ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

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, December 1, 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, December 1, 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Problem

1.2 Objectives

1.3 Ethical Considerations

1.4 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 [1]. Nodes in a distributed system may refer to physical devices or software processes [2].

To the end-user, distributed systems appear as a single, large virtual system, making the underlying logic transparent [2]. These systems achieve a shared objective by transmitting messages through various nodes and dividing computational tasks among them, increasing resilience and isolating business logic [2, 3]. 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, 3].

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

Transparency in distributed systems enables seamless user interaction by hiding the complexity of underlying operations [1, 4]. Key aspects include access transparency, which allows resource usage without concern for system differences, and location transparency, which hides the physical location of resources, as seen with Uniform Resource Locators (URLs) [1, 5]. Replication transparency ensures reliability by masking data duplication, while failure transparency enables systems to handle faults without user disruption [1, 5]. Together, these forms of transparency enhance usability, robustness, and reliability.

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 [6]. 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, 7].

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, 2, 8]. 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 [9]. Cloud services must support a large volume of simultaneous connections and interactions, making scalability a crucial factor [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 [1]. 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 [3].

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 [10].

2.1.2 Communication

Communication is fundamental in distributed systems for coordination and data exchange. Nodes communicate over networks or via Interprocess Communication (IPC) when on the same machine [2]. Synchronous communication involves blocking operations where the sender waits for a response, suitable for scenarios requiring confirmation [1, 5]. In contrast, asynchronous communication allows non-blocking operations, enabling the sender to proceed without waiting. This approach, often supported by message queues, is ideal for decoupled and heterogeneous systems [1].

2.1.3 Challenges

Distributed systems encounter numerous challenges, including scalability [6], managing software, network, and disk failures [11, 12], heterogeneity [5], coordination among nodes [2], and difficulties on debugging and testing [12, 13]. 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, 2]. This theorem underlines a critical trade-off in distributed systems: only two of these properties can be fully ensured at any given time [14, 15]. 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, 2, 14]. This concept is directly relevant to this dissertation, as fault tolerance strategies discussed later will account for these trade-offs to optimize specific properties.

2.2 Fault Tolerance

With the extensive use of software systems across various domains, the demand for reliable and available systems is 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 [3]. 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 [1, 16].

2.2.1 Fault Tolerance Taxonomy

It is important to classify and understand the types of failures that can arise. This section presents a taxonomy of fault tolerance concepts, drawing on the framework proposed by Isukapalli et al.[17]. 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 [17, 18].

Failures are the external manifestations of the internal faults, as outlined in Table 2.1. These include crash failures, where the system halts entirely, to arbitrary failures, where responses are erratic and potentially misleading [1].

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. [17]
Timing Failure	The system's response occurs outside a specified time interval, either too early or too late, causing issues in time-sensitive operations. [17]
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 2.1: Brief Description of Failure Types

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 focus on software fault tolerance strategies, and focused on those suitable for the programming languages bellow 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 [4, 16].

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, 4, 17].

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 [5, 17].

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 [4].

2.3 Distributed and Concurrency Programming

Distributed and concurrent programming languages play an important role in building resilient and fault-tolerant systems [19]. In distributed systems, where components operate across multiple nodes, and in concurrent systems, where tasks can execute in parallel or concurrently 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 [20], or should have forms to equip the language with capacities to handle this type of systems by frameworks or libraries.

The evolution of distributed programming languages help to address the complexities of developing distributed systems, which include issues such as concurrency, parallelism, fault tolerance, and secure communication [19]. This has driven the evolution of new paradigms, languages, frameworks, and libraries aimed at reducing development complexity in distributed and concurrent systems [8].

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. 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 [21]. 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 [22]. Each actor is equipped with its own mailbox, which receives messages and processes them sequentially [23].

A core principle of the Actor Model is isolation, maintaining their own internal state that is inaccessible and immutable by others [23]. This eliminates the need for shared memory, reducing complexity and potential data races [8].

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 [22].

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 [24]. 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 [22]. Comparable patterns can also be adopted in other languages like Go, Rust, and Ruby using libraries or custom abstractions.

Communicating Sequential Processes

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 [25].

