written in Python, that makes it possible to deploy an actor application over a distributed system in several different ways with minimum changes to the code. More concretely, Stork is a library that automates Actor creation on remote devices. Currently a user has to either SSH to each device to create Actors on a device, or deal with the fact that an Actor class can be initialized on a single remote device, from the cloud. Stork works around having to manually SSH to each device to create Actors and allows the user to create multiple instances of an Actor class on different remote devices by delivering the Actor class to the requested devices and creating the Actor on the device.

To illustrate the usage of Stork, we have the following example, which we will reuse throughout this section. The example consists of three remote devices and a server, which will be referred to as the leader of the distributed system. On the remote devices, data is generated. The goal of this system is to process a list of lists of integers. Out of each list of integers, the average is computed followed by the square of the average. Afterwards the data is stored on the database. As an initial configuration, where all the processing is done on the server we have the example depicted in Figure 3.

[1] https://pykka.readthedocs.io/en/stable/
[2] https://thespianpy.com/doc/

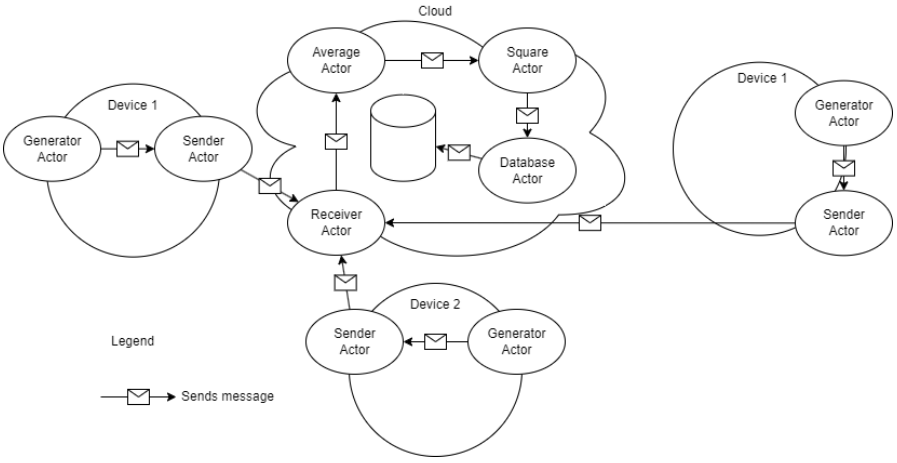

Figure 3: An initial configuration where the entire pipeline is in the cloud.

Here the generator Actors send a list of integers to a Sender Actor, which' sole responsibility is to forward data to the Receiver Actor in the server. The Receiver Actor forwards the received data to the processing pipeline which consists of the Average Actor and the Square Actor. Each computing the average and the square of the average respectively. The processed data is then forwarded to the Database Actor, which stores it in the database.

Another configuration is that the average is computed prior to the data being sent to the server. This means that the Average Actor would have to come before the Sender Actor on the remote devices. This configuration is depicted in Figure 4.

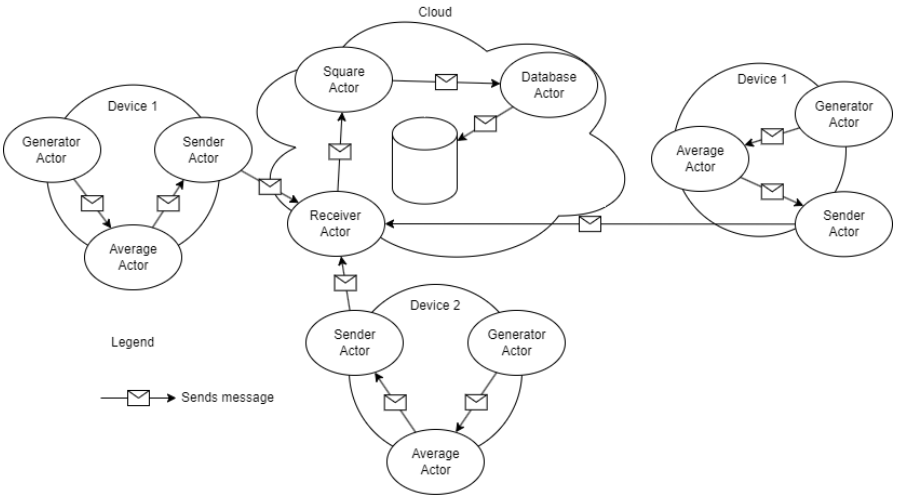

Figure 4: A secondary configuration where the average is computed on the remote devices.

Apart from the fact that the average is now computed prior to it being sent to the server, the

configuration stays the same.

