ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Comparative Analysis of Fault Tolerance in Elixir and Other Distributed and Concurrent Languages

Nuno Ribeiro

Dissertation preparation report for the degree of Master of Science, Area of Software Engineering

Advisor: Dr. Luís Nogueira

Porto, December 31, 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 31, 2024

Abstract

Fault tolerance is a critical component of distributed systems, particularly in light of the exponential growth in internet usage and system scale in recent years. The resilience and fault tolerance capabilities of these systems are essential for maintaining reliability and minimizing downtime. Elixir, with its fault-tolerant features and its foundation in Erlang's "let it crash" philosophy, stands out as a robust tool for building such systems.

Nevertheless, other distributed and concurrent programming languages also offer solutions, often leveraging similar concepts in different environments. For instance, Akka, inspired by the "let it crash" philosophy, provides a toolkit built for Scala. While both languages share conceptual foundations, Scala operates in a different environment with unique characteristics. Meanwhile, Go, a language gaining widespread popularity for its strong concurrency mechanisms, offers another compelling option for building concurrent and fault-tolerant systems.

Although all these languages can theoretically employ the Actor Model through libraries and toolkits, Elixir integrates these concepts natively, potentially providing distinct advantages. Conversely, Scala and Go include features that are not inherently available in Elixir's Open Telecom Platform (OTP). Moreover, Go's lack of native support for distributed systems raises questions about whether the advantages of using a widely adopted language can limiting on distributed computing compared with specific focus languages.

To investigate these assumptions and evaluate the comparative strengths and weaknesses of each language, benchmarking is necessary. Based on a review of existing literature, the most effective methodology, for this case, involves designing a generic application that simulates distributed operations sensitive to fault tolerance and resilience. This application will enable the creation of controlled test scenarios to yield measurable results, offering insights into the performance of each language in building fault-tolerant systems.

Future work will focus on implementing the benchmarking strategies, collecting performance, fault tolerance, and resilience metrics, as well as static code metrics. These efforts will be supported by detailed project planning to ensure the successful execution and meaningful outcomes of the project.

Keywords: Distributed Systems, Fault Tolerance Strategies, Actor Model, Benchmarking

Resumo

A tolerância a falhas é um componente fundamental dos sistemas distribuídos, especialmente tendo em conta o grande crescimento da utilização da Internet e a crescente necessidade de sistemas mais robustos nos últimos anos. As capacidades de resiliência e tolerância a falhas desses sistemas são essenciais para garantir a fiabilidade e minimizar o tempo de inatividade devido a falhas. Elixir, com as suas características de tolerância a falhas e a sua base na filosofia *"let it crash"* de Erlang, destaca-se como uma ferramenta poderosa para a construção desses sistemas. Elixir herda todas as capacidades do Erlang, assim como da sua *virtual machine*, sendo projetado para garantir alta disponibilidade nos diversos sistemas em que é utilizado, como, por exemplo, em bancos e telecomunicações.

Contudo, outras linguagens de programação distribuídas e concorrentes também oferecem soluções, muitas vezes aproveitando conceitos semelhantes em ambientes distintos, ou até mesmo inspirando-se na solução oferecida por Elixir, adaptando-a aos seus próprios conceitos, ou utilizando abordagens diferentes mas com objetivos semelhantes. Por exemplo, o Akka, inspirado pela filosofia *"let it crash"*, fornece um conjunto de ferramentas criado para Scala, permitindo imitar o comportamento visto em Elixir com o uso de padrões *supervisor*. Embora ambas as linguagens compartilhem a mesma filosofia, o Scala opera num ambiente diferente, com características únicas, funcionando sobre a *Java Virtual Machine* (JVM). Por outro lado, Go, uma linguagem que tem ganho popularidade devido aos seus mecanismos de concorrência robustos e ao seu uso em vários projetos de grande escala, adota uma abordagem diferente e oposta quanto ao controlo de erros, sendo mais explícita e rejeitando a ideia de que os erros devem ser encarados como inevitáveis, sem a necessidade de um controlo tão refinado dentro dos componentes.

Embora todas estas linguagens possam teoricamente empregar o Modelo de Ator através de bibliotecas e kits de ferramentas, como o Akka em Scala e o Proto-Actor em Go, Elixir integra esses conceitos de forma nativa, o que pode oferecer vantagens distintas. Por outro lado, Scala e Go incluem funcionalidades que não estão presentes de forma nativa na *Open Telecom Platform* (OTP) do Elixir. Além disso, a falta de suporte nativo de Go para sistemas distribuídos levanta questões sobre a conveniência de utilizar uma linguagem amplamente adotada, mas com limitações na computação distribuída, quando comparada com linguagens mais específicas para esse fim. Assim, o objetivo é perceber se, para a criação de um sistema tolerante a falhas, a utilização de uma linguagem nativa como o Elixir compensa em comparação com a utilização de uma linguagem mais genérica, mas com maior popularidade.

Para investigar estes pressupostos e avaliar as vantagens e desvantagens de cada linguagem, será necessário realizar uma avaliação comparativa através de *benchmarking*. Com base numa revisão da literatura existente, a metodologia mais eficaz para este caso envolve a criação de uma aplicação genérica que simule operações distribuídas sensíveis à tolerância a falhas e resiliência. Esta abordagem, combinando diferentes métodos utilizados na literatura, permitirá a criação de cenários de teste controlados em sistemas que tentam imitar processos reais, gerando resultados mensuráveis.

O trabalho futuro concentrar-se-á na implementação de um *benchmarking* detalhado, na análise e recolha de métricas, bem como na apresentação das conclusões. A abordagem proposta envolve o desenvolvimento de uma aplicação de chat que suporte funcionalidades básicas, sendo implementada nas três linguagens selecionadas. No caso da linguagem Go, serão realizadas duas implementações: uma utilizando o *Actor Model* através do *toolkit* Proto-Actor e outra adotando abordagens nativas para distribuição e mecanismos de tolerância a falhas próprios da linguagem e de bibliotecas externas.

A execução desta análise permitirá a obtenção de métricas relacionadas ao desempenho, tolerância a falhas e resiliência, complementadas por uma análise estática do código. Estes esforços serão conduzidos com o apoio de um planeamento rigoroso, no qual todas as fases do projeto serão detalhadamente descritas e os entregáveis devidamente definidos, garantindo assim uma execução bem-sucedida e a obtenção de resultados significativos para o projeto.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Problem

In the rapidly evolving landscape of software development, fault tolerance and resilience have become critical attributes for building robust, scalable systems [1]. Fault tolerance ensures that systems can continue operating despite failures, while resilience enables them to recover gracefully, minimizing downtime and data loss. The importance of these characteristics has grown significantly with the increasing prevalence of distributed systems and cloud computing [1, 2].

Elixir, a functional programming language built on the Erlang Virtual Machine (VM) called Bogdan/Björn's Erlang Abstract Machine (BEAM), has gained significant popularity for its "let it crash" paradigm, a philosophy that emphasizes process isolation, supervision trees, and fault recovery [3–5]. This design approach, which originates from Erlang's legacy in telecom systems, positions Elixir as a strong candidate for developing fault-tolerant systems. At the same time, other distributed and concurrent programming languages offers alternative approaches to addressing fault tolerance, each with distinct methodologies and advantages tailored to different use cases and developer communities.

Despite the recognized importance of fault tolerance in software systems, there is a lack of comprehensive, up-to-date research that directly compares the fault tolerance and resilience aspects of Elixir with other programming languages. This gap in comparative analysis makes it difficult for software developers and architects to make informed decisions when selecting a language or framework for building fault-tolerant applications.

1.2 Objectives

The primary goal of this dissertation is to study Elixir's fault tolerance in comparison with other distributed and concurrent programming languages. Specifically, it aims to determine which languages provide the best support for fault tolerance mechanisms and identify the most suitable language for implementing common techniques in fault-tolerant system design. The objectives of this study are as follows:

- Comprehensively analyze the fault-tolerant mechanisms in Elixir, including its design paradigms, implementation strategies, and practical applications.
- Identify the most popular and relevant distributed and concurrent programming languages for comparison and investigate their fault-tolerant mechanisms.

- Compare Elixir's fault-tolerant capabilities with those of other languages to elucidate their respective strengths, weaknesses, and trade-offs.
- Conduct benchmarking experiments to empirically evaluate and compare the fault tolerance and resilience of Elixir against other distributed and concurrent programming languages, providing quantitative data to support the analysis.
- Extract best practices and propose potential improvements in fault-tolerant system design across the analyzed languages.

1.3 Ethical Considerations

Ethical considerations play a crucial role in software engineering research, ensuring the integrity, transparency, and societal relevance of the work. This section outlines the ethical principles applied.

Transparency and Fairness in Results. As this research is focused on the comparison of programming languages, it is essential to maintain impartiality and avoid any bias in the results. Research integrity demands that results are not manipulated or altered to provoke more appealing discussions or gain community approval [6]. This dissertation adheres to the principle of transparency, ensuring that the benchmarking results reflect the true performance of each language.

Replicability and Verification. Replication is a important aspect of scientific research, enabling others to validate findings [7]. This dissertation involves the development of prototypes and proof-of-concepts to evaluate specific fault tolerance strategies. To uphold ethical standards, all tests and configurations will be documented on a public repository to allow replication.

Adherence to Professional Codes of Ethics. This work adheres to the ethical principles outlined in the Institute of Electrical and Electronics Engineers (IEEE) Code of Ethics [8] and the Association for Computing Machinery Code (ACM) of Ethics and Professional Conduct [9], which emphasize integrity, respect, fairness, and authorized use of content. Researchers must act responsibly by avoiding any practices that could harm the reputation or fairness of the comparison, maintaining an ethical commitment to the broader community of developers and researchers.

