

# **Comparative Analysis of Fault Tolerance in Elixir and Other Distributed Languages**

**Nuno Ribeiro**

**A dissertation submitted in partial fulfillment of  
the requirements for the degree of Master of Science,  
Specialisation Area of Software Engineering**

**Advisor: Dr. Luís Nogueira**

Porto, November 24, 2024



# Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, November 24, 2024



# Contents

<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Context . . . . .	1
1.2 Problem . . . . .	1
1.3 Objectives . . . . .	1
1.4 Ethics . . . . .	1
1.5 Document structure . . . . .	1
<b>2 Background</b>	<b>3</b>
2.1 Distributed Systems . . . . .	3
2.1.1 Characteristics . . . . .	3
Transparency . . . . .	4
Reliability and availability . . . . .	4
Scalability . . . . .	4
Fault tolerance . . . . .	5
2.1.2 Communication . . . . .	5
Synchronous and asynchronous communication . . . . .	5
Communication models . . . . .	5
2.1.3 Challenges . . . . .	6
CAP theorem . . . . .	6
2.1.4 Evolution of distributed systems . . . . .	6
2.2 Fault Tolerance . . . . .	7
2.2.1 Fault Tolerance Taxonomy . . . . .	7
2.2.2 Strategies . . . . .	7
Retry . . . . .	8
Replication . . . . .	8
Check-pointing and Message Logging . . . . .	9
2.2.3 Evolution and the State-of-the-Art . . . . .	9
2.3 Distributed and Concurrency Programming . . . . .	9
2.3.1 Models and Paradigms . . . . .	10
Actor Model . . . . .	10
Communicating Sequential Processes (CSP) . . . . .	10
Microservices Architectures . . . . .	11
Comparison of Models and Paradigms . . . . .	11
2.3.2 Distributed and Concurrent Programming Languages . . . . .	11
Analyses and Justification . . . . .	11
<b>3 Literature Review</b>	<b>13</b>

3.1	Research Questions . . . . .	13
3.2	State of Art . . . . .	13
3.3	Conclusions . . . . .	13
<b>4</b>	<b>Planning</b>	<b>15</b>
4.1	WBS . . . . .	15
4.2	Gantt Diagram . . . . .	15
4.3	Risk Management . . . . .	15

# List of Figures





# List of Tables

2.1	Types of Faults and Their Descriptions . . . . .	8
2.2	Brief Description of Failure Types . . . . .	8



# **Chapter 1**

## **Introduction**

**1.1 Context**

**1.2 Problem**

**1.3 Objectives**

**1.4 Ethics**

**1.5 Document structure**



## Chapter 2

# Background

### 2.1 Distributed Systems

In the early days of computing, computers were large and expensive, operating as standalone machines without the ability to communicate with each other. As technology advanced, smaller and more affordable computers, such as smartphones and other devices, were developed, along with high-speed networking that allowed connectivity across a network [1]. These innovations made it possible to create systems distributed across nodes where tasks could be processed collectively to achieve a common goal [2]. Nodes in a distributed system may refer to physical devices or software processes[3].

To the end-user, distributed systems appear as a single, large virtual system, making the underlying logic transparent [4]. These systems achieve a shared objective by transmitting messages through various nodes and dividing computational tasks among them, increasing resilience and isolating business logic [3, 5, 6]. Distributed systems can present heterogeneity, such as differing clocks, memory, programming languages, operating systems, or geographical locations, all of which must be abstracted from the end-user [1, 6].

While decentralized and distributed systems share characteristics, they differ in node organization and governance. Distributed systems spread nodes across computers to improve reliability and scalability, distributing logic without centralization [1]. For example, email systems scale with user demand without needing consensus. In contrast, decentralized systems, like blockchain, involve independent nodes with shared authority, requiring consensus for key operations to ensure trust [1, 7]. This document will focus on distributed systems.

Distributed systems are widely used across various fields, including banking and healthcare, and are the focus of ongoing research as they expand into emerging areas like cloud and edge computing [8–10]. Their evolution is driven by the numerous advantages they offer, such as scalability, reliability, and transparency when well-structured. However, these benefits also introduce new challenges, increasing the complexity of debugging and testing, for example [8]. The following subsections will provide a detailed exploration of these aspects.