CSP offers an abstract and formal framework for modeling interactions between concurrent processes through channels, which serve as the communication medium between them [26]. 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 [25]. While similar in some respects to the Actor Model, CSP distinguishes itself through its emphasis on direct coupling via channels and synchronization.

The CSP model influenced on programming languages and frameworks. For example, Go integrates CSP concepts in its implementation of goroutines and channels [8, 26, 27]. In addition, the language Occam attempts to offer a direct implementation of CSP principles with its focus on critical projects such as satellites [28].

Microservices Architectures

A significant evolution in designing distributed systems has emerged with the appearance of microservices architectures. This paradigm elevates the focus to a higher level of abstraction, enabling language-agnostic systems by decomposing a monolithic application into a collection of loosely coupled, independently deployable services, each responsible for a specific function [29]. These services communicate using lightweight protocols such as Hypertext Transfer Protocol (HTTP), Google Remote Procedure Call (gRPC), or message queues, fostering separation of concerns, modularity, scalability, and fault tolerance [29].

Microservices architectures allow general purpose programming languages to participate in distributed computing paradigms by leveraging frameworks, libraries, and microservices principles [30].

Although microservices are often associated with strict business principles, their abstract concepts can be adapted to focus on architectural designs that leverage communication middleware for distributed communication. By adopting these principles, it becomes possible to create distributed systems with fault-tolerant capabilities using general-purpose programming languages.

2.3.2 Distributed and Concurrent Programming Languages

Distributed and concurrent programming languages are designed to handle multiple tasks simultaneously across systems or threads. Some languages, such as Java, Rust, and lower-level languages like C with PThreads, require developers to explicitly manage concurrency [8, 26]. These approaches often introduce complexity, increasing the probability of deadlocks or race conditions. This has driven the need for languages and frameworks that abstract away these challenges, offering safer and more developer-friendly concurrency models [8].

One widely adopted paradigm for mitigating concurrency issues is the Actor Model. By avoiding shared state and using message passing for communication, the Actor Model reduces risks inherent in traditional concurrency mechanisms such as mutexes and locks [8]. Erlang, for instance, is renowned for its fault tolerance and "let-it-crash" philosophy, which delegates error handling to its virtual machine [20]. Supervising actors monitor and recover from failures, making Erlang highly suitable for building robust distributed systems [19]. Building on Erlang's foundation, Elixir introduces modern syntax and developer tooling while retaining Erlang's strengths for creating large-scale, fault-tolerant systems. These features make Elixir a popular choice for modern distributed systems development [31].

Haskell, a pure functional programming language, provides a deterministic approach to concurrency, ensuring consistent results regardless of execution order [8]. Its extension, Cloud Haskell[1], builds upon the Actor Model, drawing inspiration from Erlang, to allow distributed computation through message passing.

Similarly, Akka, a framework built with Scala, adopts the Actor Model to support distributed and concurrent applications. Akka combines Scala's strengths in functional and object-oriented programming, enabling developers to merge these paradigms effectively [8]. Unlike Erlang, Akka operates on the Java Virtual Machine (JVM), providing seamless interoperability with Java-based systems [32].

[1] Official website of Cloud Haskell: `https://haskell-distributed.github.io/` (accessed 25 November 2024)

Go, developed by Google, simplifies concurrent programming through its lightweight goroutines and channels, inspired by the CSP paradigm, which abstracts threading complexities [28]. Go's emphasis on simplicity and performance has made it a preferred choice for developing scalable microservices and cloud-native applications, particularly as microservices architectures continue to gain popularity [27].

For specialized use cases like Big Data processing, frameworks such as Hadoop provide distributed computing capabilities tailored to data-intensive tasks. Hadoop abstracts the complexities of handling distributed storage and processing, offering features such as scalability, fault tolerance, and data replication [33].

Other pioneer languages, such as Emerald, Oz, and Hermes, still exist but have minimal community and industry support, as reflected in popularity rankings like RedMonk January 2024[2] and Tiobe November 2024[3].

Conversely, some relatively recent languages have gained attention. Unison[4] employs content-addressed programming using hash references to improve code management and distribution. Gleam[5] compiles to Erlang and offers its own type-safe implementation of Open Telecom Platform (OTP), Erlang's actor framework. Pony[6], an object-oriented language based on the Actor Model, introduces reference capabilities to ensure concurrency safety. However, these languages have yet to achieve significant industry adoption, as evidenced by their absence from the RedMonk January 2024 and Tiobe November 2024 rankings.

