VRIJE
UNIVERSITEIT
BRUSSEL

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

DELIVERING SYSTEMS WITH STORK

A distributed computing deployment tool

Gérard Lichtert

May 31, 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 Internet, which also plays a significant role in energy consumption. This is because the Internet 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. This gives rise to the question of how we can declare how certain data is processed, and where it is processed and on top of that change the where without much manual intervention. Moreover, we want this to be available in Python because it could be interesting for further data science[5] or AI applications, in which the language is known to be very popular. 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 initialize the deployment of a distributed system, change the configuration of where certain parts of the distributed system are deployed in a declarative way and reverse the deployment.

2 Background

Before we discuss Stork, we need to 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 we need to choose a distributed computing paradigm that allows us to deploy our system in a distributed way, as well as change where parts of our programs reside. Next, we need to look at the available technologies and libraries in Python and choose the one(s) that best fit our needs. Let us begin with the distributed computing paradigm.

2.1 Distributed computing paradigm

To start with distributed computing, we have to look for a suitable distributed computing paradigm that allows our system to be deployed from the cloud without much manual intervention, yet is able to do what we require it to do. There are several options, such as:

1. Message Passing Interface (MPI)[6]
2. Remote Procedure Call (RPC)[7]
3. Shared Memory Model[8]
4. The Actor Model[9]

While each paradigm has their strengths and drawbacks, we will be using the Actor Model. An Actor is a computational unit which encapsulates its state and behavior, interacting with each other through asynchronous message passing. Each Actor has a mailbox, in which it receives these messages and processes them sequentially. Because of the message-based communication between Actors, we can easily decouple components, which in turn makes it easier to debug and test our systems. This is beneficial because distributed systems can quickly become very complex.

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 which data gets processed without much manual intervention. Using an IoT system as a running example because IoT devices account for 50% of networked device[10]. It is possible to encapsulate the behavior of each sensor in an Actor, 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.

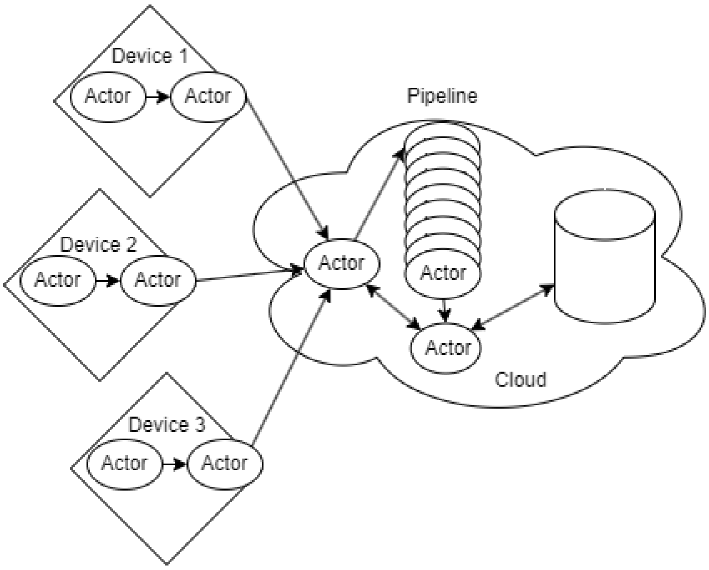

Figure 1: An example of an IoT system using the Actor Model

Conceptually we should be able to move some of the Actors that process data from the pipeline to the IoT devices and place them before the the Actor that sends the data to the Server, as depicted in figure 2. Achieving one of the possible configurations discussed in the introduction, without having to reprogram the entire system.