Avoidance of Plagiarism and Proper Citation. Plagiarism undermines the credibility and value of academic work. In alignment with the Code of Good Practices and Conduct of Polytechnic of Porto, particularly Article 10, this dissertation ensures proper attribution of all referenced works. Accurate citation is fundamental to acknowledge the contributions of others, demonstrate the research's academic honesty, and respect intellectual property.

1.4 Document Structure

The document is organized as follows:

Introduction. An initial overview of the dissertation's scope is provided, including a discussion of the context, the problem statement, and the goals the study aims to achieve.

Background. Foundational knowledge supporting the context of the dissertation is presented here. The section is divided into three parts: the first examines the general aspects of distributed systems, highlighting their characteristics and theoretical foundations. The second part delves into fault tolerance, exploring the strategies employed and the theoretical principles involved. The final part conducts a study of distributed and concurrent programming languages, offering a list of relevant options and a justification for the specific languages selected for analysis.

State of the art. Current insights into the themes explored in the dissertation are presented in this chapter. It opens with the research questions to be investigated and the research methodology to be used. The research questions are divided into two sections: the first investigates the fault tolerance mechanisms of Elixir, Scala with Akka, and Go, which were identified in the background as the primary languages of interest. The chapter concludes with a discussion of benchmarking strategies from the literature to support future work. This is followed by a brief high-level outline of the practical aspects of future work.

Project Planning. The planning and management aspects of the dissertation are outlined, including the project charter, a Gantt chart, and a Work Breakdown Structure to guide the execution of the project.

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 [2]. 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 [10].

To the end-user, distributed systems appear as a single, large virtual system, making the underlying logic transparent [10]. These systems achieve a shared objective by transmitting messages through various nodes and dividing computational tasks among them, increasing resilience and isolating business logic [10, 11]. 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 [2, 11].

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 [2, 12]. 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) [2, 13]. Replication transparency ensures reliability by masking data duplication, while failure transparency enables systems to handle faults without user disruption [2, 13]. 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 [14]. A highly reliable system maintains consistent, uninterrupted service over an extended period, minimizing disruptions for users [2], 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 [2, 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 [2, 5, 10]. 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 [16]. Cloud services must support a large volume of simultaneous connections and interactions, making scalability a crucial factor [2].

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 [2]. 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 [11].

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 [1].

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 [10]. Synchronous communication involves blocking operations where the sender waits for a response, suitable for scenarios requiring confirmation [2, 13]. 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 [2].

2.1.3 Challenges

Distributed systems encounter numerous challenges, including scalability [14], managing software, network, and disk failures [17, 18], heterogeneity [13], coordination among nodes, and difficulties on debugging and testing [10, 18]. 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 [2, 10]. This theorem underlines a critical trade-off in distributed systems: only two of these properties can be fully ensured at any given time [19]. 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 [2, 10, 19]. This concept is directly relevant to this dissertation, as fault tolerance strategies discussed later, like replication and redundancy, 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 [11]. 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, 20].

2.2.1 Fault Tolerance Taxonomy

Firstly, 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. [21]. A fault, which types are summarized in the Table 2.1, is defined as an underlying defect within a system component that can lead to a failure, 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 [21, 22].

Type of Fault	**Description**
Transient Faults	Temporary conditions like network issues or service unavailability. Can typically be resolved by restarting the application when the underlying condition is fixed [21].
Intermittent Faults	Unpredictable symptoms related to system or hardware malfunction. Difficult to detect during testing and emerge during system operation, and also hard to completely resolve [21].
Permanent Faults	Persistent issues that continue until the root cause is identified and addressed. Relatively straightforward to fix, typically related to complete component malfunction [2].
Byzantine Faults	Caused by internal system state corruption or incorrect network routing. Handling is complex and costly, often requiring multiple component replicas and voting mechanisms [21].

Table 2.1: Brief description of fauls types

Failures are the external manifestations of the 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 [2].

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. [2]
Omission Failure	The system fails to send or receive necessary messages, impacting communication and task coordination. [21]
Timing Failure	The system's response occurs outside a specified time interval, either too early or too late, causing issues in time-sensitive operations. [21]
Response Failure	The system provides incorrect outputs or deviates from expected state transitions, potentially leading to wrong results. [2]
Arbitrary Failure	The system produces random or unpredictable responses at arbitrary times, potentially with incorrect data. This type of failure is challenging to diagnose and manage. [2]

Table 2.2: 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 that are 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 Mechanism

The retry mechanism is a widely adopted and straightforward technique that involves reattempting a failed operation under the assumption that transient faults may resolve over time [12]. Despite its simplicity, this strategy is highly suitable in many scenarios, particularly when implementing more complex fault tolerance mechanisms would introduce unnecessary cost in environments with a high likelihood of transient faults. However, it is crucial to recognize that retrying in the case of a permanent error is pointless. Moreover, if the failure is caused by system overload, uncontrolled retries can aggravate the issue. To address these challenges, implementing a maximum retry limit and incorporating strategies such as exponential backoff, where retries are spaced out with increasing delays, becomes essential [1, 10].

This approach operates by attempting the operation a predefined number of times or until a set timeout is reached. If the retries ultimately fail, the system can fallback on alternative measures, such as logging the failure, invoking a fallback operation, or redirecting the request to another asset [21]. These measures ensure that in the event of a persistent fault, the system will make controlled attempts and try to fallback in a safe manner.

The retry mechanism's simplicity and low implementation overhead make it ideal for scenarios where the cost of a retry is negligible compared to the complexity of alternative solutions. It is particularly effective in network communication, where transient issues such as dropped packets or server unavailability often resolve with subsequent attempts. In case of an unresolved situation it is created an omission failure due to the missing communication. Additionally, it is well-suited for database systems to handle transient locking or deadlock conditions, as well as in microservice architectures, where downstream services may temporarily become unresponsive but recover shortly thereafter [1].

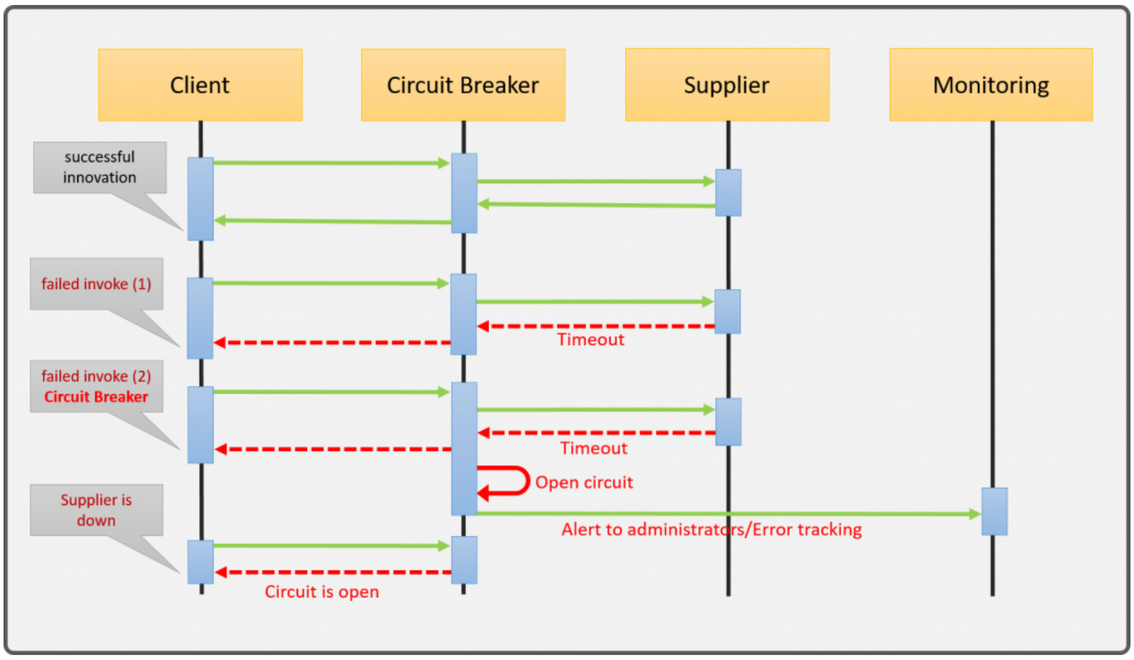


Figure 2.1: Diagram illustrating the states of a Circuit Breaker [1].

This strategy aligns effectively with Actor Models due to their inherent monitoring capabilities, which detect errors and initiate retries automatically. Frameworks such as Akka with Scala have built-in support for this mechanism [21].

Circuit Breaker Pattern

The Circuit Breaker pattern, inspired by electrical circuits, is designed to prevent the failure of a single subsystem from cascading and compromising an entire system. This pattern tries to maintain the overall system stable by isolating failing components [10]. By actively monitoring the health of operations and selectively blocking problematic ones, circuit breakers act as safeguards against system overload and degradation [23].

Circuit breakers operate in three primary states: Closed, Open, and Half-Open [10, 23]. In the Closed state, illustrated in Figure 2.1 by the first request, operations proceed as usual, with all requests passing through the circuit breaker while it monitors for potential failures. When failures exceed a predefined threshold within a specified time window, whether measured as a count or a percentage of failed attempts, the circuit breaker transitions to the Open state, illustrated in Figure 2.1 by the third request. In this state, all requests are blocked to prevent higher pressure on the failing subsystem. During this time, it is essential to issue an alert to monitoring systems to ensure operational visibility [10, 23]. After a cool-down period, the circuit breaker moves to the Half-Open state, where it permits a limited number of test requests to verify if the underlying issue has been resolved. If these test requests succeed, the circuit breaker resets to the Closed state and resumes normal operation. Otherwise, it reverts to the Open state [23].

[1]From Oscar Blancarte Blog. `https://www.oscarblancarteblog.com/2018/12/04/circuit-breaker-pattern/` (accessed 8 December 2024).