To try these different configurations, Stork stork provides the following features:

1. Initializing the distributed system, allowing Actors to be spawned.
2. De-initializing the distributed system, removing all Actors, and killing all processes related to Stork.
3. Delivering an Actor class to devices, allowing Actors to be spawned on devices. There are two variants: `deliverActor` and `deliverOrActor`.
4. Fetching the references of Actors that have been spawned across the distributed system. There are three variants: `getActorAddress`, `getActorClassAddresses` and `getActorHostAddresses`.
5. sending or broadcasting messages to the spawned Actors.

3.1 Initializing the distributed system

Before creation of Actors is possible the distributed system has to be initialized. More concretely, Stork has to be present on all the devices to be able to create Actors remotely. This is because each device must have an ActorSystem running in the background to be able to create Actors. Furthermore Stork has to manage the creation and bookkeeping of Actors, and it cannot do so if the ActorSystems have not spun up before creating the Actors. To initialize the distributed system, the host names of all devices must be known and a leader of the system must be known.

Stork starts by spinning up an ActorSystem on the leader device which then connects with the rest of the system from the leader device to spin up ActorSystems on the rest of the devices. Furthermore, the bookkeeping and creation of Actors is done from the leader device. This is because the leader device is the only device that has access to the rest of the remote devices upon initialization. As an example, using the system depicted in Figure 3 and Figure 4, the initialization of the distributed system is shown in Listing 2.

```
1 import Stork
2     devices = ["server@vub.ac.be","device1", "device2", "device3"]
3     devices = ["server@vub.ac.be","device1", "device2", "device3"]
4     # Then we call the distributeSystem method, with the leader
    device being the first host name in the list of devices.
5 devices = ["server@vub.ac.be","device1", "device2", "device3"]
6     # Then we call the distributeSystem method, with the leader
    device being the first host name in the list of devices.
7 Stork.distributeSystem(leader=devices[0], convention=devices)
```

Listing 2: Initializing the distributed system

When a user wants to assign certain roles to devices, Stork provides a way to do this during initialization of the distributed system. This is useful and helps the user spawn Actors on

devices that have common responsibilities. More concretely, if two or more devices should have the same Actor, such as compute the average as in our example in Figure 4, the user can declare the devices in a dictionary with the keys as the host names and a list of roles that the device should have. This is shown in Listing 3.

```
1 import Stork
2 devices = {
3     "server@vub.ac.be" : ["server", "database"],
4     "device1" : ["generator", "sender"],
5     "device2" : ["generator", "sender"],
6     "device3" : ["generator", "sender"]
7 }
8 Stork.distributeSystem(leader=devices.keys()[0], convention=
    devices)
```

Listing 3: Initializing the distributed system with devices have common roles

In Listing 3 the devices 1, 2 and 3 share the generator and sender roles, as they each have an instance of the Generator and Sender Actors in the configuration of Figure 3. Note that the host name is automatically added to the list of roles that each device has.

3.2 De-initializing the distributed system

Whenever a user wants to stop the distributed system, or if the user wants to change the configuration of the distributed system. For example when going from the configuration of Figure 3 to the configuration of Figure 4, the distributed system must be de-initialized and re-initialized. To de-initialize the distributed system, Stork provides a method called `undistributeSystem`. This method kills all Stork processes (including ActorSystems) on the devices. Like with the initialization method, it expects a list of the host names of the devices that are part of the distributed system and the leader of the distributed system. Again because it creates a connection from the leader device to the rest of the devices to kill the processes there first, and ends by killing the process on the leader device. Usage is shown in Listing 4.

```
1 import Stork
2 devices = [ "server@vub.ac.be" ,"device1" , "device2" , "device3"]
3 Stork.undistributeSystem(leader=devices[0], convention=devices)
```

Listing 4: De-initializing the distributed system

3.3 Delivering an Actor class to the devices

Now that the methods to initialize and de-initialize the distributed system have been discussed, we can discuss how to distribute the Actors. When deploying the distributed system, or more concretely, creating the Actors and connecting them, the user must be able to spawn Actors on different devices, from the cloud. Otherwise the user would have to manually create the Actors on the devices and connect them. Which, again is not scalable. For this purpose Stork provides the following two methods: `deliverActor` and `deliverOrActor`. However, these methods must be called within an Actor. This is because Stork uses the Actors to communicate internally, that an Actor must be spawned on a certain device. This is a limitation of Thespian, however this is further discussed in subsection 4.2.. First we discuss `deliverActor` and `deliverOrActor`.

In Listing 3 we discussed using roles to assign common responsibilities to devices. This will be helpful when creating Actors because it uses the roles to determine on which devices the Actor should be spawned. As a limited example, using the system depicted in Figure 3, we can create the GeneratorActor using the common "generator" role. However, since we are creating these Actors from the cloud, and the method can only be called from within an Actor, we need to create another Actor that will be responsible for spawning the Generator Actors on the devices. For the following example, the responsibility of spawning the Generator Actors will be given to the GeneratorSpawner Actor.