#### 2.1.1 Characteristics

On a distributed system, when being well-structured, it is possible to find, among others, the following most popular characteristic:

## Transparency

Due to their transparency, distributed systems allow end-users interact with them without them realizing how complex they are [1, 11]. This trait is called transparency, and it can manifest in a variety of ways. These consist of:

1. **Access transparency:** This enables resources to be accessed seamlessly across different nodes, whether local or remote, while hiding differences such as operating systems, programming languages, or other implementation details [1, 9].
2. **Location transparency:** This hides the physical location of resources or nodes, allowing users to access them without needing to know where they are located. For example, a Uniform Resource Locator (URL) provides location transparency by enabling users to access a resource abstracting the physical location [1, 12].
3. **Relocation transparency:** This ensures that if a resource or node is moved to a new physical location, the change is invisible to users. For instance, if a website is relocated, its URL remains the same [13].
4. **Mobility transparency:** This allows both clients and resources to move without disrupting ongoing operations for users or applications. An example of mobility transparency is a mobile phone call, where communication remains unaffected even if the involved are moving [9, 13].
5. **Replication transparency:** This hides the replication of resources or nodes, which may occur to improve reliability and availability. For instance, a distributed file system may replicate data to ensure availability even if one copy becomes inaccessible [1].
6. **Concurrency transparency:** This allows multiple processes to operate concurrently without conflict, even if they share the same physical resources [9].
7. **Failure transparency:** This enables the system to continue functioning despite certain failures, making these issues invisible to users while ensuring that tasks are completed as intended [9].

## Reliability and availability

A distributed system should have reliability and availability aspects. Reliability refers to its ability to continuously perform its intended requirements without interruption, operating exactly as designed, even in the presence of certain internal failures [14]. A highly reliable system maintains consistent, uninterrupted service over an extended period, minimizing disruptions for users [1]. On other hand, availability measures the probability that the system is operational and ready to respond correctly at any given moment, often expressed as a percentage of system up-time [1, 15].

## Scalability

Designing and building a distributed system is complex, but also enables the creation of highly scalable systems, capable of expanding to meet increasing demands [1, 3, 16]. This characteristic is particularly evident as cloud-based systems become more popular, allowing users to interact with applications over the internet rather than relying on local desktop computing power [10]. Cloud services must support a large volume of simultaneous connections and interactions, making scalability a crucial factor [1].

*"A system is described as scalable if it will remain effective when there is a significant increase in the number of resources and the number of users." [9]*

Scalability can be addressed across three dimensions: size scalability allows for additional users and resources without loss in performance, geographical scalability maintains performance despite physical distance between nodes, and administrative scalability manages increasing complexity as nodes take on distinct management roles [1].

### **Fault tolerance**

Fault tolerance is a critical characteristic of distributed systems, closely linked to reliability, availability, and scalability. For a system to maintain these properties, it must be able to mask failures and continue operating despite the presence of errors [17]. Fault tolerance is especially vital in distributed environments where system failures can lead to significant disruptions and economic losses across sectors such as finance, telecommunications, and transportation [6].

The primary goal of a fault-tolerant system is to enable continuous operation by employing specific strategies and design patterns to mask the possible errors [18]. Due to the importance of this topic to the dissertation, fault tolerance it will be detailed on the next chapter.

### **2.1.2 Communication**

In distributed systems, communication between nodes is crucial for coordination and data exchange. When nodes are separated by a network, communication occurs over that network, whereas nodes on the same machine uses Interprocess Communication (IPC) (Inter-Process Communication) [3]. Network-based communication can introduce delays or be unreliable, making asynchronous communication advantageous in many situations [19]. However, some scenarios require synchronous communication for immediate feedback.

#### **Synchronous and asynchronous communication**

In synchronous communication, the sender waits for a response before continuing, making it a blocking operation [1, 9]. This method is helpful when the sender requires confirmation or feedback to proceed.