The Circuit Breaker pattern is particularly well-suited for distributed systems, such as microservice architectures, where dependencies on external services can lead to cascading failures [23]. For instance, if a downstream service becomes unresponsive, the circuit breaker blocks further requests, providing the service with time to heal and avoiding the risk of overloading it with retries [10]. It is equally effective in scenarios involving a third-party Application Programming Interface (API), where temporary rate limits or outages can impact availability. In database systems, circuit breakers can mitigate the effects of resource contention or extended downtime by isolating problematic queries, ensuring the broader system remains operational.

When compared to pure retry mechanisms, the Circuit Breaker pattern provides a more sophisticated approach to fault tolerance. While retries focus on recovering from transient faults, they can harm even more issues under conditions such as system overload or persistent failures [10]. In contrast, circuit breakers proactively block failing operations, reducing the risk of cascading failures and preserving overall system stability. This type of isolation makes circuit breakers an valuable strategy in building fault-tolerant systems.

Replication and Redundancy

Replication is a fundamental strategy for achieving fault tolerance in distributed systems and is widely used across various domains [11]. By creating multiple replicas of data or processes, replication eliminates single points of failure, ensuring system reliability, availability, and transparency [13]. This approach allows a system to tolerate faults by introducing redundancy, which distributes operations across a group of replicas rather than relying on a single vulnerable node [2].

To effectively coordinate replicas and maintain consistency, replication mechanisms employ various strategies, which can be categorized as follows [21]:

- **Active Replication (Semi-Active):** In this strategy, all replicas process incoming requests simultaneously, and the system relies on consensus algorithms to maintain consistency among the results.
- **Passive Replication (Semi-Passive):** One replica, designated as the leader or primary, handles all client requests and updates other replicas with state information. In case of a primary failure, backup replicas are promoted or synchronized to restore the system's functioning.
- **Passive Backup (Fully Passive):** Replicas act as standby backups in this approach. A backup replica is only activated when the primary fails, minimizing overhead during normal operation.

These replication strategies align with the principles of the CAP theorem, which states that a distributed system can guarantee at most two of the following three properties: consistency, availability, and partition tolerance. Replication strategies often emphasize availability and partition tolerance, potentially compromising consistency due to the inherent challenges of achieving consensus and synchronizing data across replicas. Nonetheless, this trade-off enables systems to scale, increase availability, and provide transparency to end users [1].

Consensus Algorithms

Achieving consensus is essential in distributed systems to ensure that a group of processes operates cohesively as a single entity [2]. Consensus algorithms enable replicas to agree on a shared state or a sequence of operations, even in the presence of faults. Two famous used consensus algorithms in distributed systems are Raft and Paxos [2].

Paxos

Paxos appeared in 1989 and has evolved over time, earning a reputation as a complex and difficult to understand algorithm [2]. Due to its challenges and the availability of newer alternatives, such as Raft, that is described bellow. This Paxos explanation will focus on its core concepts without delving into exhaustive details.

Paxos ensures that a group of distributed replicas agrees on a single value, even in the presence of faults. It operates under challenging conditions: replicas may crash and recover, messages can be delayed, reordered, or lost, and no assumptions are made about message delivery timing [2, 24]. The algorithm revolves around three distinct roles [13, 24]:

- **Proposers:** Suggest values for the system to agree upon.
- **Acceptors:** Vote on proposals, ensuring fault tolerance by requiring a majority for decisions.
- **Learners:** Observe the final agreed value and disseminate the result across the system.

With the roles defined, the Paxos algorithm progresses through the following phases to achieve consensus [2, 13, 24]:

- **Prepare Phase:** A proposer generates a proposal with a unique sequence number and sends a *prepare request* to a list of acceptors.
  - Acceptors respond with a *promise* not to accept earlier proposals.
  - If an acceptor has already accepted a value, it shares this value with the proposer.
- **Accept Phase:** Based on responses from the prepare phase, the proposer sends an *accept request* with a value:
  - If an acceptor had previously accepted a value, the proposer adopts that value.
  - Otherwise, the proposer chooses its own value.
  - Acceptors respond by accepting the value if it doesn't conflict with their earlier promises.
- **Commit or Learn Phase:** Once a majority of acceptors accepts the value, consensus is achieved.
  - The proposer informs all replicas, which then commit the value.

While Paxos is robust in theory and guarantees consistency, its complexity and subtle behaviors make it difficult to implement correctly or faithfully to its original design [2]. Over time, new variations and extensions, such as Multi-Paxos, have been developed. Multi-Paxos enables the system to achieve consensus on multiple values, making it more practical for real-world applications, like Chubby lock service of Google [13], but in general it is not considered a highly adopted algorithm.

The inherent complexity of Paxos has also driven the creation of simpler consensus algorithms, such as Raft, which aim to provide the same guarantees while being easier to understand and implement [2, 24].

Raft

Raft is a consensus protocol designed to enable fault-tolerant operation in distributed systems. It ensures that a process will eventually detect if another process has failed and take appropriate corrective action. Raft was developed as a more comprehensible and practical alternative to Paxos, addressing its complexity and promoting clarity [2, 25]. This algorithm fits on semi passive replication strategy.

Each process in Raft maintains a log of operations, which may include both committed and uncommitted entries. The primary goal of Raft is to ensure that these logs remain consistent across all servers, such that committed operations appear in the same order and position in every log [2]. To achieve this, Raft uses a leader-based approach, where one server assumes the role of leader while the remaining servers act as followers. The leader is responsible for determining the sequence of operations and ensuring their consistent replication [10]. The typical number of nodes is five, which theoretically allows for two failures [25].

When a operation request is submitted, the leader appends the operation to its log as a tuple where it contains: the operation to be executed, the current term of the leader, and the index of the operation in the leader's logs [2]. The term is reset every time an election occurs, starting from zero [25]. This information is then propagated to the followers using a process inspired by the two-phase commit protocol, where it consists on [2, 10]:

1. **Append Phase:** The leader sends the new log entry to all followers. The followers append the entry to their logs and send an acknowledgment signal back to the leader.
2. **Commit Phase:** Upon receiving the acknowledgments from a majority of followers, the leader marks the entry as committed, executes the operation, updates its state, and notifies the client of the result. At the same time, the leader informs all followers of the commitment, ensuring their logs reflect the updated status.

This two-step process guarantees that committed entries are replicated on a majority of servers, preserving durability and consistency, even in case of server failures [2]. However, there are cases where the leader fails and an election starts among the followers. The followers acknowledge the leader's failure through the heartbeat strategy, where after a certain time without receiving any signal sent by the leader, the follower starts an election [10, 25], like represented on Figure 2.2 by the change of state from follower to candidate. To prevent multiple followers from initiating elections simultaneously, heartbeat timeouts are randomized [2, 10].

The change of state of the node is displayed on the Figure 2.2 and the process consists on the following steps [10, 25]:

1. **Transition to Candidate:** A follower transitions to a candidate state, increments its term number, and broadcasts requests for votes from other servers.
2. **Voting:** Each server can vote for one candidate per term. A server grants its vote only if the candidate's log is at least as complete as its own, ensuring that the elected leader has the most up-to-date log.
3. **Leader Selection:** If a candidate receives votes from a majority of servers, it becomes the leader for the current term.

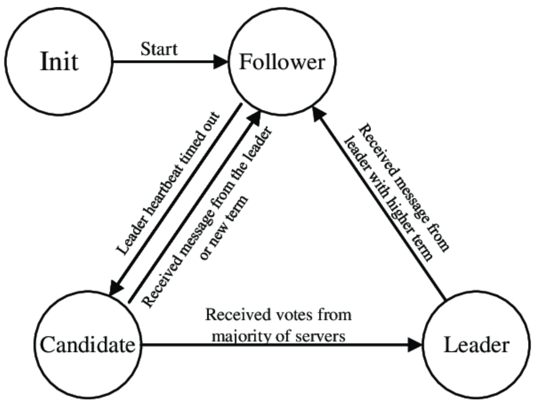

Init
Start
Follower
Leader heartbeat timed out
Received message from the leader or new term
Received message from leader with higher term
Candidate
Received votes from majority of servers
Leader

Figure 2.2: Diagram illustrating the states of a node in the Raft algorithm [2].

Once elected, the new leader reconciles any inconsistencies by broadcasting missing log entries to followers during subsequent operations, ensuring consistency across the cluster [2].

Raft's structured and modular design prioritizes simplicity, reliability, and fault tolerance. Its leader-based model centralizes decision-making and log synchronization, while its robust mechanisms for log replication and leader election ensure consistency and availability even in the face of failures.

Comparison of Paxos and Raft

Both Paxos and Raft are leader-based distributed consensus algorithms designed to ensure consistency in replicated state machines [24]. The key difference lies in their approach to leader election and log replication. Raft prioritizes simplicity and readability, requiring candidates to have up-to-date logs before becoming leaders, avoiding complex log synchronization steps common in Paxos. Paxos, while general and flexible, allows for greater complexity, such as out-of-order log replication and term adjustments during leadership changes [2].

Raft is widely used in distributed systems requiring high availability and consistent state management. Notable applications use adaptations of Raft, including etcd for distributed key-value storage, Consul for service discovery, and CockroachDB for scalable, consistent databases [10, 24]. Paxos, on the other hand, is often found in legacy systems and specialized applications like Google's Chubby service [24].

Check-pointing and Message Logging

When an error compromises the system's state, recovery actions must be taken. These strategies focus on fault recovery by addressing errors after they occur. Their primary goal is to restore the system to an error-free state. Recovery strategies are generally classified as either backward recovery or forward recovery [2].