Note that all Actors still need to implement the `receiveMessage` method, as discussed in subsection 2.2. even if it is empty. This is because Thespian requires it. `deliverActor` and `deliverOrActor` takes the actor calling the method as a first argument as well as the class of actor that must be created and a list of roles. Here is where `deliverActor` and `deliverOrActor` differ. `deliverActor` checks if a device has all the roles in the list, while `deliverOrActor` checks if the device has at least one of the roles in the list.

```
1  import Stork
2  from thespian.actors import Actor
3
4  class GeneratorActor(Actor):
5
6      def receiveMessage(self, message, sender):
7          # generate data and send it to the sender actor
8
9  class GeneratorSpawner(Actor):
10
11     def receiveMessage(self, message, sender):
12         if message == "spawn":
13             Stork.deliverActor(self, GeneratorActor, ["generator"
       ])
```

Listing 5: Delivering Actors

Note that the delivery method is triggered when the spawner receives the "spawn" message. This is necessary because of another limitation of thespian, where methods making use of the asynchronous message passing cannot be called during initialization of the actor. Only after,

which means that a user has to create the spawner and send "spawn" to it after it is initialized.

The example in Listing 5 will create Generator Actors on all devices that have the "generator" role. More concretely, there will be Generator Actors on devices 1, 2 and 3 but not the cloud. If the example used
`Stork.deliverOrActor(self, GeneratorActor, ["device1", "device2", "device3"])`
instead, the result would have been the same. This is because each device has at least one of the roles in the list, and as previously mentioned, the host names are automatically added to the roles of the device.

How does a user of Stork interact with the created Actors? Whenever the delivery methods are called, Stork creates references to the created Actors and sends these references back to the actor that called the method. However, these references will be wrapped in a class called `DeliveredActors` which not only contains the references of the created Actors but also other metadata, such as the host name of where the actor was created and the class of the actor. Since this is received as a message, the actor that called the delivery method has to handle the message in `receiveMessage`. However, prior to showing an example, we discuss what can be done with the `DeliveredActors` class, as it does provide some useful methods to interact with the created Actors.

Whenever a user wants to send a message to the created Actors, there are a few options of doing so. A message can be sent to a single actor, depending on its location, or more concretely host name, and class. For this the method `emit` exists. However, if a message is meant to be sent to multiple Actors, broadcasting a message is a better solution. Depending on if a message has to be broadcasted to a specific device or to all the Actors, `broadcast` or `broadcastHost` can be used, respectively. If a reference to an actor needs to be saved for later use, or if the previous methods are not sufficient, it is best to unpack the class through `unpackActors`, which returns a dictionary of the references, including the metadata. The structure is shown below in Listing 6

1. `emit(self: Actor, host: str, Actorclass: Actor, message: Any)`: This method sends a message to the Actor that was spawned on the device with the given host name.
2. `broadcastHost(self: Actor, host: str, message: Any)`: This method sends a message to all the Actors that were spawned on the device with the given host name.
3. `broadcast(self: Actor, message: Any)`: This method sends a message to all the Actors that were spawned.
4. `unpackActors()`: This method returns a dictionary with the host names of the devices as keys and a list as value. The list as value contains dictionaries of the spawned Actor class as keys and the reference as value. This way we can address the Actor not only by the device that it is on but also by the class. Using the example from listing 5, unpacking the class sent to the caller of the delivery method would look as follows:

```
{
"device1": [{ GeneratorActor: reference_to_GeneratorActor }],
"device2": [{ GeneratorActor: reference_to_GeneratorActor }],
"device3": [{ GeneratorActor: reference_to_GeneratorActor }]
}
```

```
6
```

Listing 6: lst:Unpacking structure

As with the `deliverActor` method, the `DeliveredActors` class must be used within an Actor. This is because the `DeliveredActors` class uses the Actor to send messages to other Actors. Extending the example from Listing 5 the next example will show a sample usage of the `DeliveredActors` class, and broadcast to all GeneratorActors to create a SenderActor on their device. The message that triggers the creation of the Sender Actor is wrapped in a class that contains the hostname of the device where the GeneratorActor should create the SenderActor

```
1  import Stork
2  from thespian.actors import Actor
3
4  class GeneratorActor(Actor):
5
6      def receiveMessage(self, message, sender):
7          if isinstance(message, CreateSenderActor):
8              stork.deliverActor(self, SenderActor, [message.device
       ])
9
10 class SenderActor(Actor):
11
12     def receiveMessage(self, message, sender):
13         # forward message to the cloud
14
15 class SpawnerActor(Actor):
16
17     def receiveMessage(self, message, sender):
18         if message == "spawn":
19             Stork.deliverActor(self, ReceiverActor, ["server"])
20             Stork.deliverActor(self, GeneratorActor, ["generator"
       ])
21         elif isinstance(message, Stork.DeliveredActors):
22             unpacked = message.unpackActors()
23             for device, v in unpacked:
24                 message.broadcastHost(self, device,
       CreateSenderActor(device))
```

Listing 7: Receiving the references of the spawned Actors and broadcasting CreateSendActor

Noticeably, in the example the generators are created first, then the senders. But what if the ReceiverActor from Figure 3 is created before the generators and the senders are created? In this case Stork provides methods to fetch the address of an existing actor. This can be done with `Stork.getActorAddress` which expects the Actor calling it as first argument, the host name of the device we want to get the reference of, and the class of the Actor that we want to get a reference of. Essentially the same as the `DeliveredActors.emit` method but without the message parameter.