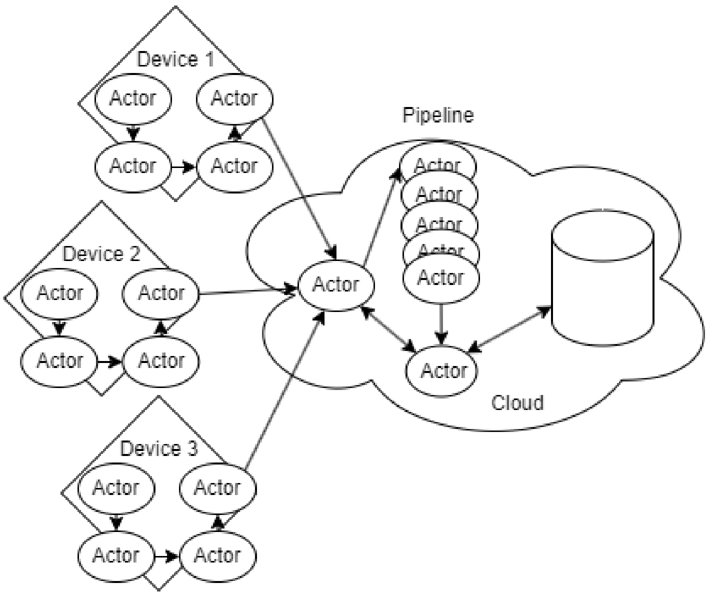

Figure 2: An example of an IoT system where some of the Actors that process data are moved to the IoT devices

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 behavior 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.

2.2 Implementing Actors in Python

When using Stork to deploy distributed systems and trying out different configurations, we need to create Actors that can be deployed by Stork. However, before we can discuss how to use Stork, we need to discuss how to implement Actors in Python. Since Stork is built with Thespian[1], a Python Actor library, we will be using Thespian to implement our Actors. Consequently Stork only works with Thespian Actors. To create an Actor one must create a class that inherits 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 will call the `receiveMessage` method. An example of an Actor is given below

```python
from thespian.actors import Actor

class HelloWorld(Actor):

    def receiveMessage(self, message, sender):
        if message == "are you there?":
            self.send(sender, "Hello World! back")
```

Listing 1: Actor example

3 Stork

As mentioned in the introduction, 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. As perviously mentioned there are some implications when using Stork, as it is built on Thespian, a Python Actor library. This means that Stork only works with Thespian Actors. Stork provides the following features:

1. Initializing the distributed system, allowing Actors to be spawned on the declared devices.
2. De-initializing the distributed system, removing all Actors from the declared devices.
3. Delivering an Actor class to the devices, allowing the Actor to be spawned on the device. This comes in two variants.

[1] https://thespianpy.com/doc/

3.1 Initializing the distributed system

When the user of Stork wants to initialize the distributed system, the user has to declare which devices take part of the distributed system. More importantly, Stork needs to know from which of these devices the distributed system should be initialized. This is because Thespian requires a leader device, to which the other devices register themselves to. Stork in turn uses this leader device to keep track of the registered, or remote devices, and spawn Actors on the remote devices and the leader device. It is expected that the user uses this method prior to deploying the user defined Actors. Otherwise the user defined Actors cannot be spawned.

```python
import Stork

if __name__ == "__main__":
    # First we declare a list of the host names of the devices that have to be part of our distributed system
    devices = ["server@vub.ac.be","device1", "device2", "device3"]
    # Then we call the distributeSystem method, with the leader device being the first host name in the list of devices.
    Stork.distributeSystem(leader=devices[0], convention=devices)
```

Listing 2: Initializing the distributed system

It is expected that the user calls this method on the leader device, because it is expected that the leader device has access to the other devices.

Alternatively, a user of Stork can instead of declaring the devices in a list, declare them in a dictionary with the keys as the host names and a list of properties as the value. This way the user can declare the capabilities of the devices, which can be used to spawn Actors on the devices that have the requested capabilities.

```python
import Stork

if __name__ == "__main__":
    # Just like before we declare our devices, but this time in a dictionary with the host names as keys and the capabilities as values.
    devices = {
        "server@vub.ac.be" : ["server", "database"],
        "device1" : ["sensor"],
        "device2" : ["sensor"],
        "device3" : ["sensor"]
    }
    Stork.distributeSystem(leader=devices.keys()[0], convention=devices)
```

Listing 3: Initializing the distributed system with capabilities

Again, it is expected that the user calls this method on the leader device, because it is expected that the leader device has access to the other devices.

3.2 De-initializing the distributed system

De-initializing the distributed is the opposite of initializing the distributed system. This is necessary because when the user is done with the distributed system, all processes must be stopped. Since all processes of the Actors are running on the background on the devices declared when calling `distributeSystem`, Stork creates a connection with the devices and runs a script to end all Actor related processes. Just like `distributeSystem`, it is expected that the user calls this method on the leader device.

```python
import Stork

if __name__ == "__main__":
    Stork.undistributeSystem()
```

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. Whenever the user wants to spawn an Actor on a device, the user can use two different methods to spawn the given Actor class. Stork provides two methods for this: `deliverActor` and `deliverOrActor`. However, these methods must be called within an Actor. This is because Stork uses the Actor to communicate internally, that an Actor must be spawned on a certain device.