Forward recovery seeks to return the system to a correct state after it has entered an erroneous one. However, this approach requires prior knowledge of potential errors to execute fixes, which can be challenging or even unreliable [2]. Alternatively, backward recovery involves periodically saving the system's state and restoring it to the last known correct

[2]Adapted from Jinjie Xu and colleagues. `https://doi.org/10.3390/sym14061122/` (accessed 8 December 2024).

state when issues arise. This approach uses checkpoints, which are recorded snapshots of the system state that enables the recovery process [2].

Check-pointing is a backward recovery technique. It periodically saves the state of a process, enabling it to restart from the last saved state in the event of a failure. However, this approach is computationally expensive and introduces performance overhead [2, 21]. Furthermore, some operations are inherently irreversible, which limits the effectiveness of this strategy [2]. Despite these challenges, check-pointing is a popular option for restoring the system to a known correct state where it is the reasonable strategy to be applied [12, 21].

To address the performance overhead of check-pointing, a lighter approach, message logging, has been developed. This strategy involves maintaining a log of action messages of the system. By replaying these messages in the correct order, the system can recover to a consistent state without the need to record its entire state continuously [12, 21]. Although checkpoints are still required to avoid replaying all messages from the beginning, the overhead is significantly reduced compared to traditional check-pointing [21]. For this strategy to work effectively, system operations must be deterministic, ensuring that replaying messages reproduces the correct state, which is also a limitation or even an impossibility in some cases [2].

Apache Flink employs check-pointing to recover state and stream positions, ensuring that the application maintains consistent semantics even in the event of failures, as outlined in the project's documentation [26]. However, these strategies can be resource-intensive, and it is crucial to carefully evaluate their suitability. In database environments where data consistency and order are critical it could be justified.

Briefly Comparison Between Strategies

Fault tolerance is essential to ensure reliability and performance. Four key strategies have been explored, each suitable for specific scenarios of varying complexity. Retry mechanisms are a simple but effective way of dealing with transient failures such as network interruptions. For example, in an e-commerce platform, retrying a failed payment gateway request can often resolve temporary connectivity issues. In microservices, retries can be enhanced with circuit breakers that temporarily halt requests to unstable services to prevent cascading failures, such as when a downstream service becomes overloaded. For critical scenarios where fault transparency is essential, replication is more appropriate. For example, in distributed databases, replication ensures data availability even if a node fails, providing resilience for applications such as online banking or real-time analytics. Check-pointing and message logging, on the other hand, are ideal for systems where restarting is costly or inefficient. In resource-intensive processes such as Artificial Intelligence (AI) model training or large-scale simulations, check-pointing could allow the system to recover from the last point of progress, rather than starting from scratch and losing all data.

2.3 Distributed and Concurrency Programming

Distributed and concurrent programming languages plays a important role in building resilient and fault-tolerant systems [27]. 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 [28], 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 [27]. This has driven the evolution of new paradigms, languages, frameworks, and libraries aimed at reducing development complexity in distributed and concurrent systems [5].

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 [29]. 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 [30]. Each actor is equipped with its own mailbox, which receives messages and processes them sequentially [31].

A core principle of the Actor Model is isolation, maintaining their own internal state that is inaccessible and immutable by others [31]. This eliminates the need for shared memory, reducing complexity and potential data races [5].

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 [30].

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 [32]. 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 [30]. 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 [33].

CSP offers an abstract and formal framework for modeling interactions between concurrent processes through channels, which serve as the communication medium between them [34].

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 [33]. 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 [4, 5, 34]. In addition, the language Occam attempts to offer a direct implementation of CSP principles with its focus on critical projects such as satellites [35].

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 [36]. 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 [36].

Microservices architectures allow general purpose programming languages to participate in distributed computing paradigms by leveraging frameworks, libraries, and microservices principles [37].

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 [5, 34]. 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 [5].

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 [5]. Erlang, for instance, is renowned for its fault tolerance and "let-it-crash" philosophy [27]. Supervising actors monitor and recover from failures, making Erlang highly suitable for building robust distributed systems [27]. 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 [3].

Haskell, a pure functional programming language, provides a deterministic approach to concurrency, ensuring consistent results regardless of execution order [5]. Its extension Cloud

Haskell[3], 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 [5]. Unlike Erlang, Akka operates on the Java Virtual Machine (JVM), providing seamless interoperability with Java-based systems [38].

Go, developed by Google, simplifies concurrent programming through its lightweight goroutines and channels, inspired by the CSP paradigm, which abstracts threading complexities [35]. 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 [4].

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 [39].

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[4] and Tiobe November 2024[5].

Conversely, some relatively recent languages have gained attention. Unison[6] employs content-addressed programming using hash references to improve code management and distribution. Gleam[7] compiles to Erlang and offers its own type-safe implementation of Open Telecom Platform (OTP), Erlang's actor framework. Pony[8], 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 2.3, 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[9] 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 also know as Erlang VM [3]. Since Elixir inherits all the strengths of Erlang [5], 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.

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

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

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

[6]Unison: `https://www.unisonweb.org/` (accessed 27 November 2024)

[7]Gleam: `https://gleam.run/` (accessed 27 November 2024)

[8]Pony: `https://www.ponylang.io/` (accessed 27 November 2024)

[9]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 2.3: Characteristics of distributed and concurrent programming languages

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 [5], 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 [38]. Additionally, Scala with Akka is notable for its community acceptance [5]. 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 2.3 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 [35]. By examining how Go can achieve fault tolerance using libraries and abstractions under a microservices strategy, for example, 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

State of the art

3.1 Research Questions

For the literature review and the state of the art analysis, two research questions have been formulated to address the essential background and focus of this dissertation. The first research question centers on the fault tolerance mechanisms of each language under study. The second research question examines the benchmarking strategies that simulate distributed systems, including microservices, and identifies the metrics necessary to measure the fault tolerance aspects.

- **RQ1:** How do the programming languages Elixir, Scala with Akka, and Go implement fault tolerance mechanisms in distributed systems, and what are the comparative strengths, weaknesses, and trade-offs of each approach?
- **RQ2:** What are the most effective benchmarking strategies for distributed environments focusing on fault tolerance aspects?

3.1.1 Research Methodology

This section outlines the research methodology adopted in this dissertation. It is important to note that this study addresses, to the best of our knowledge, a gap in the literature regarding a direct comparison of fault tolerance mechanisms in the programming languages Elixir, Scala with Akka, and Go. To the best of our knowledge, no prior study has evaluated these three languages side-by-side in fault tolerance test scenarios.

Relevant work can be found in the study by Valkov et al. [5], which compared Erlang, Go, and Scala in terms of IPC latency, process creation time, the maximum number of supported processes, and throughput. Another study by Randtoul and Trinder [32] shares a similar intent, focusing on reliability, however, it excludes Go and does not encompass a comprehensive range of test scenarios, concentrating primarily on message throughput rather than other metrics. While these studies provide valuable insights, they do not explore fault tolerance test scenarios, which is the primary focus of this dissertation.

To explore the existing body of knowledge and identify studies relevant to the three programming languages, searches were conducted in major academic databases, specifically IEEE Xplore and ACM Digital Library. The objective was to find articles that included the three languages in their titles, allowing for Erlang to be considered in place of Elixir. The search was initiated with a starting date of 2013, coinciding with the publication of the second edition of Joe Armstrong's book on Erlang. However, the search yielded no results in the IEEE Xplore database, while the ACM Digital Library returned only two articles, ultimately filtering down to the work by Valkov et al. By modifying the query to search for the same language name but across all metadata, it was identified 13 articles in the ACM Digital Library. Among these, only the work of Valkov et al. [5] is relevant. The query used it was the following:

```
"query": {
    Title:((Elixir OR Erlang) AND (Go OR Golang))
}
"filter": {
     E-Publication Date: (01/01/2013 TO 12/31/2024)
}
```

RQ1: How do the programming languages Elixir, Scala with Akka, and Go implement fault tolerance mechanisms in distributed systems, and what are the comparative strengths, weaknesses, and trade-offs of each approach?

Given the scope of the research question, developing a research strategy that yields precise and relevant results it challenging. Fault tolerance is a broad subject, spanning diverse areas from hardware to electronic devices until critical systems. Moreover, the programming languages under study are employed in varied contexts, such as Elixir's popularity in Internet of Things (IoT) and Go's extensive use in cloud-based applications and microservices, at the same time being general purpose languages used in diverse areas. This diversity introduces complexity when conducting research queries, resulting in an huge volume of information on a wide range of topics, or the lack of results in a more narrow query like the following one, that contains the important keywords for the work:

```
"query": {
    Abstract:(
        ("fault tolerance" OR "error handling" OR "resilience") AND
        ("distributed systems" OR "microservices" OR "software") AND
        ("Elixir" OR "Scala" OR "Akka" OR "Go" OR "Golang" OR "Erlang")
    )
}
```

To address this problem, the methodology employed covered using books, due to their mature and structure content, as also grey literature. Books provide comprehensive insights into foundational principles, and given that the evolution of programming languages tends to be gradual, they serve as reliable resources for understanding their core concepts and implementations, taking in consideration the choose of recently books. At the same time, white literature, including academic papers and recent articles, was included to capture the latest advancements, structures, and innovations within these languages.

Given the technical and practical focus of the study, a more ad-hoc research approach was adopted, like searching on the academic databases for more focused themes, not systematic queries or keywords. Official documentation for each language it was also consulted, as it provides up-to-date information directly from the creators. Furthermore, trusted blogs and community resources were consulted in order to utilize the collective knowledge and practical experiences of developers, which often provide valuable insight that may not be addressed elsewhere.

RQ2: What are the most effective benchmarking strategies for distributed environments focusing on fault tolerance aspects?