The second method is `Stork.getActorClassAddresses` which works similarly to `DeliveredActors.broadcastHost`. However instead of the hostname it expects a class. More concretely, it expects the Actor calling it as the first argument and the Actor class of which we want to get addresses as the second argument. `Stork.getActorClassAddresses` gets all the addresses of the Actors of a specific class.

Thirdly there is `Stork.getActorHostAddresses` which works similarly to `DeliveredActors.broadcastHost`, but without the message parameter. It expects the Actor calling it as the first argument and the host name of the device of which we want to get the references of the Actors as the second argument. `Stork.getActorHostAddresses` gets all the addresses of the Actors on a specific device.

Just like in 5 the response has to be handled in the `receiveMessage` method. For these methods Stork provides another class, to differentiate it from the `DeliveredActors` class. This class is called `ActorAddresses` and has the same methods as `DeliveredActors`. Extending Listing 7 we have the following example usage, where the ReceiverActor is created before the generators and senders. This means that the sender has to get the reference of the receiver.

```
1    import Stork
2    from thespian.actors import Actor
3
4    class GeneratorActor(Actor):
5
6        def receiveMessage(self, message, sender):
7            if isinstance(message, CreateSenderActor):
8                stork.deliverActor(self, SenderActor, [message.
     device])
9            if isintance(message, DeliveredActors):
10               message.broadcast("fetch receiver")
11
12   class SenderActor(Actor):
13
14       def receiveMessage(self, message, sender):
15           if message = "fetch receiver":
16               stork.getActorAddress(self, "server@vub.ac.be",
     ReceiverActor)
17           if isinstance(message, ActorAddresses):
18               receiver = message
19           else:
20               # forward message to receiver
21
22   class ReceiverActor(Actor):
```

```
23
24          def receiveMessage(self, message, sender):
25              # forward message to averageActor
26
27
28      class SpawnerActor(Actor):
29
30          def receiveMessage(self, message, sender):
31              if message == "spawn":
32                  Stork.deliverActor(self, GeneratorActor, ["
    generator"])
33              elif isinstance(message, Stork.DeliveredActors):
34                  if isintance(next(iter(next(iter(refs.values()))
    [0].keys())), GeneratorActor):
35                      # checks if the DeliveredActors is a result
    from generating the generators
36                      for device, _ in message.unpackActors():
37                          message.broadcastHost(self, device,
    CreateSenderActor(device))
38
39
```

Listing 8: Fetching an actor address

Till now we have worked with the examples from the initial configuration as depicted in Figure 3. However to try the configuration from Figure 4 the user needs to add "average" to the list of roles of the devices 1, 2 and 3. Change the linkage by for example changing which actors create which other actors. More concretely, change the generator to create the AverageActors instead of the SenderActors, which also means that now the AverageActors need to create the SenderActors. Furthermore the receiver Actors should now forward the data to the SquareActors. After changing the linkage and the roles that each device has. It suffices to re-initialize the distributed system and spawn the actors again and the new configuration is ready to be tested.

4 Implementing Stork

Before discussing the implementation of Stork there are a few things that must be discussed about Thespian. When creating Actors it must be done within an ActorSystem. Meaning that on every device, on which we want to create an Actor, an ActorSystem must be present. This is because the ActorSystem manages the Actors and their communication. Consequently this means that an active ActorSystem is required on each device before we can create an Actor. This can be done by running a small script that creates an ActorSystem, nothing more, since the ActorSystem will be running in the background. This means we need to SSH to all devices and run the script on each device. For a small amount of devices this probably will not be an issue if it has to be done once or twice. However, when the distributed system is large, or the deployment is repetitive, it will be very taxing to the deployer, since it is a lot of repetitive manual work.

4.1 Automating the deployment of ActorSystems

To automate this, we introduce Fabric, a python library that allows us to create connections with remote devices. There is still one issue though, when using Fabric to create a connection and running a command, it expects that the process returns. It does so because it expects the command to be a foreground process. However, when creating an ActorSystem, as this starts a background process, it never returns. This is circumvented by using "tmux" before executing the program to start the ActorSystem. This ensures that the process is running in the background and the command returns. This is also how the `Stork.distributeSystem` works, as it uses Fabric to create the ActorSystems on the devices. The `Stork.undistributeSystem` method works in a similar way, as it uses Fabric to stop the ActorSystems on the devices. This is done by running a script that stops the ActorSystem on the devices. This script is also run in a "tmux" session, to ensure that the command returns. Essentially, both Stork methods call methods implemented for Fabric that create a SSH connection to the devices and run a command, that runs the initialization or de-initialization script. Hence the importance that this is done from the leader device.