The `deliverActor` method takes the Actor calling the method as an argument and a list of capabilities as a second argument. The list of capabilities is used to determine on which devices the Actor should be spawned. Taking a look at the example from listing 3, if we want to spawn an actor on all the devices with the capability "sensor", we can use `Stork.deliverActor(self, ActorClass, ["sensor"])`. However, if we only want to spawn the Actor on one of the sensors, we can use `Stork.deliverActor(self, ActorClass, ["device1"])`, or `Stork.deliverActor(self, ActorClass, ["sensor", "device1"])`. Regardless, `deliverActor` will spawn the Actor on all devices that have the all requested capabilities.

```python
import Stork
from thespian.actors import Actor

class Pong(Actor):

    def receiveMessage(self, message, sender):
        self.send(sender, "pong")

class Ping(Actor):

    def receiveMessage(self, message, sender):
        if message == "spawn":
            Stork.deliverActor(self, Pong, ["sensor"])
```

Listing 5: Delivering an Actor

Now that Stork spawned Actors on all the devices that have the “sensor” capability, how does a user of Stork interact with them? As mentioned in subsection 2.1., Actors communicate through asynchronous messaging. In Thespian and therefore Stork, this means that when a message is sent from an Actor to another Actor, it will never receive a return value. `deliverActor` and `deliverOrActor` are no exceptions to this rule, since they work in a similar manner. However, when an Actor sends a message to another Actor it can send a message back. Consequently, the user of Stork needs to extend the `receiveMessage` method to handle the message that Stork sends to the Actor calling the delivery methods. This message contains the references to the Actors that have been spawned, as well as the hostname of the device where the Actors were spawned. Once the Ping Actor has the references of the Pong Actors it can interact with the Pong Actors in a similar manner as it would with other thespian Actors.

For this purpose, Stork also provides a class to easily encapsulate the message that is sent to the Actor that called the delivery methods. This class is called `DeliveredActors`. The `DeliveredActors` class understands the following methods:

1. `unpackActors()`: This method returns a dictionary with the host names of the devices as keys and another dictionary as value. The dictionary as value has 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, the dictionary would look like this:

```
{
    "device1": { Pong: reference_to_Pong },
    "device2": { Pong: reference_to_Pong },
    "device3": { Pong: reference_to_Pong }
}
```

2. `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.

3. `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.

4. `broadcast(self: Actor, message: Any)`: This method sends a message to all the Actors that were spawned.

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.

```python
import Stork
from thespian.actors import Actor

class Pong(Actor):

    def receiveMessage(self, message, sender):
        self.send(sender, "pong")

class Ping(Actor):

    def receiveMessage(self, message, sender):
        if message == "spawn":
            Stork.deliverActor(self, Pong, ["sensor"])
        elif isinstance(message, Stork.DeliveredActors):
            message.broadcast("ping")
```

Listing 6: Receiving the references of the spawned Actors and broadcasting "ping"

The `deliverOrActor` method works in a similar way as the `deliverActor` method. The difference lies in the way the capabilities are checked. With `deliverActor` the device must have all specified capabilities. With `deliverOrActor` it suffices that the device has one of the required capabilities. Other than that it works in the same way as `deliverActor`. It will also send a message to the Actor that called the method, containing the references to the spawned Actors in the `DeliveredActors` class.

Sometimes however, an Actor is already created and we need to get a reference of said Actor to communicate with it. Stork provides three methods for this. The first one only gets the address of a single Actor class of a specific device. 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 and lastly 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`.

4 Implementing Stork

Now that we have discussed the features that Stork provides, we can discuss how it is implemented. As previously mentioned, Stork is built on Thespian, a Python Actor library. This means that most features that Stork provides are built on Thespian. But, perhaps it is worth discussing why Thespian, and not another Python Actor library, as there is another one called Pykka.