The methodology for addressing this research question is described below. It focuses on aspects crucial to this work, particularly fault tolerance and actor-based approaches due to the nature of the benchmarking process.

```
"query": {
    "Abstract": (
        ("fault tolerance" OR "reliability" OR "resilience") AND
        ("actor-based" OR "actor model" OR "actor programming") AND
        ("benchmark" OR "benchmarking") AND
        ("performance evaluation" OR "comparative") AND
        ("Erlang" OR "Scala" OR "Akka") AND
        ("concurrent" OR "distributed")
    ),
    "Title": (
        ("benchmark" OR "benchmarking") AND
        ("Actor" OR "Reliability")
    ),
    "Keywords": (
        ("benchmark" AND "fault tolerance")
    )
}
```

The query returned no results from IEEE Xplore but yielded 39 articles from the ACM Digital Library. However, some of these articles did not fully meet the accessibility criteria defined by the query keywords, leading to their exclusion. Additionally, certain papers primarily focused on product development rather than benchmarking techniques, which also contributed to the filtering process. To enhance the search, the snowballing technique was applied, allowing for the leveraging of references from relevant articles.

3.2 Elixir Programming Language Analysis

The following sections provide an overview of Elixir and its foundational principles within the Erlang ecosystem. This discussion will explore how the ecosystem relates to Elixir's modernization and how it enhances fault tolerance. Additionally, the fault tolerance strategies employed within this ecosystem will be examined, including their drawbacks and real-world applications, such as third-party libraries.

3.2.1 The Foundation of Erlang

Elixir is built on top of Erlang, making it essential to first understand Erlang's core principles and environment to move into Elixir's capabilities. Elixir leverages Erlang's foundation for constructing fault-tolerant and distributed systems, benefiting from its mature ecosystem and proven reliability [3, 27].

Erlang, developed in the mids of 1980s by Ericsson, was specifically designed to support systems that are highly reliable, responsive, scalable, and always available [3, 27]. Over the years, Erlang has evolved significantly, and Elixir represents a major milestone in this environment's evolution. Elixir enhances the ecosystem with modern features, such as a

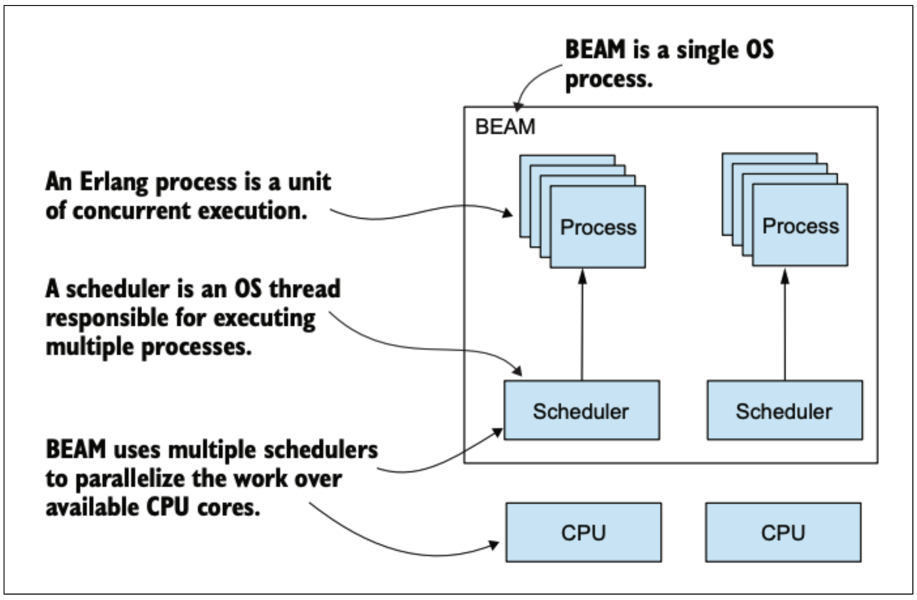

BEAM is a single OS process.
BEAM
An Erlang process is a unit of concurrent execution.
Process
Process
A scheduler is an OS thread responsible for executing multiple processes.
Scheduler
Scheduler
BEAM uses multiple schedulers to parallelize the work over available CPU cores.
CPU
CPU

Figure 3.1: Concurrency in the Erlang virtual machine [3].

more developer-friendly syntax, powerful metaprogramming capabilities with macros, and improved tooling, all while maintaining full compatibility with the Erlang runtime [3]. This success is closely tied to its coupling with Erlang's semantics, also the inclusion of the OTP, which provides robust libraries and tools. Additionally, Elixir inherits the power of BEAM, the Erlang VM, which could be considered as a state of art concurrent programming model [40].

Concurrency in BEAM

Concurrency is one of the most defining aspects of the Erlang environment, earning it the title of being a concurrency oriented language by many. At the heart of this model are processes, which adhere to the Actor Model [3, 5]. In this paradigm, each process acts as an independent actor, being lightweight and isolated, communicating with others through message-passing via mailboxes. These processes differ from heavyweight Operating System (OS) processes or threads, which rely heavily on the OS for management and lack the flexibility needed for optimizations. For instance, in the JVM, platform threads are a thin abstraction over OS threads, limiting control and optimization due to the fact of OS threads are heavy. However, virtual threads, introduced on Java 21, brings more capabilities to the JVM allowing a more fined scheduler like BEAM does, but in a less adoption for now [40].

In contrast, the BEAM virtual machine employs a concurrency-oriented programming model, where a single thread per CPU core manages numerous lightweight processes. This architecture enables BEAM to effectively handle parallelism by assigning one scheduler per CPU to oversee multiple lightweight processes. This approach is illustrated in Figure 3.1, which demonstrates how this architecture facilitates fault tolerance through process isolation. In the figure, the BEAM thread is shown alongside all associated processes, with each process linked to a scheduler, which in turn is connected to a CPU [3].

The BEAM scheduler is considered preemptive, meaning that assigns short execution time

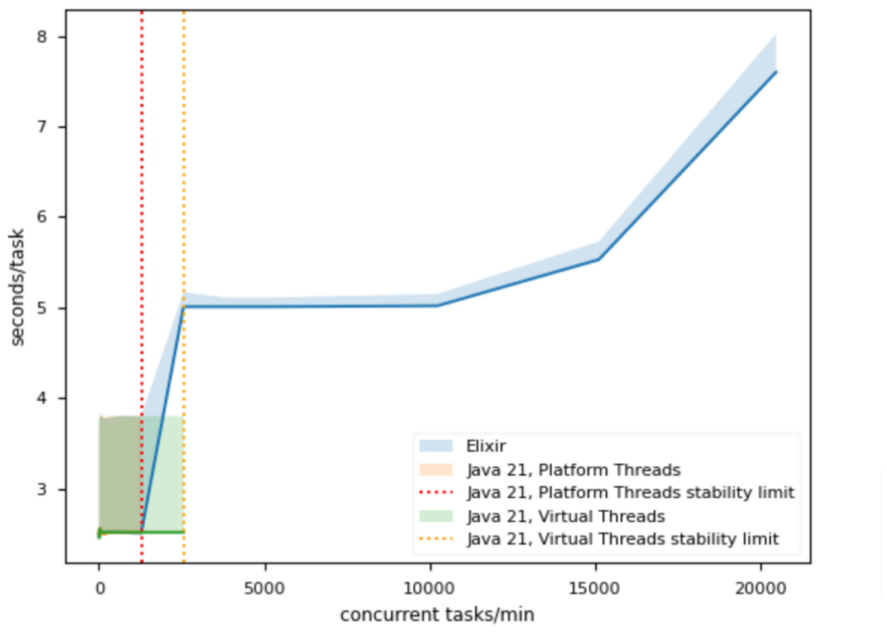


Figure 3.2: Elixir/BEAM processes vs JVM threads [40].

slices to each process. This ensures that long-running tasks do not monopolize system resources, promoting fairness and responsiveness [27]. Also, it promotes fault tolerance characteristic by stopping processes carried with permanent faults, where on a non-preemptive scheduler could harm the overall system. Processes that are blocked due to I/O operations or waiting for messages are efficiently managed by separate threads or a kernel polling service, preventing unnecessary CPU usage and ensuring that waiting processes do not stop the execution of others [3, 40].

In a direct comparison of Elixir's processes running on BEAM with the two threading techniques of the JVM [40], as illustrated in Figure 3.2, notable differences appears. Under low load conditions, all three strategies, Elixir's lightweight processes, the JVM's platform, and virtual threads, perform effectively. However, as the system approaches the stability limits of platform threads, approximately 2500 concurrent units, Elixir continues to handle additional processes, scaling up to approximately 200,000 concurrent processes. Although the per-task completion time increases slightly under such high loads, the system remains operational and stable. The opposite occurs with both JVM techniques, resulting in an overload that makes maintaining pace impossible. Furthermore, BEAM imposes a theoretical limit of roughly 134 millions processes, where this limit are lowered where the underlying implementation are a direct relationship with OS threads, like what happens in JVM [3]. The approximate minimum size of a process is more less than one kilobyte.

This scalability advantage can be attributed to the architecture of the underlying BEAM. Unlike the JVM, which relies on a shared heap and tightly integrates with OS threads. However, the JVM threading model is better suited for low-concurrency scenarios involving long-lived threads. In contrast, Elixir/BEAM excels in high-concurrency situations with short-lived processes [5, 40].

Garbage Collection and Immutability

Erlang and Elixir enforce immutability as a fundamental principle, ensuring that all data remains unchangeable. This eliminates many common concurrency issues in systems with shared memory, such as race conditions [5]. Instead of sharing memory, processes communicate by passing immutable data. When a message is sent, the receiving process creates a copy of the data in its stack, eliminating the need for semaphore controls or similar synchronization mechanisms [3, 40].

Because processes are completely isolated and do not share memory, BEAM can execute garbage collection at the process level. This per-process garbage collection allows the VM to reclaim memory for a single process without pausing the entire system, unlike the global garbage collection approach commonly used in the JVM, where all processes share a single heap. Additionally, BEAM optimizes garbage collection by focusing on individual schedulers enhancing its efficiency [3, 27].