In Table 2.2, the most relevant languages and frameworks for this theme are presented to facilitate a concise analysis. Additionally, rankings from TIOBE November 2024 and IEEE Spectrum August 2024[7] are included to provide an overview of their popularity and adoption.

Analyses and Language Choice Justification

The focus of this dissertation is on Elixir as the central language for comparison. Elixir is chosen due to its modern syntax, developer-friendly tooling, and robust foundation on the BEAM virtual machine [31]. Since Elixir inherits all the strengths of Erlang [8], including fault tolerance and the Actor Model, a direct comparison with Erlang is unnecessary as they share the same core runtime and strategies. Such a comparison would likely yield redundant results and add little value to the research.

On the other hand, comparing Elixir with low-level languages like Java, Rust, and C would also be less effective. These languages require explicit management of concurrency and fault tolerance [8], introducing complexities that diverge significantly from Elixir's high-level abstractions. A comparison in this context might be unfair and would not provide meaningful insights given the focus on fault tolerance and distributed systems.

Instead, a comparison with Scala and Akka provides a more relevant perspective. Both Elixir and Akka share the paradigm Actor Model for concurrency and fault tolerance, but their underlying virtual machines differ: the BEAM for Elixir and the JVM for Akka [32].

[2] RedMonk January 2024: `https://redmonk.com/sogrady/2024/03/08/language-rankings-1-24/` (accessed 28 November 2024)

[3] Tiobe November 2024: `https://www.tiobe.com/tiobe-index/` (accessed 28 November 2024)

[4] Official website of Unison: `https://www.unisonweb.org/` (accessed 27 November 2024)

[5] Official website of Gleam: `https://gleam.run/` (accessed 27 November 2024)

[6] Official website of Pony: `https://www.ponylang.io/` (accessed 27 November 2024)

[7] IEEE Spectrum 2024: `https://spectrum.ieee.org/top-programming-languages-2024/` (accessed 28 November 2024)

Name	**Concurrency Strategy**	**Model**	**TIOBE Nov 2024**	**IEEE Spectrum 2024**
Java	Explicit	Object-Oriented	3	2
Rust	Explicit	Procedural	14	11
C (PThreads)	Explicit	Procedural	4	9
Erlang	Actor Model	Functional	50+	48
Elixir	Actor Model	Functional	44	35
Haskell	Evaluation Strategy	Functional	34	38
Scala (Akka)	Actor Model	Functional	30	16
Go	CSP	Procedural	7	8
Hadoop	Distributed Framework	Procedural	N/A	N/A
Unison	Hash References	Functional	N/A	N/A
Gleam	Actor Model	Functional	N/A	N/A
Pony	Actor Model	Object-Oriented	N/A	N/A

Table 2.2: Characteristics of Distributed and Concurrent Programming Languages

Additionally, Scala with Akka is notable for its community acceptance [8]. This comparison is valuable because it explores how different implementations of the same paradigm can influence fault tolerance strategies and performance.

Furthermore, too recent or older languages with minimal popularity, such as Emerald, Oz, Unison and Gleam are excluded from this study. These languages lack widespread adoption, and insights derived from them would have limited applicability for the majority of developers, as demonstrated in Table 2.2 with a non-appearance in the Tiobe and IEEE Spectum rankings.

From another perspective, the inclusion of Go in this study adds an interesting dimension to the comparison. Go, unlike Elixir and Akka, does not have built-in support for native distributed systems. However, its increasing popularity and industry adoption make it a strong candidate for exploration [28]. By examining how Go can achieve fault tolerance using libraries and abstractions under a microservices strategy, the study can assess whether an external abstraction layer can match or exceed the capabilities of languages with native support. This investigation could reveal whether the flexibility of a non-native distributed model can compensate for the lack of built-in features, providing valuable insights for developers operating in modern cloud-native environments.