Pykka is a Python implementation of the Actor Model. It introduces some simple rules to control the sharing of state and cooperation between execution units, which makes it easier to build concurrent applications[2]. However, Pykka does not support distributed communication out of the box, like Thespian does. This is important since we we want to build a distributed system. Consequently, we choose Thespian over Pykka.

Before discussing the implementation of Stork there are a few things that must be discusses 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 we need to start an ActorSystem 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

For this we require to implement a program that automatically SSH's to the all devices of the distributed system and runs the script to create an ActorSystem. However, we need to keep the order in mind. Thespian calls these distributed systems "a convention". Each convention must have a convention leader, to which the other ActorSystems register themselves to. To automate this, we need a library or technology that serves as infrastructure as code (IaC). The options are:

1. Ansible
2. Fabric
3. Chef

[2]https://pykka.readthedocs.io/en/stable/

4. Puppet
5. Terraform

Most of these IaC tools are quite heavyweight and would require us to learn another language. Fabric however is a Python library that allows the user to create SSH connections and run commands on remote devices. This is perfect for our use case, as we need to use Python to create the script that creates the ActorSystem, and thus stay in the same language. Consequently, we can use Fabric to automate the creation of the ActorSystems on the devices.

There is still one issue though, when using Fabric to create a SSH 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 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.

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 asks 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 [11]. 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. As an example we can use the Ping and Pong Actor classes from listing 6

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.

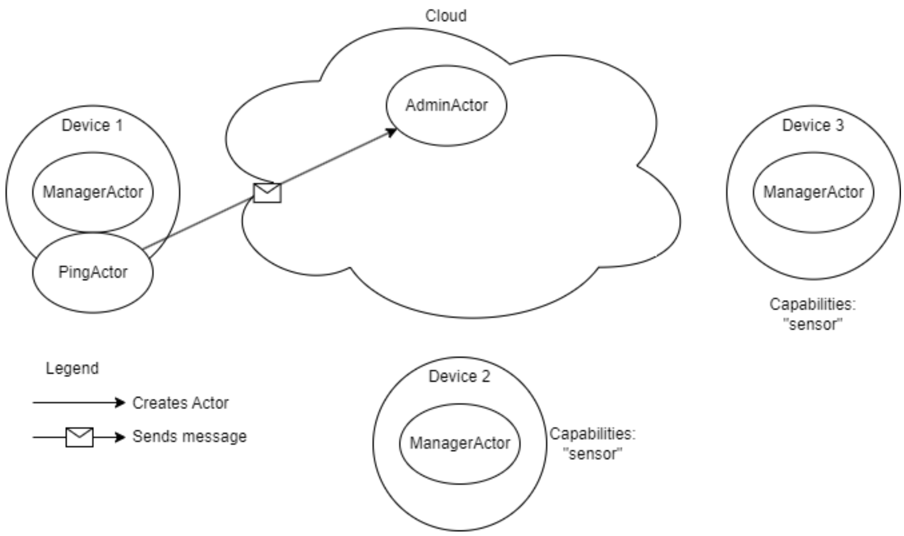

Figure 3: Ping sends a request to spawn Pong Actors on all devices with the "sensor" capability

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.

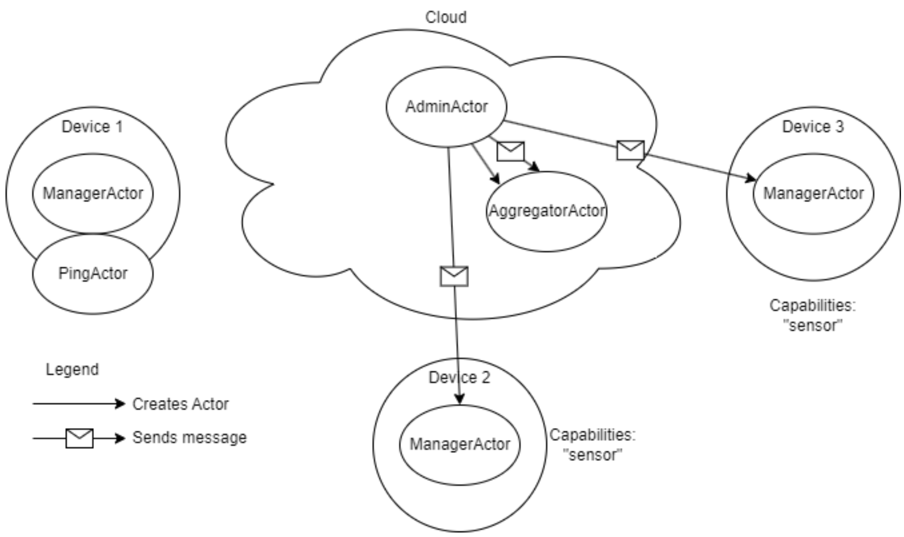

Figure 4: The creation request is forwarded and an aggregator Actor is created to collect the references of the created Actors

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.