Asynchronous communication, by contrast, enables the sender to continue processing without waiting for a response, supporting a non-blocking flow [1, 20]. This approach is well-suited to systems with high heterogeneity or where decoupling is essential, often implemented with message queues that provide transparency between sender and receiver [1].

#### **Communication models**

Within synchronous and asynchronous types, several communication models exist:

##### *Remote Procedure Call (RPC)*

RPC supports location transparency, allowing the sender to make a request to a remote node as if the call were local, within the same process or environment [9, 18]. This is achieved through a mechanism known as a stub, which handles request and response processes on

both client and server sides, ensuring data parsing and call transparency over the network [1].

#### *Message passing*

In message passing, data is encapsulated into a message and sent to a queue on the receiver's side. This model typically follows an asynchronous approach, where the sender does not expect an immediate reply [9, 18]. However, message passing can also be synchronous if the sender requires an acknowledgment of receipt [9]. Communication via message passing can be managed by a broker, such as RabbitMQ,<sup>1</sup> for example, or implemented natively, as seen in Erlang's messaging passing system [1, 21].

### 2.1.3 Challenges

Distributed systems encounter numerous challenges, including scalability [14], managing software, network, and disk failures [4, 22], heterogeneity [9], coordination among nodes [3], and difficulties on debugging and testing [22, 23]. For the scope of this dissertation only the CAP theorem will be discussed.

#### **CAP theorem**

The CAP theorem says that in a system where nodes are networked and share data, it is impossible to simultaneously achieve all three properties of Consistency, Availability, and Partition Tolerance [1, 3]. This theorem underlines a critical trade-off in distributed systems: only two of these properties can be fully ensured at any given time [24, 25]. A description of the properties can be given by:

- **Consistency:** Ensures that all nodes in the system reflect the same data at any time, so each read returns the latest write.
- **Availability:** Guarantees that every request receives a response, whether successful or not, even if some nodes are offline.
- **Partition tolerance:** Allows the system to continue operating despite network partitions, where nodes may temporarily lose the ability to communicate.

According to the CAP theorem, when a network partition occurs, a distributed system must prioritize either consistency or availability, as achieving all three properties is not feasible in practice [1, 3, 24]. This concept is directly relevant to this dissertation, as fault tolerance strategies discussed later will account for these trade-offs to optimize specific properties.

### 2.1.4 Evolution of distributed systems

Distributed systems have been a significant area of research in computer science. The journey began with early client-server models in mainframe computing, which allowed virtualization to increase efficiency, which is a popular approach still in use today [10]. This was followed by the development of cluster computing, offering affordable ways to boost computing power without relying on costly mainframes [10, 26]. During this phase, the Internet and the World Wide Web (WWW) were created, marking an important point in history [10].

---

<sup>1</sup>«Official website of RabbitMQ». <https://www.rabbitmq.com/> (accessed 02 November 2024)



Subsequently, Peer to Peer (P2P) systems emerged, decentralizing computing through co-operative and distributed processes. With the rise of cloud computing, mobile computing, and the Internet of Things (IoT) reached new levels of scalability, flexibility, and accessibility [26]. Another design that emerged is micro-services architecture, which shares many properties with distributed systems and is widely used on the industry [27]. Today, cutting-edge advancements focus on the following paradigms [26]:

- **Edge Computing:** Extends computing resources closer to data sources and end users, reducing latency and improving performance in real-time applications. This approach is critical for IoT and applications where immediacy is essential [10, 26, 28].
- **Fog Computing:** Provides a layer of computing power between the cloud and edge devices, enhancing processing capabilities closer to the data source [10, 26, 28].
- **Dew Computing:** Complements cloud, edge, and fog computing by enabling local devices to perform data processing even when disconnected from centralized resources, enhancing resilience and accessibility for remote or offline scenarios [26, 29].
- **Serverless Computing:** Enables developers to execute code without managing underlying servers, automatically scaling resources based on demand. This model reduces operational overhead and allows efficient, event-driven architectures [26, 30].