Chapter 3

Literature Review

3.1 Research Questions

3.2 State of Art

3.3 Conclusions

Chapter 4

Planning

This chapter serves to outline the planning aspects of the project. It begins with the definition of the Project Charter, where it is defined the important aspects of the project, marking it as the starting point. After this is defined, the Work Breakdown Structure (WBS) is defined, leading to an overall view of the project. For the last one a Gantt diagram is presented detailing in a more fine-grand aspects all the tasks necessary to the project.

4.1 Project Charter

The project charter provides a comprehensive overview of the project scope, stakeholders, benefits, assumptions, and restrictions.

Stakeholder

Identification	**Power**	**Interest**
External reviewers who may evaluate the dissertation	Low	Low
Students or developers in the areas of distributed systems and fault tolerance	Low	Low
Administration of the master's program, responsible for dissertation evaluation	Medium	Low
Supervisor (advisor)	High	High

The external reviewers and developers represent an indirect audience with limited direct influence but potential interest in the outcomes. The master's program administration has moderate influence due to institutional requirements, while the supervisor holds the most significant influence and interest, as their guidance and approval are critical for project success.

Benefits

- **Decision Support for Developers:** The project will provide a detailed analysis of fault tolerance aspects in Elixir, Go, and Scala with Akka, offering developers and system architects a practical guide to help them choose the most suitable language for specific fault-tolerant distributed systems scenarios.
- **Open Source Opportunities:** The findings could reveal areas for improvement in the evaluated languages, inspiring open-source developers to create libraries, frameworks, or enhancements tailored to fault tolerance in distributed environments.

- **Academic Contributions:** The dissertation will contribute to the existing body of knowledge in the areas of distributed systems, fault tolerance, and microservices. It will provide insights into comparative aspects in the languages of debate..

Assumptions

- **Computational Resources:** It is assumed that the available computational resources, including hardware and software tools, will suffice to simulate the benchmarking scenarios for each language under realistic system conditions.
- **Support and Guidance:** The supervisor will provide timely and effective feedback on each deliverable, ensuring alignment with project objectives.
- **Consistency Across Languages:** The chosen languages (Elixir, Go, Scala with Akka) have sufficient community support, libraries, and tools to implement the required benchmarking scenarios consistently.

Restrictions

- **Timeframe:** The project must adhere to the deadlines set by the master's program, requiring the completion of all deliverables before the final submission date.
- **Academic Guidelines:** All deliverables, including the dissertation and presentation, must comply with institutional formatting, structure, and content standards.

Risks

4.2 Work Breakdown Structure

The objective of the WBS is to detail the project scope in a visual and hierarchical manner, enabling a clear understanding of how each deliverable connects to the overall project [1]. As shown in Figure 4.1, the focal point of this project is the dissertation. With the objective defined, the first phase focuses on project planning. This phase establishes the foundation by outlining the project charter, creating a WBS, and developing a timeline through a Gantt chart.

Once the planning phase is complete, the subsequent phases align with the Design and Creation research method. This research method was chosen given the nature of the project: while the final objective is clear, there are uncertainties about how to achieve each stage, as every step builds on the outcomes of the previous one. Consequently, the method divides the project into sequential phases: design, implementation, and conclusion. Each phase has clearly defined deliverables that align with the WBS, ensuring that progress can be monitored and adjustments can be made as needed.

The final phase, the conclusion, consolidates all findings and results, translating them into the completed dissertation.

Work Breakdown Structure Dictionary Following is described the WBS Dictionary that as the responsibility of detailing each phase in order to be defined what are the goals and the acceptance criteria in a concise and clear way.

[1] WBS Practice Standard: https://www.projectmanagement.com/deliverables/311939/work-breakdown-structureâĂŞwbsâĂŞpractice-standard-package/ (accessed 30 November 2024)