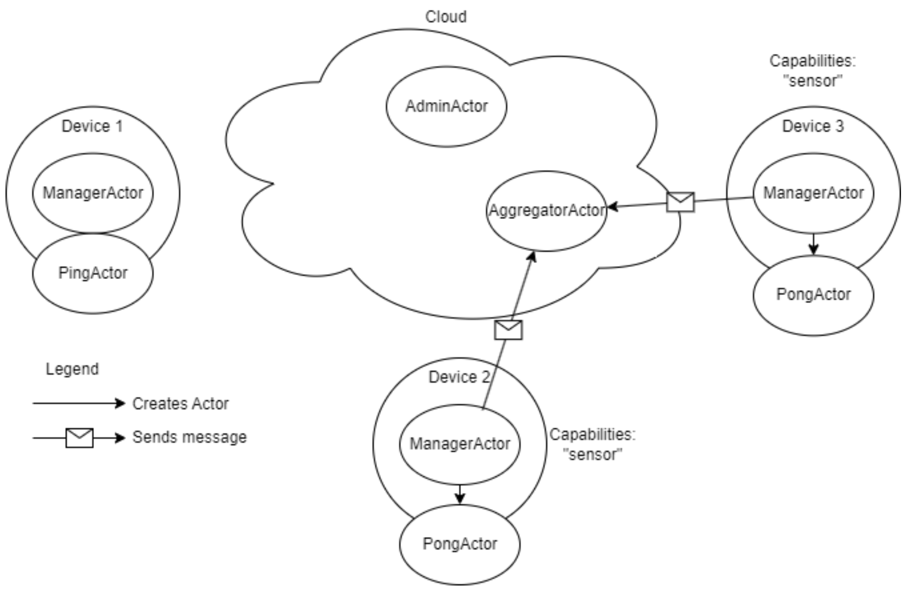

Figure 5: The Manager Actor creates the Pong Actor and sends the reference to the aggregator Actor

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.

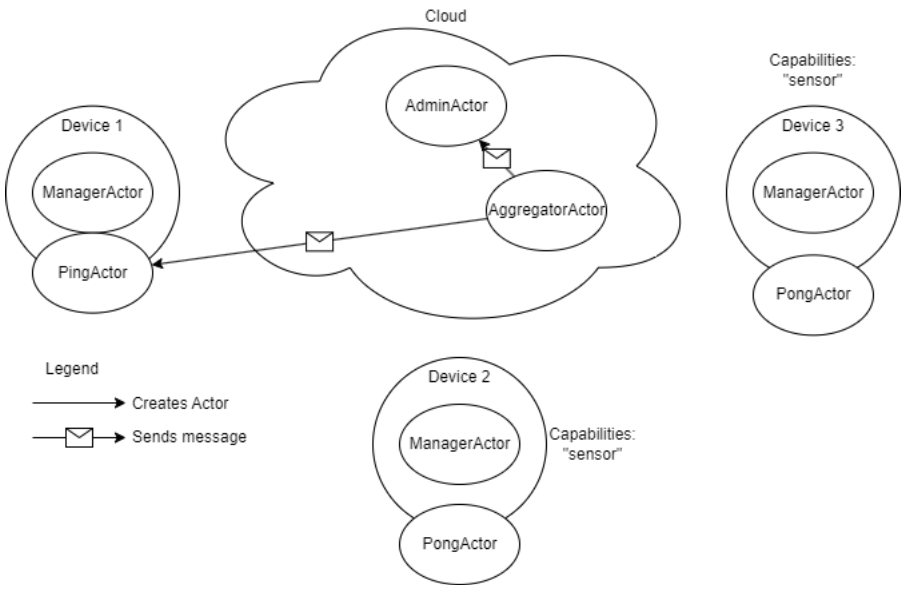

Figure 6: The aggregator Actor sends the collected references to the Ping Actor and the administrative Actor