## 2.2 Fault Tolerance

With the extensive use of software systems across various domains, such as banking, transportation, and more, the demand for reliable and available systems is increasingly essential. However, errors in software are inevitable, making fault tolerance a critical attribute for systems to continue functioning correctly even in the presence of failures [6]. Fault tolerance can address a range of issues, including networking, hardware, software, and other dimensions, with various strategies designed to manage these different fault types [2, 31].

### 2.2.1 Fault Tolerance Taxonomy

It is important to classify and understand the types of faults and failures that can arise. This section presents a taxonomy of fault tolerance concepts, drawing on the framework proposed by Isukapalli et al.[32]. A fault is defined as an underlying defect within a system component that may lead to an error, which is a deviation from the intended internal state. If this error remains unresolved, it may escalate into a system failure, potentially impacting system functionality either partially or completely [32, 33].

Table 2.1 provides a classification of faults that commonly arise in fault tolerant systems. Each type requires different strategies for detection and mitigation, depending on the nature and persistence of the fault [1, 32].

Failures, on the other hand, are the external manifestations of these internal faults, as outlined in Table 2.2. These include crash failures, where the system halts entirely, to arbitrary failures, where responses are erratic and potentially misleading [1, 34].

### 2.2.2 Strategies

Various strategies and mechanisms can be applied to a system to achieve fault tolerance, and these must be chosen to suit the specific system type. This dissertation will primarily

Fault Type	Description
Transient Faults	Temporary faults that occur one time and retrying or restarting the operation the problem it disappear [1, 33].
Intermittent Faults	Faults that appear sporadically, similar to transient faults, but with a higher persistence often due to hardware or environmental factors (e.g., temperature fluctuations affecting a hard disk) [32, 33].
Permanent Faults	Persistent faults caused by a complete failure of a system component. These faults remain until the root cause is identified and corrected, making them relatively straightforward to fix [33].

Table 2.1: Types of Faults and Their Descriptions

Type of Failure	Description
Crash Failure	The system halts and stops all operations entirely. Although it was functioning correctly before the halt, it does not resume operations or provide responses after the failure. [1]
Omission Failure	The system fails to send or receive necessary messages, impacting communication and task coordination. [34]
Timing Failure	The system's response occurs outside a specified time interval, either too early or too late, causing issues in time-sensitive operations. [32]
Response Failure	The system provides incorrect outputs or deviates from expected state transitions, potentially leading to erroneous results. [1]
Arbitrary Failure	The system produces random or unpredictable responses at arbitrary times, potentially with incorrect or nonsensical data. This type of failure is challenging to diagnose and manage. [1]

Table 2.2: Brief Description of Failure Types

focus on software fault tolerance strategies, and focused on those suitable for the distributed languages below presented. Therefore, next it will be shown some strategies that it will serve as a theoretical basis for some of techniques that it will be used.

### Retry

The retry strategy is a popular and straightforward technique that involves repeating an operation that initially failed, under the assumption that it might succeed upon retry [11, 31]. This strategy may include configurations like a back-off delay between attempts, but its fundamental principle remains the same.

### Replication

Replication is a technique aimed at masking errors by creating redundant task clones. In this approach, multiple replicas of a job run simultaneously, acting as a group that performs the same operations. This redundancy allows the system to provide a response even in the event of a host, network, or other types of errors. Replication strategies can vary in communication modes, which may be synchronous or asynchronous. In some cases, a consensus algorithm is needed to reach a final decision among the replicas [1, 11, 32].

### Check-pointing and Message Logging

The check-pointing strategy periodically saves the state of a process so that, in the event of a failure, the process can restart from the last saved state, or "checkpoint," rather than start all over. This approach reduces the need to repeat the entire operation [9, 32].

Message logging is a lighter-weight approach with a similar goal. Instead of saving entire checkpoints, it records all the necessary messages that lead the process to a specific state. In case of a failure, the messages are replayed in the same order, guiding the system back to the desired state [11].

### 2.2.3 Evolution and the State-of-the-Art

[ajustar isto](#)

Fault tolerance has significantly evolved, with substantial advancements in both software and hardware techniques. As studied by Arshad A. et al. [35], recent developments focus on leveraging novel technologies to enhance system resilience. Furthermore, with Artificial Intelligence (AI) and Machine Learning (ML) has introduced predictive fault tolerance techniques that aim to anticipate and mitigate faults before they lead to system failures.