ÇÇ

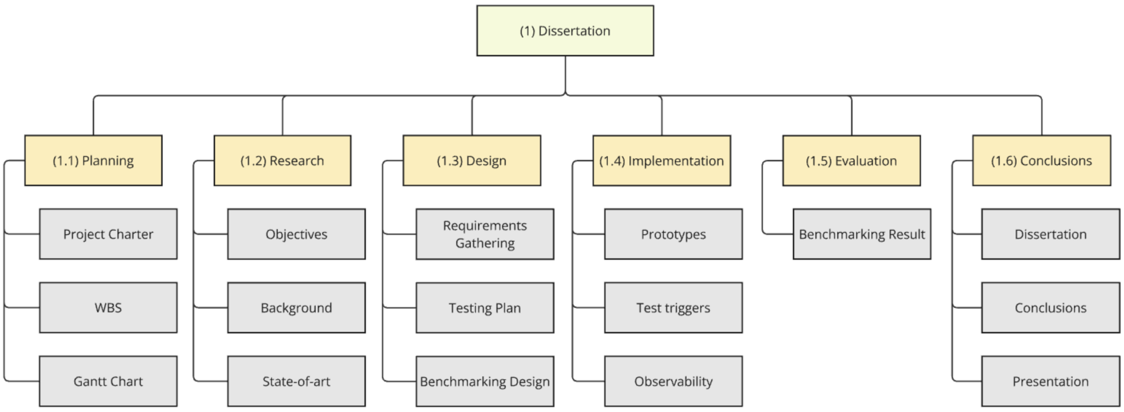

Figure 4.1: The WBS of the project.

Item Name	Type of Item	Additional Description / Acceptance Criteria
(1.1) Planning	Phase	This phase includes all initial project setup tasks.
(1.1.1) Project Charter	Deliverable	The project charter must be created following the project's scope and management guidelines. **Acceptance Criteria:** The project charter must be approved by the supervisor.
(1.1.2) WBS	Deliverable	The WBS should break down the project into manageable components. **Acceptance Criteria:** The WBS should be validated by the supervisor and include all project elements.
(1.1.3) Gantt Chart	Deliverable	A detailed timeline outlining tasks, dependencies, action plan, milestones, and the dissertation deadline. **Acceptance Criteria:** The Gantt chart must accurately reflect project phases and be reviewed by the supervisor.
(1.2) Research	Phase	This phase focuses on gathering the required knowledge and literature to support the project.
(1.2.1) Objectives	Deliverable	Clear objectives for the project, that must detail what are the excepted outcomes. **Acceptance Criteria:** Objectives should align with the research goals and be validated by the supervisor.
(1.2.2) Background	Deliverable	Research and summarize the background of fault tolerance in distributed systems and the distributed and concurrent programming languages. **Acceptance Criteria:** The background section should include sufficient theoretical content approved by the supervisor, and must include a clear justification for the languages chosen.

Item Name	Type of Item	Additional Description / Acceptance Criteria
(1.2.3) State-of-art	Deliverable	Review the current literature on fault tolerance in Elixir, Go, and Scala with Akka. Also, what are the latest techniques for benchmarking distributed and concurrent programming, and if there are already studies on this topic. **Acceptance Criteria:** State-of-the-art review must highlight gaps and relevance to the project scope.
(1.3) Design	Phase	This phase involves requirements gathering, testing plan, and benchmarking design.
(1.3.1) Requirements Gathering	Deliverable	Collect requirements for the benchmarking and evaluation of fault tolerance aspects. **Acceptance Criteria:** Requirements must be detailed, reviewed, and approved by the supervisor.
(1.3.2) Testing Plan	Deliverable	A plan for testing different fault tolerance strategies and mechanisms in Elixir, Go, and Scala with Akka. **Acceptance Criteria:** Testing plan must include scenarios and validation methods, reviewed by the supervisor.
(1.3.3) Benchmarking Design	Deliverable	Define the design for benchmarking environments. **Acceptance Criteria:** Benchmarking environments design must be validated by the supervisor, and must adhere to the test plan created.
(1.4) Implementation	Phase	This phase involves the development of benchmarking prototypes.
(1.4.1) Prototypes	Deliverable	Develop prototypes in Elixir, Go, and Scala with Akka for fault tolerance testing. **Acceptance Criteria:** Prototypes must meet the test plan previously created and must be supported on the benchmarking design planned.
(1.4.2) Test Triggers	Deliverable	Create fault injection mechanisms for testing fault tolerance. **Acceptance Criteria:** Fault injection methods must simulate real-world scenarios and be validated by tests.
(1.4.3) Observability	Deliverable	Implement observability tools for monitoring system behavior during tests. **Acceptance Criteria:** Observability setup must capture the metrics defined on the test validations methods.
(1.5) Evaluation	Phase	Evaluate the results of the benchmarking tests.
(1.5.1) Benchmarking Result	Deliverable	Analyze and document the outcomes of benchmarking fault tolerance aspects. **Acceptance Criteria:** Results must be clear, reproducible, and reviewed by the supervisor.

