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 24, 2024

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

Sciences and bioengineering 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. According to Ritchie and Rosado, 2020, the majority of our electricity production still comes from non-renewable sources, such as coal, oil, and gas. 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, according to Pesce, 2021 cloud computing takes 1% of worldwide energy consumption. While this may not seem like much, it is still a significant amount of energy.

Computing like most electronics, require electricity to function, however when computers communicate with each other they also have to send data, or make requests through the Internet. This leads to a very high network usage, since in today's world, we are unequivocally connected with each other. Furthermore, the amount of devices using the internet keeps growing, which leads to even more network usage. Higher network usage also leads to higher energy consumption, which leads to a higher carbon footprint, as stated in Ratheesh et al., 2023. J. and Klarin, 2021 states that there is also an increase in energy consumption due to the infrastructure required for the increase in data volume. 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, if the data is smaller after preprocessing, the network load should be reduced, and consequently the energy consumption. However, for a certain set of devices 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 it could be optimal to send the data directly to the server. Sometimes the optimal configuration could be a combination of the two. This gives rise to the question of how we can easily declare where which data gets processed.

In this thesis we introduce a tool for this purpose: Stork. Stork is a distributed computing deployment tool that makes it possible to deploy distributed systems in a declarative way. It allows us to change the deployment configuration as well as declare where which data gets processed. It does so by delivering the correct parts of the program to the correct devices. Hence, the name Stork.

We will know that the tool is successful if we can use the tool as a library to deploy a distributed system without manual intervention. Furthermore, the tool should allow us to change the configuration of the deployment, and declare where which data gets processed. Lastly the data must correctly be processed on the declared devices.

2 Background

To start building Stork we need to look at the goals that our tool should achieve. 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 choose a programming language of implementation, while keeping several factors in mind, such as, popularity, the use case of the programming language and the available libraries or technologies that exist in the language for our chosen paradigm. After choosing our programming language, we need to look at the available technologies and libraries in the programming language and choose the one(s) that best fit our needs. It could happen that we need other libraries or technologies to support the ones we choose but, we will cross that bridge when we get there.

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) as described in "MPI: A message passing interface", 1993
2. Remote Procedure Call (RPC) as described in Tay and Ananda, 1990
3. Shared Memory Model as described in Herlihy et al., 2013
4. The Actor Model as described in Hewitt, 2015
5. Publish/Subscribe (Pub/Sub) as described in Lin et al., n.d.

While each paradigm has their strengths and drawbacks, we will be using the Actor Model. Primarily because its modularity. Each actor encapsulates its state and behavior, which makes it a prime candidate to encapsulate the behavior of the parts that process certain parts of data. This allows us to easily move part of the data processing pipeline from one device to another. Which helps with our goal of declaring where which data gets processed. Interestingly, it can also encapsulate the behavior of IoT devices, which according to Grossetete, 2020 will account for 50% of all networked devices by 2023. Seeing as IoT devices are becoming increasingly important, they should be a factor in our choice of distributed computing paradigm.

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. The

Pub/Sub model is also a good candidate, however, it is more suited for event-driven systems, rather than systems with direct communication.

Other than the modularity, we also choose it for its maintainability. Since each part of the system is encapsulated in an Actor, we can easily change the behavior of parts of the system, instead of having to change our entire system. Another reason is scalability. We are able to distribute actors across different systems and machines. Which makes it perfect for moving parts of the data processing. Fault-tolerance makes it easier to contain errors and within individual actors and not propagate this to the entire system. The asynchronous nature of the Actor Model allows us to sparely use resources as needed, which will presumably reduce the required energy of the system, helping with the energy optimization. Lastly, through 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.

2.2 Programming language of implementation

The next step is to choose a programming language that supports the Actor Model. However, before listing the programming languages that support the Actor model, we need to consider a few things first. We want our tool to be maintainable. This means that choosing a programming language that is popular and has a large community is important. Furthermore, when processing data, companies tend to analyze the data, and perform data science on it. This means that ideally we should choose a programming language that is also used in data science, so that users of Stork can stay in the same language. Lastly but more importantly, we should choose a programming language that has libraries or technologies that support the Actor Model. This is important because otherwise we have to make our own implementation of the Actor Model, which is not ideal, considering there are already several implementations available.