Current strategies can be categorized into three types:

- **Reactive Strategies:** These methods aim to reduce the impact of faults after they occur. For instance, techniques like task migration on processor level [Fochi2021] and checkpoint/restart mechanisms [Hussain2021] allow systems to recover by either transferring tasks or restarting operations from saved states.
- **Proactive Strategies:** Proactive methods focus on preventing faults by monitoring system conditions and taking preemptive actions. For example, Fayyaz et al. proposed a distributed satellite architecture that integrates fault-tolerant performance measurements to manage processor faults effectively [36].
- **Predictive Strategies:** With advancements in AI and ML, predictive fault tolerance has emerged as a cutting-edge approach, wherein algorithms forecast potential failures and initiate corrective measures. For instance, Ghosh et al. utilized a fault management subsystem within a transactive energy application to improve system resilience by analyzing interaction patterns [Ghosh2021].

Several innovative techniques have been proposed to address specific challenges in fault tolerance. Examples include the use of erasure coding in distributed file systems to balance storage costs and redundancy [Arafa2021], adaptive learning for fault detection in multi-agent systems [Khalili2021], and circuit breaker patterns for fault recovery in stream processing architectures [Knasmueller2021].

## 2.3 Distributed and Concurrency Programming

Distributed and concurrent programming languages play an important role in building resilient, scalable, and fault-tolerant systems [37]. In distributed systems, where components operate across multiple nodes, and in concurrent systems, where tasks can execute in parallel on the same machine's Central Processing Unit (CPU), programming languages must provide mechanisms to manage faults effectively. These mechanisms should isolate faults

to prevent cascading failures, at the same time ensuring overall system reliability and availability [21], or should have forms to equip the language with capacities to handle this type of systems.

The evolution of distributed programming languages has been driven by the need to address the complexities of developing distributed systems, which include issues such as concurrency, parallelism, fault tolerance, and secure communication. This has driven the evolution of new paradigms, languages, frameworks, and libraries aimed at reducing development complexity in distributed and concurrent systems [16].

### 2.3.1 Models and Paradigms

The field of distributed programming has been shaped by research and development in concurrency and parallelism, and some models and paradigms have been developed to address this challenge, where some ideas had some focus restricted to the research others have been addressed to the industry. In the following it will be described the models and paradigms that bring interest to this dissertation:

#### Actor Model

The Actor Model, a conceptual framework for concurrent and distributed computing, was introduced by Carl Hewitt in 1973 [38]. It defines a communication paradigm where an actor, the fundamental unit of computation, interacts with other actors exclusively through asynchronous message passing, with messages serving as the basic unit of communication [39]. Each actor is equipped with its own mailbox, which receives messages and processes them sequentially [40].

A core principle of the Actor Model is isolation, maintaining their own internal state that is inaccessible and immutable by others [40]. This eliminates the need for shared memory, reducing complexity and potential data races. Another key feature is location communication, which allows actors to communicate seamlessly regardless of their physical location, whether on the same machine or across a distributed network. This transparency is achieved because all interactions are inherently message-based [39].

The Actor Model also introduces the concept of supervision, where actors can monitor the behavior of other actors and take corrective actions in the event of a failure. This supervisory mechanism significantly enhances fault tolerance, enabling systems to recover gracefully from errors without compromising overall reliability [39].

The Actor Model has been instrumental in shaping distributed system design and has been natively implemented in programming languages such as Erlang, Clojure and Elixir [41]. Additionally, the model has been extended to other languages through frameworks and libraries. For instance, Akka brings actor-based concurrency to Scala, C# and F# while Kilim offers similar functionality for Java [39]. Comparable patterns can also be adopted in other languages like Go, Rust, and Ruby using libraries or custom abstractions.

#### Communicating Sequential Processes (CSP)

The field of distributed computing emphasizes mathematical rigor in algorithm analysis, with one of the most influential models being Communicating Sequential Processes (CSP), introduced by C.A.R. Hoare in 1978 [42].