Item Name	Type of Item	Additional Description / Acceptance Criteria
(1.6) Conclusions	Phase	Finalize and present the results of the dissertation.
(1.6.1) Dissertation	Deliverable	Compile the dissertation document with findings and analyses. **Acceptance Criteria:** Dissertation must meet university formatting and content guidelines.
(1.6.2) Conclusions	Deliverable	Write concise conclusions summarizing key findings from the research, with the goal of creating a guide for future developers consult. **Acceptance Criteria:** Conclusions must be concise and detail what are the cons and pros of using each language for each specific case, so that develops can easily decide.
(1.6.3) Presentation	Deliverable	Prepare and deliver the final presentation to the evaluation committee. **Acceptance Criteria:** Presentation must be clear and precise.

4.3 Gantt Diagram

4.4 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] Roberto Vitillo. *Understanding Distributed Systems: What every developer should know about large distributed applications*. 2021. isbn: 1838430202.

[3] 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.

[4] Mohamed. Amroune, Makhlouf. Derdour, and Ahmed. Ahmim. *Fault Tolerance in Distributed Systems: A Survey*. IEEE, 2018. isbn: 9781538642382.

[5] George Coulouris et al. *Distributed Systems - Concepts and Design*. 2012. isbn: 978-0-13-214301-1.

[6] 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.

[7] Atlassian. *Reliability vs availability: Understanding the differences*. Accessed at 16.10.2024. url: https://www.atlassian.com/incident-management/kpis/reliability-vs-availability.

[8] 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.

[9] 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.

[10] Martin Kleppmann. *Designing Data Intensive Applications*. 2017. isbn: 978-1449373320.

[11] 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.

[12] AWS - Amazon. *Challenges with distributed systems*. Accessed at 03.11.2024. url: https://aws.amazon.com/builders-library/challenges-with-distributed-systems/.

[13] 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.

[14] IBM. *What is the CAP theorem?* Accessed at 03.11.2024. Aug. 2024. url: https://www.ibm.com/topics/cap-theorem.

[15] Seth Gilbert and Nancy A Lynch. *Perspectives on the CAP Theorem*. 2012. doi: 10.1109/MC.2011.389.

[16] Ahmad Shukri Mohd Noor, Nur Farhah Mat Zian, and Fatin Nurhanani M. Shaiful 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.

[17] 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.

[18] 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.

[19] Joe Armstrong. *Early Praise for Programming Erlang, Second Edition*. 2013. isbn: 978-1-937785-53-6.

[20] Jan Henry Nystrom. *Fault Tolerance in Erlang*. 2009. isbn: 978-91-554-7532-1.

[21] 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. doi: doi/10.5555/1624775.1624804.

[22] 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.

[23] 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.

[24] 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.

[25] C. A. R. Hoare. "Communicating sequential processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. issn: 0001-0782. doi: 10.1145/359576.359585. url: https://doi.org/10.1145/359576.359585.

[26] 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.

[27] Go. *Official documentation of Go programming language*. Accessed at 10.11.2024. url: https://go.dev/doc//.

[28] 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.

[29] Pooyan Jamshidi et al. "Microservices: The Journey So Far and Challenges Ahead". In: *IEEE Software* 35.3 (2018), pp. 24–35. doi: 10.1109/MS.2018.2141039.

[30] Claudio Guidi et al. "Microservices: A language-based approach". In: Springer International Publishing, Nov. 2017, pp. 217–225. isbn: 9783319674254. doi: 10.1007/978-3-319-67425-4_13.

[31] Saša Jurić and Francesco Cesarini. *Elixir in Action, Third Edition*. 2024. isbn: 9781633438514.

[32] Francisco Lopez-Sancho Abraham. *Akka in Action, Second Edition*. Simon and Schuster, 2023. isbn: 9781617299216.

[33] Ivanilton Polato et al. *A comprehensive view of Hadoop research - A systematic literature review*. 2014. doi: `10.1016/j.jnca.2014.07.022`.