4.2 Creating and delivering Actors

The `Stork.deliverActor` and `Stork.deliverOrActor` methods work in a different manner, though they do make use of the ActorSystems that have been created by the `Stork.distributeSystem` method. In Thespian there are two ways to create Actors. One through the ActorSystem and one through an Actor, meaning that Actors can create other Actors. This begs the question of how we can create Actors on remote devices? More specifically from the leader device, since we do not want to have to SSH to the remote devices to create Actors there. While Thespian does support a mechanism to restrict the creation of certain Actor classes to certain devices, there is no way to guarantee that the Actor is created on multiple different devices. This is because Thespian will always create an Actor on the first ActorSystem it can. Consequently, most of the times it will spawn the Actor on the same ActorSystem. Creating a remote Actor from the local ActorSystem is no good either, as it will either create it locally or, when using the Thespian mechanism, it will throw an error that it cannot create that Actor. This is because the mechanism only works when creating Actors from Actor, however, as mentioned before it does not work when the user wants to create an instance of an Actor on multiple devices.

To work around this, we implement an administrative Actor that has to run on the leader device. As there is only one instance of the administrative Actor, we can use Thespians restriction mechanism to our advantage. This means that we can restrict the creation of the administrative Actor to the leader device. When running the scripts to initialize the ActorSystems on all devices, the script now also creates a reference to this administrative Actor. Meaning that all remote ActorSystems are now able to communicate with an administrative Actor on the leader device. However, this still does not guarantee that the references are all pointing to the same administrative Actor. Meaning that if two remote ActorSystem ask the administrative Actor for a reference to a certain Actor, they will likely not get the same response, because they will not be asking the same Actor. This is an issue. To work around this, Thespian provides another feature called the "global Actor", which essentially gives the Actor a name, and thus now all remote ActorSystems referencing the

administrative Actor will be referencing the same Actor. To prevent users of Stork having a name clash with the administrative Actor, the name is a generated hash.

Now that we have an administrative Actor that can be referenced by all remote ActorSystems, we can use this Actor to fetch addresses as well as create Actors on the leader device. However, We cannot yet create Actors on remote ActorSystems. To achieve this, we create another administrative Actor, called a manager Actor, the manager Actor will be created on all remote ActorSystems. The only thing that the manager Actor does is, register itself with the administrative Actor, create Actors on the device it resides in, followed by returning the reference of the Actors it created to an aggregator Actor. The aggregator Actor is created by the administrative Actor to collect the references of the created Actors, as per the aggregator pattern [10]. The manager Actor registers itself to the administrative Actor by sending its reference to the administrative Actor, as well as the hostname of the device it resides in. This way the administrative Actor can ask the manager Actor to create an Actor on the device it resides in. The manager Actor then creates the Actor and forwards the reference to the aggregator Actor. When all the references are collected by the aggregator Actor, it sends the result back to the administrative Actor as well as to the Actor that requested the creation of the Actors. This way other Actors can ask the administrative Actor for the addresses of the Actors that were created on the remote devices, and the Actor that originally requested the creation of the Actors has the reference of the Actors that were created on the remote devices. To illustrate this we have the following Actors: Ping and Pong, where the Ping Actor spawns a Pong Actor and sends it "ping", to which the pong actor replies with "pong".

When a user wants to create a Pong actor on the device with the capability "sensor" from the Ping Actor which is on a remote device. The Ping Actor sends a request to the administrative Actor on the leader device to create a Pong Actor on all the devices with the "sensor" capability.

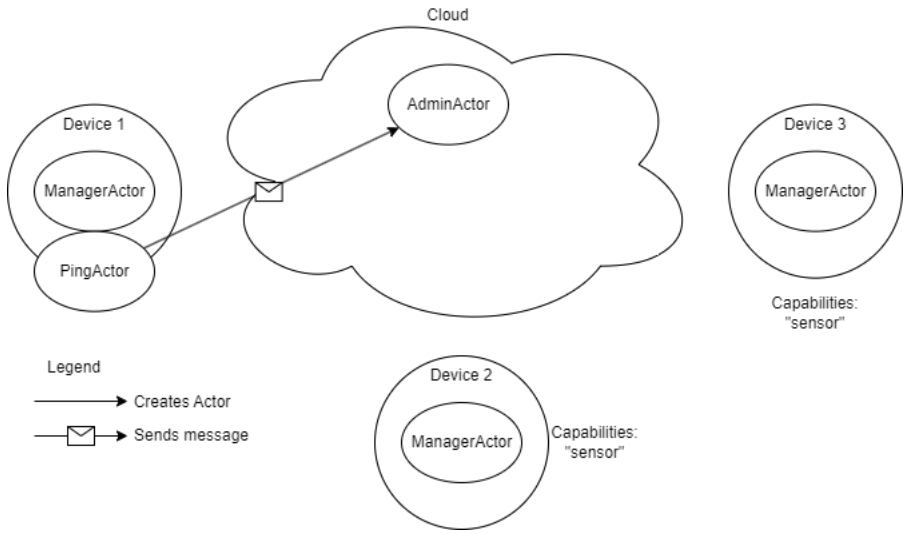

Figure 5: Ping sends a request to spawn Pong Actors on all devices with the "sensor" capability