The garbage collector can significantly impact the performance of both the BEAM and JVM. As illustrated in Figure 3.2, the load on BEAM outperforms that of the JVM. This difference may be attributed to the JVM's "stop-the-world" garbage collection, which can create performance bottlenecks. In contrast, BEAM utilizes a more targeted garbage collection approach, benefiting from process isolation, which can lead to enhanced performance [3, 5].

Hot-code swapping

Hot-code swapping is a beneficial feature for building fault-tolerant systems, allowing the modification of code that is actively running in real time. This mechanism enhances fault tolerance by enabling the replacement of fault code without requiring system downtime. The process is typically achieved by sending a message to the server, which then handles the exchange [27].

It is important to note that this capability is not implemented in the same way on the JVM. While the JVM supports class reloading, it is not comparable to hot-code swapping of BEAM and introduces significant complexities, such as managing already instantiated objects. In contrast, the hot-code swapping mechanism in systems that rely on BEAM allow targeted changes, focused on specific parts without disrupting the system [40]. Furthermore, in comparison with the Go language, which is a compiled language, does not permit hot code swapping in a production environment natively [4].

3.2.2 Fault Tolerance Mechanism and Strategies

Elixir's fault tolerance strategies and mechanisms are associated to the Erlang ecosystem, leveraging the features of the BEAM. A fundamental aspect of Elixir's fault tolerance is its adherence to the "let it crash" philosophy, which, combined with the Actor Model and extensive support from third-party tools, enhances its resilience. This is elaborated upon in the following sections.

Let It Crash Philosophy and Actor Model

Elixir inherits the “let it crash” philosophy from Erlang, which forms the foundation of its fault tolerance strategy. This philosophy is based on the principle that failures are unavoidable in distributed systems, and the optimal approach is not to prevent them entirely but to design systems that can recover autonomously and gracefully [1, 27]. Instead of defensive

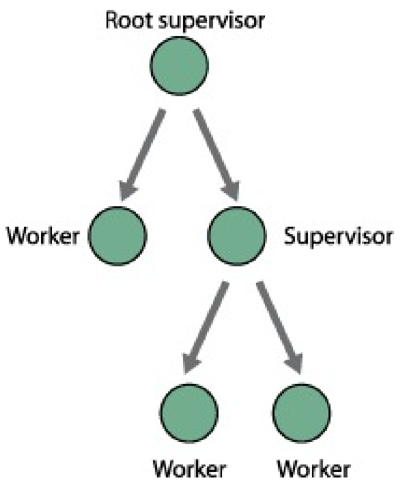


Figure 3.3: Supervisor tree pattern [3].

programming to anticipate every potential error, Elixir encourages developers to isolate processes so that faults can occur without compromise the stability of the entire system [3].

The Actor Model plays a central role in achieving this resilience. In Elixir, lightweight processes act as independent actors that do not share memory and communicate exclusively through message-passing. When a process encounters an unrecoverable error, it is allowed to fail and terminate. This termination is both deliberate and beneficial, as it enables easy fault detection and ensures that failures do not propagate, preserving the integrity of the overall system [3, 27]. This model naturally integrates with the supervisor pattern, which is one of Elixir's primary mechanisms for fault recovery.

Supervisor Pattern

The supervisor pattern is a practical implementation of the "let it crash" philosophy, built on the Actor Model. While the concept is not exclusive to Elixir, other frameworks like Akka also use it. Elixir leverage this pattern to build fault-tolerant systems [5]. In this approach, processes are classified into two types [3]:

- **Workers:** Processes that perform tasks or contain application logic but do not oversee other processes.
- **Supervisors:** Processes responsible for monitoring and managing other processes.

Supervisors are organized into a hierarchical supervision tree, as illustrated in Figure 3.3. This tree defines the relationships between supervisors and workers, with each supervisor manage a group of processes. This structure provides modularity and ensures that fault recovery is localized, reducing the impact of failures [27].

Supervisors in Elixir, as well as in the supervisor pattern used in other frameworks, employ restart strategies to manage failures effectively. The options provided by the OTP supervisors, which are among the most commonly used, include the following [3, 27, 41]:

- **One-for-One:** If a single worker process fails, the supervisor restarts only that process.
- **One-for-All:** If one process fails, the supervisor restarts all processes it manages.
- **Rest-for-One:** If a process fails, the supervisor restarts it and all other processes started after it in the hierarchy.

Each restart strategy addresses specific use cases. Additionally, supervisors can enforce restrictions on the restart process through a restart frequency configuration. This mechanism monitors the frequency of the worker process failures within a specified time frame. If a worker process fails repeatedly and exceeds the configured threshold, the supervisor itself terminates to avoid harming the system or entering an infinite restart loop [27].

The One-for-One strategy is best suited for independent processes [27]. For instance, in a web server handling multiple concurrent requests, this strategy could allow for the rapid recovery of a single failed process without affecting others. In contrast, the One-for-All strategy is ideal for tightly coupled processes [27]. When one process fails, all other processes under the same supervisor are restarted, this could be useful for processes that need synchronization among them. Finally, the Rest-for-One strategy could be used in workflows with sequential dependencies [41]. For example, in a data pipeline where each stage relies on the output of the previous stage, a failure in one process triggers the restart of the failed process along with any subsequent ones.

The supervision pattern is not uniquely associated with Elixir, it is also used in other languages that follow the Actor Model, as well as in various frameworks that implement this programming style. One of the most notable JVM frameworks is Akka [42]. Additionally, in other paradigms such as Go, there are libraries capable of unifying CSP with the Actor Model, such as the Proto-Actor[1] library [43]. Both are described at the flow of this document.

Tools and Support

Elixir's state of art in the fault tolerance area is related to the integration with the Erlang ecosystem, the BEAM, the Actor Model, and the "let it crash" philosophy. These elements are further enhanced by Elixir's compatibility with the OTP, which provides a suite of design principles and tools for building fault-tolerant and distributed systems. This integration allows Elixir inherit and extend the mechanisms that have been tested and proven in real case scenarios [3, 27].

The OTP framework enables Elixir to use the supervision tree pattern, an important element on fault tolerance like described early. By combining the supervision tree with tools like GenServer, Elixir simplifies the management of stateful processes, facilitates concurrent operations, and ensures the efficient handling of asynchronous message passing [41].

Additionally, OTP supports features like hot-code swapping, enabling systems to update running code in real-time without downtime. The inclusion of the Mnesia distributed database within OTP further strengthens Elixir's fault tolerance capabilities. Mnesia allows state storage across distributed nodes, ensuring data consistency and availability even in the presence of node failures [41, 44].

Beyond the core features of OTP, Elixir also includes Mix, a build tool that simplifies dependency management, testing, project configuration, and documentation generation. Mix integrates into the Elixir ecosystem, simplifying development workflows and contributing to the reliability of applications by ensuring consistent builds[41, 44].

In addition to the built-in capabilities of OTP, Elixir's ecosystem benefits from third-party projects that extend its fault-tolerant capabilities. For instance, Graft, developed by Le Brun

[1]Proto-Actor: `https://proto.actor/` (accessed 4 December 2024)

et al. [45] in 2019, and Ra[2], developed by the RabbitMq team, provide an implementation of the Raft consensus algorithm. Similarly, the Fuse[3] library, a widely-used implementation of the Circuit Breaker pattern, developed in Erlang, is also compatible with Elixir.

Another aspect of Elixir that is important to reference is its metaprogramming capabilities through macros, which allow developers to write code that generates code. This enables the Elixir codebase to be partially constructed using its own macros, extending the language's functionality and reducing boilerplate [3].

Lastly, it's important to mention the Elixir environment, which includes frameworks that enhance software development. Phoenix[4] is a popular framework for building scalable web applications. It inherits Elixir's fault tolerance, allowing applications to handle errors gracefully and maintain uptime. Phoenix also supports real-time features through channels for live updates [3]. Nerves[5] focuses on embedded systems, leveraging Elixir's fault tolerance to create resilient IoT devices. It simplifies firmware development and management, ensuring efficient hardware and system updates while addressing fault tolerance concerns.

These integrations, extensions demonstrate Elixir's ability to not only leverage the proven robustness of OTP but also adapt and grow through innovative tools and libraries, solidifying its position as a leading choice for building fault-tolerant, distributed applications.

3.2.3 Drawbacks and Real Applications

The benefits of Elixir are closely tied to the powerful features of the BEAM, as mentioned earlier. However, there are some drawbacks to consider. One major limitation is the lack of third-party libraries, despite OTP providing good support. Currently, it is challenging for Elixir to compete with more popular languages in this regard, like Java. Additionally, although the BEAM has a distributed nature, its single-threaded architecture with a garbage collector makes it less suitable for fault-tolerant applications in critical systems that require fault tolerance at low level [3]. Another potential drawback of Elixir is that it is a dynamically-typed programming language, which can result in errors from type mismatches or programming mistakes [46]. In contrast, languages such as Scala, Java, and Go offer advantages in this regard due to their static typing. However, there have been efforts to introduce a type system to Elixir without sacrificing the language's inherent dynamism, as demonstrated in the work of Cassola et al. [46]. Despite these efforts, the proposed type system has yet to gain widespread industry adoption.

Despite these limitations, Elixir has been successfully utilized in numerous prominent projects. Taking as reference the official website of Elixir, for example, Discord relies on Elixir as the backbone of its chat infrastructure, leveraging its ability to handle real-time communication effectively. PepsiCo also employs Elixir in a central role within its data pipeline, providing marketing and sales teams with tools to query, analyze, and integrate data from various search marketing partners. Other notable examples of Elixir's application include Heroku, SparkMeter, and several others.