## refs

CSP offers an abstract and formal framework for modeling interactions between concurrent processes through channels, which serve as the communication medium between them [43]. Processes operate independently, but they are coupled via these channels, and communication is typically synchronous, requiring the sender and receiver to synchronize for message transfer. While similar in some respects to the Actor Model, CSP distinguishes itself through its emphasis on direct coupling via channels and strict synchronization [**<empty citation>**].

The CSP model has had a profound influence on programming languages and frameworks. For example, Go integrates CSP concepts in its implementation of goroutines and channels, enabling structured concurrency and communication [16, 44]. Additionally, the language Occam offers a more direct and faithful realization of CSP principles, making it a pioneering choice [45].

### Microservices Architectures

Despite extensive research in distributed programming languages, many have not achieved widespread adoption in industry [**<empty citation>**]. As a result, a new approach for designing distributed systems has emerged: microservices architectures. This paradigm shifts the focus to a higher level of abstraction, making the architecture language-agnostic by decomposing a system into a collection of loosely coupled services, each responsible for a specific function. These services interact using lightweight protocols such as HTTP/REST, gRPC, or message queues, promoting separation of concerns, modularity, and flexibility [**Fowler2014**].

This design approach has enabled general-purpose programming languages to compete with, and sometimes complement, native distributed programming paradigms. By combining robust frameworks or libraries with microservices principles, these languages can facilitate the development of distributed and fault-tolerant systems, bridging the gap between traditional distributed models and modern software architecture practices. In some cases, microservices architectures even incorporate features from native paradigms, merging their strengths to create scalable, resilient systems.

### Comparison of Models and Paradigms

#### 2.3.2 Distributed and Concurrent Programming Languages

##### Analyses and Justification



## **Chapter 3**

# **Literature Review**

### **3.1 Research Questions**

### **3.2 State of Art**

### **3.3 Conclusions**





## **Chapter 4**

# **Planning**

### **4.1 WBS**

### **4.2 Gantt Diagram**

### **4.3 Risk Management**



# Bibliography

- [1] S. Andrew Tanenbaum and M. Maarten Van Steen. *Distributed systems*. 4th ed. Maarten Van Steen, 2023. isbn: 978-90-815406-3-6.
- [2] Sanjeev Sharma, Rajiv Gandhi Proudhyogiki, and Sanjay Bansal. *A Detailed Review of Fault-Tolerance Techniques in Distributed System*. 2016. url: <https://www.researchgate.net/publication/228619369>.
- [3] Roberto Vitillo. *Understanding Distributed Systems: What every developer should know about large distributed applications*. 2021.
- [4] Nitin Naik. "Performance Evaluation of Distributed Systems in Multiple Clouds using Docker Swarm". In: *15th Annual IEEE International Systems Conference, SysCon 2021 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., Apr. 2021. isbn: 9781665444392. doi: 10.1109/SysCon48628.2021.9447123.
- [5] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System and the Ordering of Events in a Distributed System*. 1978. doi: <https://doi.org/10.1145/359545.359563>.
- [6] Arif Sari and Murat Akkaya. "Fault Tolerance Mechanisms in Distributed Systems". In: *International Journal of Communications, Network and System Sciences* 08 (12 2015), pp. 471–482. issn: 1913-3715. doi: 10.4236/ijcns.2015.812042.
- [7] AWS - Amazon. *What is Decentralization? - Decentralization in Blockchain Explained - AWS*. Accessed at 03.11.2024. url: <https://aws.amazon.com/web3/decentralization-in-blockchain/>.
- [8] Brendan Burns. *Designing Distributed Systems Patterns and Paradigms for scalable, reliable services*. 2018.
- [9] George Coulouris et al. *Distributed Systems - Concepts and Design*. 2012.
- [10] Dominic Lindsay et al. "The evolution of distributed computing systems: from fundamental to new frontiers". In: *Computing* 103 (8 Aug. 2021), pp. 1859–1878. issn: 14365057. doi: 10.1007/s00607-020-00900-y.
- [11] Mohamed. Amroune, Makhlof. Derdour, and Ahmed. Ahmim. *Fault Tolerance in Distributed Systems: A Survey*. IEEE, 2018. isbn: 9781538642382.
- [12] Michel Banatre. *Hiding distribution in distributed systems*. 1991. doi: 10.1109/ICSE.1991.130643.
- [13] Maarten van Steen and Andrew S. Tanenbaum. "A brief introduction to distributed systems". In: *Computing* 98 (10 Oct. 2016), pp. 967–1009. issn: 0010485X. doi: 10.1007/s00607-016-0508-7.
- [14] Waseem Ahmed and Yong Wei Wu. "A survey on reliability in distributed systems". In: *Journal of Computer and System Sciences*. Vol. 79. Academic Press Inc., 2013, pp. 1243–1255. doi: 10.1016/j.jcss.2013.02.006.
- [15] Atlassian. *Reliability vs availability: Understanding the differences*. Accessed at 16.10.2024. url: <https://www.atlassian.com/incident-management/kpis/reliability-vs-availability>.
- [16] Ivan Valkov, Natalia Chechina, and Phil Trinder. "Comparing languages for engineering server software: Erlang, go, and scala with akka". In: *Proceedings of the ACM*