When the administrative Actor processes this request, it collects the references of the registered manager Actors that that are on remote devices with the requested capability. It counts the amount of manager Actors that it should create the Pong Actor and creates an aggregator Actor. The administrative Actor sends to the aggregator Actor the reference of the Ping Actor, that requested the creation of the Pong Actors, as well as the expected amount of results. Furthermore the administrative Actor forwards the creation request to the Manager Actors with the reference of the Aggregator Actor.

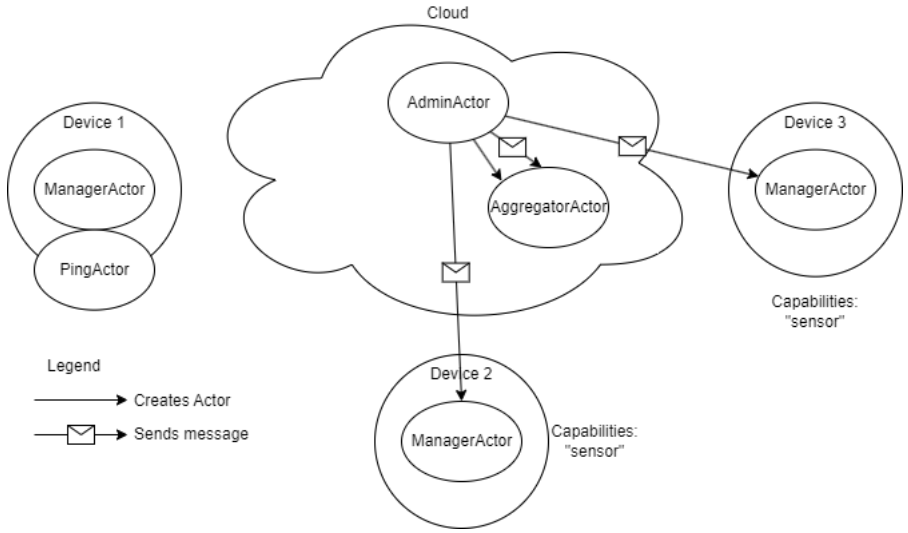

Figure 6: The creation request is forwarded and an aggregator Actor is created to collect the references of the created Actors

Upon receiving the creation request from the administrative Actor, the Manager Actor creates the Pong Actor and sends the reference of the Pong Actor to the aggregator Actor.

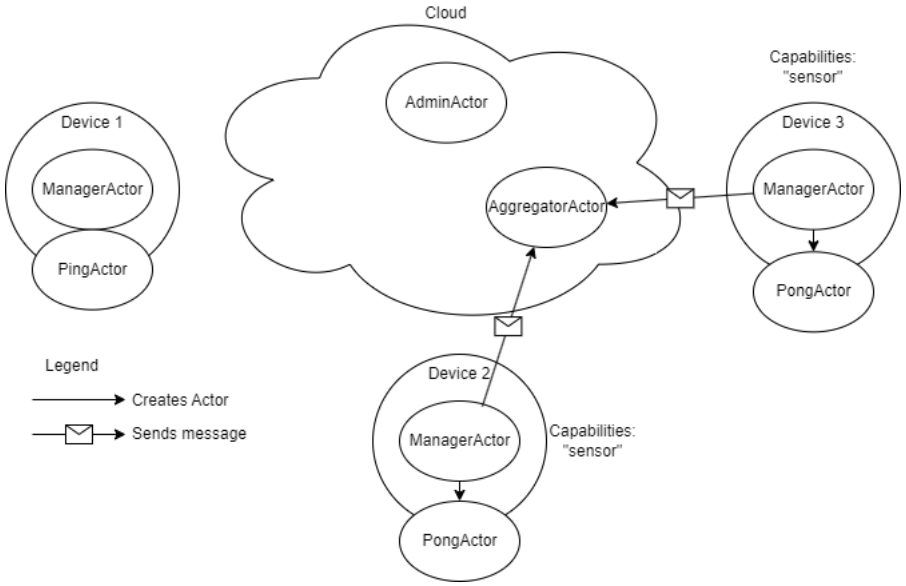

Figure 7: The Manager Actor creates the Pong Actor and sends the reference to the aggregator Actor

When the aggregator collected all the expected results, it sends the collected references to both the Ping Actor as well as the administrative Actor. The administrative Actor will update its collection of Actor references while the Ping actor will only get the created references.