[2]Ra: `https://github.com/rabbitmq/ra/` (accessed 4 December 2024)
[3]Fuse: `https://github.com/jlouis/fuse/` (accessed 4 December 2024)
[4]Phoenix: `https://phoenixframework.org/` (accessed 4 December 2024)
[5]Nerves: `https://nerves-project.org/` (accessed 4 December 2024)

3.3 Scala Programming Language with Akka Toolkit Analysis

The success of the Actor Model, particularly through its implementation in Erlang inspired other programming languages to replicate its concepts [3, 38]. Among these were languages running on the JVM, such as Scala and Java. However, significant differences emerged due to the JVM's inherent concurrency challenges. Unlike BEAM, which was built with lightweight process isolation and message passing at its core, the JVM relied on low-level thread management and shared memory, necessitating careful synchronization through locks and other mechanisms [5, 38]. Although this languages offered APIs for concurrency management, these general-purpose languages placed much of the responsibility on the developer [38, 42], going against the philosophy of Erlang/Elixir, that it was created for easy concurrency programming.

This gap on the JVM led to the creation of the Akka toolkit, designed to bring the actor programming model to its ecosystem. Inspired by Erlang, Akka offers a runtime and comprehensive tools to support actor-based programming, enabling developers to leverage the JVM while benefiting from a more structured approach to concurrency [42]. By abstracting thread management and offering a framework for distributed communication and fault tolerance, Akka provides a robust solution for building scalable and distributed systems. This capability is comparable to what Elixir offers [38], while also leveraging the constraints and advantages of the underlying JVM.

3.3.1 How Akka Handles the Actor Model

This section examines two key characteristics of the actor model and how the Akka toolkit addresses them. Specifically, it focuses on the distributed nature and communication aspects of the model, as well as the isolation that Akka provides. The Akka toolkit serves as an abstraction layer built on top of Scala and the JVM, facilitating the implementation of these principles in a robust and efficient manner.

Location Transparency and Communication

The Actor Model, like described before, defines a concurrent paradigm where actors operate independently, maintain their own mailboxes, and communicate exclusively via message passing. This communication should ideally respect the location transparent characteristic of distributed systems, allowing actors to interact seamlessly regardless of their physical location [27]. On the JVM, threading enables scaling within a single machine by utilizing additional CPUs and memory due to the shared heap memory and concurrency model [38]. However, native support for scaling across distributed systems is lacking, a contrast to BEAM's distributed aspect.

Elixir facilitates communication natively, with the Erlang distribution protocol [44]. Akka addresses this limitation through its latest remoting protocol, Artery, which builds upon the older remoting mechanisms making improvements [38, 42]. Artery employs either Transmission Control Protocol (TCP) or Aeron UDP for communication. While Aeron UDP delivers high throughput and low latency, it lacks encryption, making it suitable for specific trusted environments. TCP, on the other hand, offers encrypted communication with similarly high throughput, although with potentially higher latency under extreme load [42].

Akka facilitates scalability and communication with the discovery module. The discovery module serves a purpose similar to namespaces in Elixir, allowing actors to be registered with a specific name. In this context, actors can be registered using a designated key [38], and other actor can communicate with that name without knowing the specific address. Furthermore, communication with these actors occurs through their mailboxes, following a standard First-In-First-Out (FIFO) protocol, equals to the behavior in Elixir [3, 47].

Actors Isolation

Actor isolation is a foundational principle of the Actor Model, where actors are designed to operate independently and avoid shared state [27]. In Java, this independence can be implemented using low-level concurrency mechanisms, while in Scala it is often achieved through immutability. However, Akka significantly simplifies the process by providing an Actor API that inherently enforces isolation [38, 48]. Initially, Akka introduced the Classic Actors API, which supported untyped actor logic. In this model, messages were transmitted without type safety and processed using pattern matching, similar to the approach employed in Elixir [42]. In contrast, the modern standard is the Typed Actor API, which provides a more robust and type-safe solution. The Typed Actor API enforces type safety through, ensuring that only messages of the defined type can be sent to an actor [38, 42]. This differs from Elixir, where actor communication is dynamically typed and does not provide type guarantees.

Unlike Elixir, Scala permits mutable programming, which introduces the potential for shared mutable data to be passed between actors, a practice strongly discouraged in Akka's documentation. Despite these guidelines, the use of mutable data within Akka is technically possible. If misused, this could reintroduce well-known concurrency issues in the JVM, such as race conditions and thread interference [42]. This limitation stands in contrast to Elixir's approach, where the VM guarantees strict process isolation, effectively eliminating such risks [3, 5].

To achieve efficient performance, actors in Akka often share underlying threads, as individual threads are resource-intensive [47]. Akka actors mimic the lightweight processes of the BEAM by managing multiple actors within a single thread. This approach significantly reduces memory consumption compared to the heavyweight JVM threads. For instance, approximately 2.7 million Akka actors can fit within 1 GB of memory, a considerable contrast to the 4,096 threads that would occupy the same space [38]. That makes the minimum size of an acctor in Akka on an average 400 bytes.

Randtoul et al. [32] examined the effectiveness of actor isolation in Erlang and Scala with Akka. Their findings revealed that server throughput is affected by the termination of server actors. Specifically, they observed that the throughput for both Erlang and Scala with Akka decreases only in proportion to the percentage of processes that fail. This leads to the conclusion that both Erlang and Scala/Akka offer robust process isolation.

3.3.2 Fault Tolerance Mechanism and Strategies

Akka, built upon the Actor Model, inherits its fault tolerance philosophy from Erlang's design principles, like described on Elixir's section. This is the foundation of the fault tolerance aspects, at its core, the Actor Model facilitates fault tolerance through the supervisor pattern. By defining strategies for handling failures, such as restarting, stopping, or resuming child actors, supervisors ensure that errors are contained and localized, preventing system-wide failures [3, 38, 42], like it was detailed before.

In addition to its foundational fault tolerance mechanisms, Akka's modular environment offers some modules that extend the toolkit's capabilities, many of which directly contribute to improving fault tolerance.

Akka Clustering

Akka Cluster is a module that enables peer-to-peer communication among a group of nodes, allowing them to function as a unified distributed system [42, 49]. Designed to enhance fault tolerance, it implements replication and redundancy strategies while achieving location transparency through the Artery protocol, enabling nodes to communicate without knowing the physical locations of others [38].

The module provides an API for managing cluster operations, including managing nodes, designating a leader node, and obtaining cluster information. Although, Akka Cluster is autonomous, capable of redistributing workloads and managing actor states without manual intervention via API [38]. To ensure consistency and reliability, Akka Cluster employs a gossip protocol, a decentralized communication mechanism that facilitates the propagation of information across nodes [2, 42]. This protocol enables nodes to maintain a shared, consistent view of the cluster state [2].

For job processing, Akka Cluster employs a divide-and-conquer architecture through the use of master and worker actors. The master actor decomposes large tasks into smaller subtasks, distributing these among worker actors for parallel processing. Once processing is complete, the workers return their results to the master actor, which aggregates them to produce the final output. This design enhances performance and fault tolerance, as the master actor can reassign tasks from failed workers to other available nodes [42].

Leader election is a important aspect of a cluster management, and it is efficiently managed within Akka Cluster. In contrast to the Raft consensus, algorithm Akka Cluster adopts a simpler method by automatically assigning the leadership role to the node with the lowest address. This approach minimizes the overhead associated with the election process [25, 42].

Furthermore, Akka Cluster supports advanced features such as cluster sharding, which distributes actors across the cluster while preserving their identity and state. This facilitates load balancing and enhances performance by ensuring that workloads are distributed evenly. The module also includes monitoring and health management capabilities, enabling to verify the performance and status of nodes and actors [38].

Akka Circuit Breaker

Akka provides a dedicated module for implementing the Circuit Breaker pattern, a technique for improve system stability by isolating faults and preventing cascading failures. As presented earlier, the Circuit Breaker pattern temporarily suspends operations to a failing component, allowing it time to recover while safeguarding the overall system [23].

Akka's Circuit Breaker module offers a straightforward integration, making it possible to configure the way the circuit it will activate, where it can be defined failure thresholds, timeouts, and recovery intervals. This module is effective in monitoring interactions between actors and external services, ensuring that errors are contained and managed without disrupting the broader system [42].

Compared to Elixir's ecosystem, where circuit breaker functionality often relies on third-party libraries, Akka's Circuit Breaker benefits from being a included part of its toolkit.

Akka Persistence and Event Sourcing

Akka offers support for distributed persistence, similar to the functionality provided by Mnesia in Elixir and Erlang [41, 42]. It also features event sourcing, which contributes to fault tolerance through mechanisms such as message logging and check-pointing. Akka Persistence allows actors to recover their state after a failure. This is achieved by keeping an event log that records all changes to an actor's state in the order they occur. Upon restart, typically initiated by a supervisor, the actor replays the logged events to reconstruct its previous state, allowing it to resume operations from the point prior to the failure [42].

To optimize the recovery process, Akka Persistence also supports snapshots, supporting also the check-pointing strategy. Instead of replaying all events from the beginning of the event log, the actor can restore its state from the most recent snapshot and then replay only the events that occurred after that snapshot [38, 42].

3.3.3 Comparison with Elixir/BEAM and Real Applications

The Akka toolkit marks a significant step forward in extending the JVM to support modern concurrency models. As Valkov et al. [5] highlight, Akka improves Scala's performance by reducing communication latency. However, since Akka functions as an abstraction layer on top of the JVM, the same study by Valkov observed that Erlang exhibits lower communication latency compared to Scala with Akka. This difference is likely attributed to the BEAM, which is considered a state of the art concurrency model [40]. While the JVM was originally developed to meet general-purpose programming needs with an emphasis on efficiency, it does not natively prioritize the actor model or process-level isolation in the way the BEAM does.