A quick google search yields the following programming languages that support the Actor Model:

1. Scala with its Akka library
2. Java with its Akka library
3. Erlang, where the Actor Model is built into the language
4. Elixir, where the Actor model is built into the language, and based on Erlang, Open telecom platform (OTP)
5. JavaScript with its js.Actor library
6. Python with its Pykka and Thespianpy libraries
7. C# with its Akka.NET library
8. Rust with its Actix library

In terms of popularity, according to "Stack Overflow Developer Survey", 2023, they are ranked as follows[1]:

1. JavaScript
2. Python
3. Java
4. C#
5. Rust
6. Scala
7. Elixir
8. Erlang

With the last three being used by less than 5% of the respondents, and the top two being used by roughly $49\% - 63\%$ of respondents, we can focus on the top two. To stay in the same programming language that is often used by data scientists, we choose Python. This because according to Gupta, 2022, Python is the most popular language for data science, and while JavaScript is also used for data science, it does not offer nearly as much data science libraries as Python does.

2.3 Technologies and libraries

As mentioned when choosing the programming language of implementation, Python offers two Actor Model libraries. Pykka and Thespianpy. Setting Pykka and thespian next to each other and comparing them yields the following results:

2.3.1 Pykka

Quoting the documentation of the library: "Pykka is a Python implementation of the actor model. The actor model introduces some simple rules to control the sharing of state and cooperation between execution units, which makes it easier to build concurrent applications." However, reading further down in the documentation we see that Pykka has some shortcomings, as it does not support linking actors, supervisors, or supervisor groups. It does not support communicating with actors running on other hosts, which is a requirement for our tool. Lastly, it does not come with a set of predefined message router, which would mean that we have to implement this ourselves. Notably it does have 1200 stars on Github, which comparing it to Thespianpy, is more.

[1]TypeScript is emitted since it uses JavaScript libraries

2.3.2 Thespianpy

The documentation states that: "Thespian is a Python library providing a framework for developing concurrent, distributed, fault tolerant applications. It is built on the Actor Model which allows applications to be written as a group of independently executing but cooperating "Actors" which communicate via messages. These Actors run within the Actor System provided by the Thespian library." Since all Actors run independently within the Actor System, we can create concurrent applications. Thespianpy also allows us to run Actors independently anywhere. This means that we can have multiple servers running Thespianpy and an Actor can be run in any of these systems. Thespianpy will handle the communication between the Actors and the management process of distributing the Actors across the system. Which is something we would have to implement ourselves if we were to user Pykka.

References

Grossetete, P. (2020). *Iot and the network: What is the future?* https://blogs.cisco.com/networking/iot-and-the-network-what-is-the-future

Gupta, S. (2022). *11 best programming languages for data science in 2024.* https://www.springboard.com/blog/data-science/best-language-beginner-data-scientists-learn/

Herlihy, M., Rajsbaum, S., & Raynal, M. (2013). Power and limits of distributed computing shared memory models [Structural Information and Communication Complexity]. *Theoretical Computer Science*, *509*, 3–24. https://doi.org/https://doi.org/10.1016/j.tcs.2013.03.002

Hewitt, C. (2015). Actor model of computation: Scalable robust information systems.

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

Lin, W.-T., Wolski, R., & Krintz, C. (n.d.). *A programmable and reliable publish/subscribe system for multi-tier iot.* https://sites.cs.ucsb.edu/~ckrintz/papers/canal21.pdf

Mpi: A message passing interface. (1993). *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 878–883. https://doi.org/10.1145/169627.169855

Pesce, M. (2021). *Cloud computing's coming energy crisis.* https://spectrum.ieee.org/cloud-computings-coming-energy-crisis

Ratheesh, R., Nair, M. S., Edwin, M., & Lakshmi, N. S. R. (2023). Traffic based power consumption and node deployment in green lte-a cellular networks. *Ad Hoc Networks*, *149*, 103248. https://doi.org/https://doi.org/10.1016/j.adhoc.2023.103248

Ritchie, H., & Rosado, P. (2020). Energy mix [https://ourworldindata.org/energy-mix]. *Our World in Data*.

Stack overflow developer survey. (2023). https://survey.stackoverflow.co/2023/

Tay, B. H., & Ananda, A. L. (1990). A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, *24*(3), 68–79. https://doi.org/10.1145/382244.382832