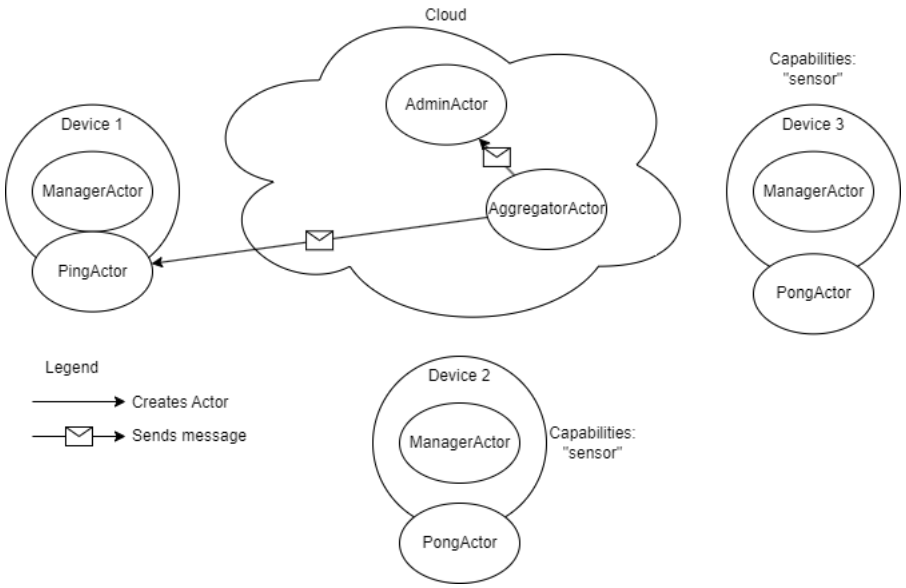

Figure 8: The aggregator Actor sends the collected references to the Ping Actor and the administrative Actor

This concludes the creation of the Pong Actors on the remote devices, however in the example from listing 7 the Ping Actor also sends a message to the Pong Actors. While this can be done with looping over the references and sending the message, it helps to have a method that can send a message to all the created Actors, or a subset of the created Actors. For this purpose the methods from the `DeliveredActors` are implemented. Schematically it goes as follows:

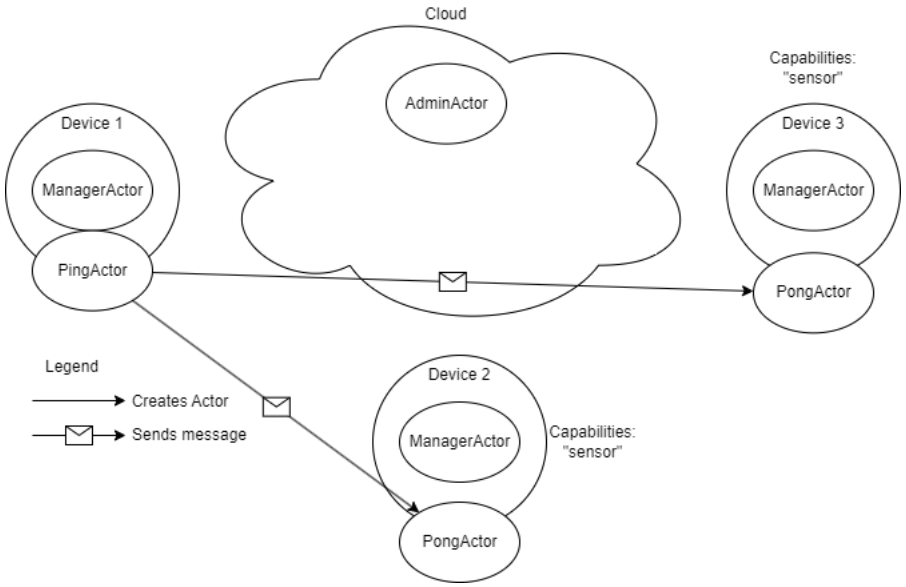

Figure 9: Ping broadcasts "ping" to all the created Pong Actors

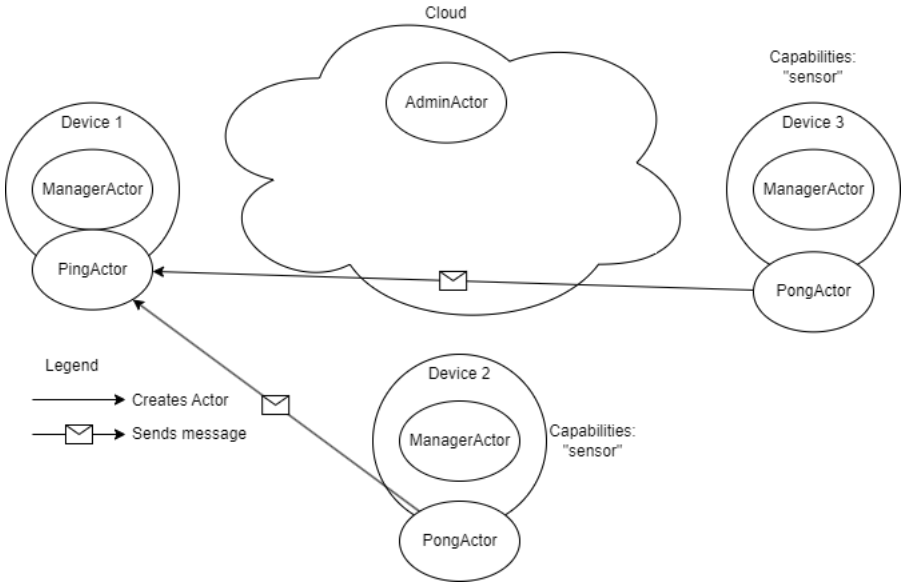

Figure 10: The Pong Actors receive the message and send "pong" back to the Ping Actor

4.3 Getting Actor references

The `Stork.getActorAddress`, `Stork.getActorClassAddresses` and `Stork.getActorHostAddresses` methods work in a similar manner. However the requests do not go further than the administrative Actor, as depicted in figure 5. The administrative Actor will collect the results from its collection and return the results to the Actor that requested the references. Like with the `DeliveredActors` class, it could prove to be useful if we can interact with the Actors directly, instead of the user having to implement a method themselves. This is why the `ActorAddresses` class has the same methods as the `DeliveredActors` class.

5 Conclusion

At the beginning of this bachelor thesis, we wanted to be able to try out different configurations of distributed systems, without much manual intervention. We wanted to do this to see which configuration was the most energy efficient. To achieve this, we introduced Stork, a distributed computing deployment tool, written in Python that makes it possible to deploy distributed systems and removes the overhead of having to SSH to each device to initialize the distributed system manually. It also provides features to create Actors on remote devices, which is necessary to try out a configuration of the distributed system. To test out another configuration of the distributed system, or more concretely moving Actors around. It suffices for the user to change the flow of data, call the `Stork.undistributeSystem` method, change the configuration of the capabilities that each device has, and call `Stork.distributeSystem` again, before creating the Actors.

Essentially it boils down to changing the flow of data and changing some strings in a dictionary. This is a lot less work than having to SSH to each device manually, start an ActorSystem on each device and then return to the leader device. Especially when the user will still be required to create the Actors on the leader device, SSH to the remote devices again to create the Actors there and then return to the leader device to try out a single configuration. Depending on the amount of devices, configurations and the amount of times the user wants to try out a configuration, this can be very time consuming and not to mention energy inefficient, especially when a tool like Stork exists.

References

[1] H. Ritchie and P. Rosado, "Energy mix," *Our World in Data*, 2020. [Online]. Available: `https://ourworldindata.org/energy-mix`.

[2] M. Pesce. "Cloud computing's coming energy crisis." (2021), [Online]. Available: `https://spectrum.ieee.org/cloud-computings-coming-energy-crisis`.

[3] R. Ratheesh, M. S. Nair, M. Edwin, and N. S. R. Lakshmi, "Traffic based power consumption and node deployment in green lte-a cellular networks," *Ad Hoc Networks*, vol. 149, p. 103 248, 2023, ISSN: 1570-8705. DOI: `https://doi.org/10.1016/j.adhoc.2023.103248`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S1570870523001683`.

[4] L. J. and Z. Klarin, "How trend of increasing data volume affects the energy efficiency of 5g networks," *Sensors (Basel, Switzerland)*, 2021. DOI: https://doi.org/10.3390/s22010255.

[5] "Mpi: A message passing interface," in *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993, pp. 878–883. DOI: 10.1145/169627.169855.

[6] B. H. Tay and A. L. Ananda, "A survey of remote procedure calls," *SIGOPS Oper. Syst. Rev.*, vol. 24, no. 3, pp. 68–79, 1990, ISSN: 0163-5980. DOI: 10.1145/382244.382832. [Online]. Available: https://doi.org/10.1145/382244.382832.

[7] M. Herlihy, S. Rajsbaum, and M. Raynal, "Power and limits of distributed computing shared memory models," *Theoretical Computer Science*, vol. 509, pp. 3–24, 2013, Structural Information and Communication Complexity, ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2013.03.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397513001813.

[8] C. Hewitt, *Actor model of computation: Scalable robust information systems*, 2015. arXiv: 1008.1459 [cs.PL].

[9] P. Grossetete. "Iot and the network: What is the future?" (2020), [Online]. Available: https://blogs.cisco.com/networking/iot-and-the-network-what-is-the-future.

[10] Unknown, 2013. [Online]. Available: http://s-akara.blogspot.com/2013/08/an-aggregator-pattern-for-akka-actor.html.