- Symposium on Applied Computing*. Association for Computing Machinery, Apr. 2018, pp. 218–225. isbn: 9781450351911. doi: 10.1145/3167132.3167144.
- [17] Lorenzo Strigini. “Fault Tolerance and Resilience: Meanings, Measures and Assessment”. In: Springer Berlin Heidelberg, 2012, pp. 3–24. doi: 10.1007/978-3-642-29032-9\_1.
  - [18] Martin Kleppmann. *Designing Data Intensive Applications*. 2017.
  - [19] Xinhao Yuan and Junfeng Yang. “Effective concurrency testing for distributed systems”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*. Association for Computing Machinery, Mar. 2020, pp. 1141–1156. isbn: 9781450371025. doi: 10.1145/3373376.3378484.
  - [20] Rob van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke. “On Synchronous and Asynchronous Interaction in Distributed Systems”. In: (Dec. 2008). url: <http://arxiv.org/abs/0901.0048>.
  - [21] Jan Henry Nystrom. *Fault Tolerance in Erlang*. 2009.
  - [22] AWS - Amazon. *Challenges with distributed systems*. Accessed at 03.11.2024. url: <https://aws.amazon.com/builders-library/challenges-with-distributed-systems/>.
  - [23] Ivan Beschastnikh et al. “Visualizing Distributed System Executions”. In: *ACM Transactions on Software Engineering and Methodology* 29 (2 Mar. 2020). issn: 15577392. doi: 10.1145/3375633.
  - [24] IBM. *What is the CAP theorem?* Accessed at 03.11.2024. Aug. 2024. url: <https://www.ibm.com/topics/cap-theorem>.
  - [25] Seth Gilbert and Nancy A Lynch. *Perspectives on the CAP Theorem*. 2012. doi: 10.1109/MC.2011.389.
  - [26] Sukhpal Singh Gill et al. “Modern Computing: Vision and Challenges”. In: (Jan. 2024). doi: 10.1016/j.teler.2024.100116. url: <http://arxiv.org/abs/2401.02469> %20<http://dx.doi.org/10.1016/j.teler.2024.100116>.
  - [27] Amr S. Abdelfattah and Tomas Cerny. “Roadmap to Reasoning in Microservice Systems: A Rapid Review”. In: *Applied Sciences (Switzerland)* 13 (3 Feb. 2023). issn: 20763417. doi: 10.3390/app13031838.
  - [28] Keyan Cao et al. *An Overview on Edge Computing Research*. 2020. doi: 10.1109/ACCESS.2020.2991734.
  - [29] Marjan Gusev. “What makes dew computing more than edge computing for internet of things”. In: *Proceedings - 2021 IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021*. Institute of Electrical and Electronics Engineers Inc., July 2021, pp. 1795–1800. isbn: 9781665424639. doi: 10.1109/COMPSAC51774.2021.00269.
  - [30] Yongkang Li et al. “Serverless Computing: State-of-the-Art, Challenges and Opportunities”. In: *IEEE Transactions on Services Computing* 16 (2 Mar. 2023), pp. 1522–1539. issn: 19391374. doi: 10.1109/TSC.2022.3166553.
  - [31] Ahmad Shukri Mohd Noor, Nur Farhah Mat Zian, and Fatin Nurhanani M. Shai-ful Bahri. “Survey on replication techniques for distributed system”. In: *International Journal of Electrical and Computer Engineering* 9 (2 Apr. 2019), pp. 1298–1303. issn: 20888708. doi: 10.11591/ijece.v9i2.pp1298-1303.
  - [32] Sucharitha Isukapalli and Satish Narayana Srirama. *A systematic survey on fault-tolerant solutions for distributed data analytics: Taxonomy, comparison, and future directions*. Aug. 2024. doi: 10.1016/j.cosrev.2024.100660.