This concludes the creation of the Pong Actors on the remote devices, however in the example from listing 6 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:

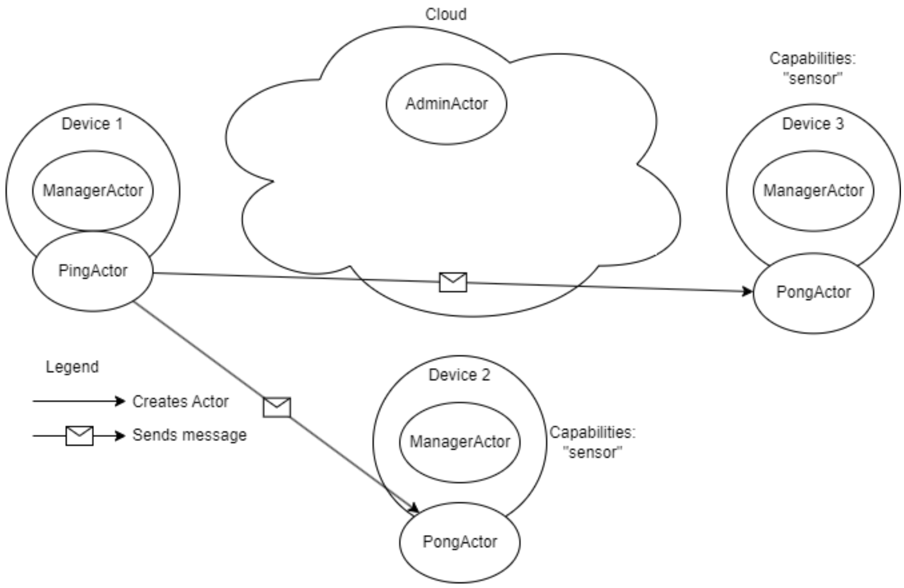

Figure 7: Ping broadcasts "ping" to all the created Pong Actors

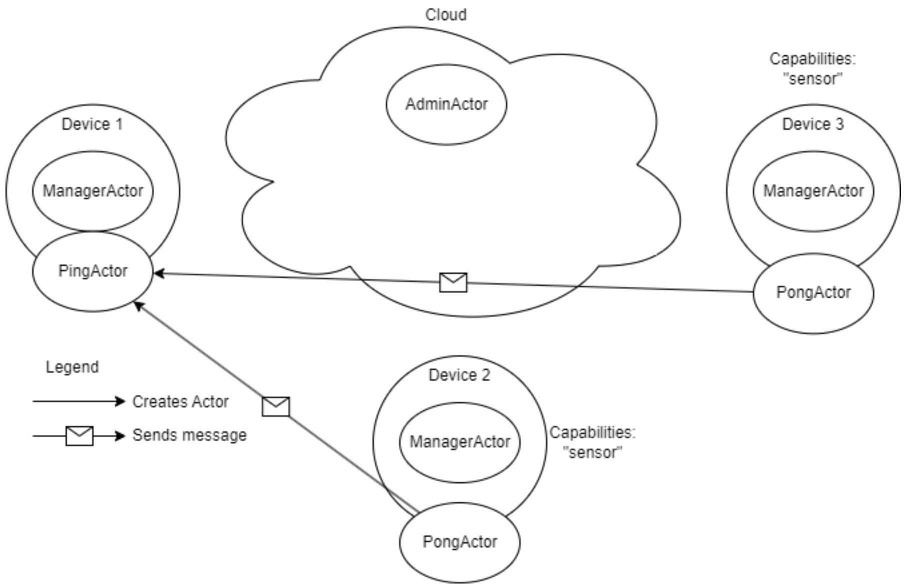

Figure 8: The Pong Actors receive the message and send "pong" back to the Ping Actor

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 3. 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

References

[1] H. Ritchie and P. Rosado, "Energy mix," *Our World in Data*, 2020, 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] S. Gupta. "11 best programming languages for data science in 2024." (2022), [Online]. Available: `https://www.springboard.com/blog/data-science/best-language-beginner-data-scientists-learn/`.

[6] "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`.

[7] 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`.

[8] 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`.

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

[10] 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.

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