- [33] Federico Reghenzani, Zhishan Guo, and William Fornaciari. "Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions". In: *ACM Computing Surveys* 55 (14 Dec. 2023). issn: 15577341. doi: 10.1145/3589950.
- [34] Sajjad Haider et al. *Fault Tolerance in Distributed Paradigms*. fala sobre algumas técnicas de fault-tolerance e faz uma revisão de literatura resumida sobre as estratégias que são usadas<br/>. 2011.
- [35] Arshad A. Hussein et al. "State of Art Survey for Fault Tolerance Feasibility in Distributed Systems". In: *Asian Journal of Research in Computer Science* (Sept. 2021), pp. 19–34. doi: 10.9734/ajrcos/2021/v11i430268.
- [36] Tanya Vladimirova and Muhammad Fayyaz. "Fault-Tolerant Distributed Attitude and Orbit Control System for Space Applications". In: *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. 2018, pp. 43–50. doi: 10.1109/AHS.2018.8541486.
- [37] Joe Armstrong. *Early Praise for Programming Erlang, Second Edition*. 2013.
- [38] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular ACTOR formalism for artificial intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.
- [39] Phil Trinder et al. "Scaling reliably: Improving the scalability of the Erlang distributed actor platform". In: *ACM Transactions on Programming Languages and Systems* 39 (4 Aug. 2017). issn: 15584593. doi: 10.1145/3107937.
- [40] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. "43 years of actors: a taxonomy of actor models and their key properties". In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 31–40. isbn: 9781450346399. doi: 10.1145/3001886.3001890. url: <https://doi.org/10.1145/3001886.3001890>.
- [41] Aidan Randtoul and Phil Trinder. "A reliability benchmark for actor-based server languages". In: *Erlang 2022 - Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*. Association for Computing Machinery, Inc, Sept. 2022, pp. 21–32. isbn: 9781450394352. doi: 10.1145/3546186.3549928.
- [42] C. A.R. Hoare. "Communicating sequential processes". In: *Communications of the ACM* 26 (1 Jan. 1983), pp. 100–106. issn: 15577317. doi: 10.1145/357980.358021. url: <https://dl.acm.org/doi/10.1145/357980.358021>.
- [43] Ciprian Paduraru and Marius Constantin Melemciuc. "Parallelism in C++ Using Sequential Communicating Processes". In: *Proceedings - 17th International Symposium on Parallel and Distributed Computing, ISPDC 2018*. Institute of Electrical and Electronics Engineers Inc., Aug. 2018, pp. 157–163. isbn: 9781538653302. doi: 10.1109/ISPDC2018.2018.00030.
- [44] Go. *Official documentation of Go programming language*. Accessed at 10.11.2024. url: <https://go.dev/doc/>.
- [45] Matilde Brolos, Carl Johannes Johnsen, and Kenneth Skovhede. "Occam to Go translator". In: *Proceedings - 2021 Concurrent Processes Architectures and Embedded Systems Conference, COPA 2021*. Institute of Electrical and Electronics Engineers Inc., Apr. 2021. isbn: 9781728166834. doi: 10.1109/COPA51043.2021.9541431.