However, Randtoul et al. [32] studied how Erlang and Scala with Akka manage server actors failures using a supervisor control pattern. They tested two supervisor-to-actor ratios (1:1 and 1:64) to see how throughput is impacted by different failure types. Their findings showed that both systems had similar throughput reductions during burst and random failures, especially with the 1:1 ratio. However, surprisingly, Akka outperformed Erlang in uniform failure scenarios with the 1:1 ratio. Despite of not being a directly Elixir comparison, the underline it is the same making it a valid comparison and showing the potential of Akka.

Garbage Collection. A notable difference between Akka and the BEAM lies in their approach to garbage collection. Akka relies on the JVM's garbage collection strategy, which can introduce latency during stop-the-world events [38, 42]. These pauses can negatively affect the performance of highly concurrent systems, especially under heavy load. However, advancements in garbage collection technology, such as the Z Garbage Collector, have shown possibilities in reducing pause times significantly. As noted by Chaudhary et al. [50], Z Garbage Collector represents a state of the art approach that is well-suited for applications requiring minimal pauses due to garbage collection. In contrast, the BEAM employs a per-process garbage collection mechanism, which localizes memory management to individual lightweight processes [3]. This architecture ensures that garbage collection in one process does not impact others, making the BEAM particularly effective in low-latency and high-reliability scenarios.

Scheduling Model. Scheduling represents a fundamental difference between Akka and the BEAM. The BEAM employs a preemptive scheduling model designed to efficiently handle numerous small, short-lived tasks, such as high-frequency message handling in highly concurrent systems. This approach ensures equitable CPU time distribution and mitigates process starvation [3, 40, 41]. However, frequent context switching can introduce overhead for long-running processes, potentially reducing efficiency in such scenarios. Akka, on the other hand, relies on the JVM's cooperative scheduling model and enhances it with its Message Dispatcher, which supports configurable thread pools like Fork-Join, that leverages a work-stealing algorithm, and Fixed Thread Pools, optimizing resource usage. This mechanism enables Akka to efficiently manage long-lived and computationally intensive tasks [38, 42].

Built-in Libraries and Support. Both the Akka and Elixir/BEAM ecosystems offer comprehensive libraries to address common design patterns for concurrency and fault tolerance. Elixir's OTP framework, like stated before, provides a robust suite of built-in patterns, such as supervisors, specifically tailored for managing concurrency and ensuring system reliability [40, 41]. Akka, on the other hand, adopts a modular architecture with an easily pluggable library of features, including implementations for replication and circuit breakers. In the Elixir ecosystem, equivalent functionality is often achieved through third-party libraries maintained by the community. While these community-driven libraries are highly effective and widely used, their reliance on external maintenance and updates can present a potential drawback compared to Akka's more integrated and officially supported approach.

Real-World Applications. Akka, just as Elixir, has demonstrated its capabilities in considerable large-scale applications, emphasizing its scalability, reliability, and performance. For instance, PayPal leverages Akka actors to manage over a billion financial transactions daily, ensuring high availability and robust fault tolerance [48]. Similarly, the Spark big data ecosystem depends on Akka for efficiently shuffling hundreds of terabytes of data across distributed nodes. Other prominent companies, including Twitter, LinkedIn, and Walmart use Akka to solve concurrency and distributed system challenges [42, 48].

3.4 Go Programming Language Analysis

The Go programming language was designed to facilitate rapid software development while ensuring high execution speed [51, 52]. It addresses the drawbacks of traditional low-level languages like C, which, while performant, can be complex for modern development. At the same time, Go offers a solution to the performance limitations of scripting languages such as Python, which prioritize ease of use but often fails in execution speed [51]. As a statically typed, compiled language, Go enforces type safety [4], setting it apart from dynamic and immutable languages like Elixir, as well as frameworks like Akka that also emphasize immutability with Scala.

While Go is not explicitly classified as a distributed or fault-tolerant programming language, it has gained considerable popularity in areas such as microservices, cloud applications, and high-concurrency systems. This popularity can be attributed to its simplicity, speed, and robust concurrency model [53, 54]. In contrast to Elixir, which is designed with immutability and a "let it crash" fault tolerance philosophy, Go does not inherently prioritize fault tolerance or the "let it crash" approach [52]. Nevertheless, Go can be a good candidate for integration into distributed architectures when paired with complementary technologies and libraries. By take advantage of Go's concurrency capabilities, it is possible to replicate the fault-tolerant strategies typical of Elixir-based systems, making it a valuable language to explore in this context.

3.4.1 Concurrency and Distribution in Go

Go adopts a concurrency model rooted in the CSP paradigm, distinguishing itself from other approaches such as the Actor Model. While both paradigms prioritize concurrent communication, they are slightly different in their focus [52]. In CSP, channels are treated as first-class entities, emphasizing the communication mechanism itself, whereas the Actor Model considers processes to be first-class, focusing on the entities performing the computation, as described earlier. Additionally, the Actor Model enforces strict isolation between processes, with no shared memory, while CSP organizes concurrent processes to interact explicitly through channels rather than directly access to a single shared memory object [52].

Go is one of the first programming languages to integrate CSP directly into its design, emphasizing data sharing through channels rather than passing references to shared memory among its lightweight threads, known as goroutines [52]. This design choice minimizes potential synchronization complexities and aligns with Go's guiding principle: "Do not communicate by sharing memory; instead, share memory by communicating" [4]. This philosophy contrasts with shared-memory concurrency models, where processes directly access and modify shared state. In Go, by design, only one goroutine can access a value at any given time, effectively eliminating the possibility of data races [4, 51]. Nevertheless, Go also provides manual synchronization mechanisms, such as explicit mutexes, for situations where they are necessary [52].

In contrast to the Actor Model, where actors encapsulate state and communicate through asynchronous messages, Go's CSP-based model prioritizes structured communication patterns via channels. This distinction encourages developers to design systems by focusing on the flow of data and the relationships between processes, rather than the individual behavior of the computational units [4].

Go's strategy utilizing CSP is centered around goroutines and channels, which are described below. Additionally, it will be briefly discuss how Go's garbage collector operates, as well as the limitations of channels and goroutines in supporting distributed communication.

Goroutines

Due to the overhead associated with OS threads, Go enhances efficiency by implementing a multiplexing logic that allows multiple processes to run on the same OS thread [52, 53], similar to the approaches used in Elixir and Akka. Go achieves this through goroutines, which are lightweight abstractions that enable concurrent and parallel execution. By utilizing goroutines, Go can efficiently manage numerous tasks without significant resource consumption [4].

The OS is responsible for scheduling threads to run on physical processors, whereas Go handles the scheduling of lightweight goroutines onto logical processors, which are subsequently bound to OS threads [51]. As shown in Figure 3.4, the example illustrate two OS threads (M2 and M3), with goroutines identified by the prefix *G*. The scheduling process involves allocating goroutines to logical processors via a local run queue. However, initially, goroutines are placed in the global scheduler run queue, and only afterward are they assigned

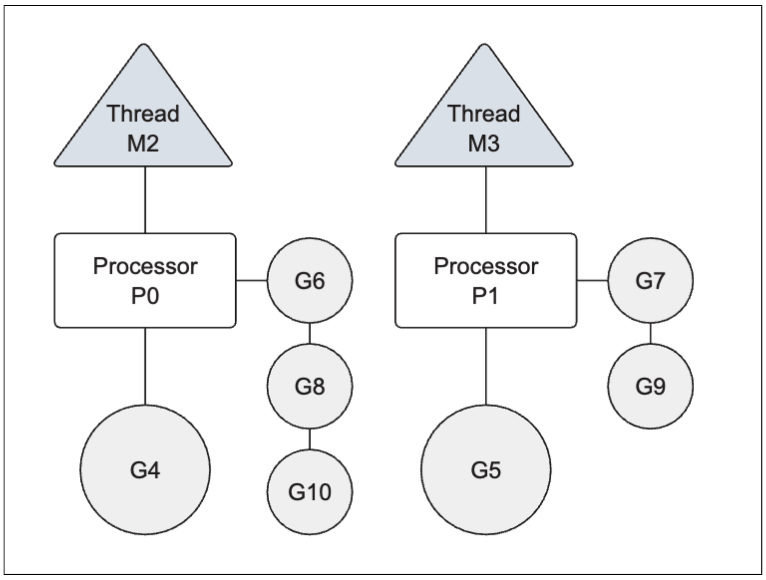


Figure 3.4: Go's scheduler logic of distributing goroutine by the logical processors. Adapted from [51].

to the local queues of logical processors [51, 52]. The same Figure 3.4, also illustrates how Go achieves parallelism. Goroutines can be distributed across multiple CPU cores if the hardware supports parallel execution [51].

Channels

Building on the CSP model, Go integrates channels as a core element of its concurrency paradigm. These data structures facilitate safe and efficient communication between goroutines, enhancing synchronization while addressing common challenges associated with shared memory access [51]. By following the principle that only one goroutine should modify a piece of data at any given time, channels help ensure data integrity, reducing the risk of concurrent modification and giving predictable behavior in concurrent programs [52].

However, channels in Go do not inherently enforce data access protection features such as immutability and isolation [51], which are fundamental to the concurrency models found in languages like Elixir and Scala. These languages are specifically designed to facilitate effective concurrency management, like analyzed earlier. Nevertheless, it is possible to adopt a strategy of using immutable data within Go channels, where all information is a copy of the original data [52]. This approach aligns more closely with the methodologies employed by Elixir and Scala, even though it is not the primary purpose of channels in Go.

Garbage Collector

In Go, the garbage collection strategy is based on a concurrent, non-generational mark-and-sweep algorithm [4], which operates globally on the heap [55]. This approach differs significantly from the BEAM's garbage collector, which performs garbage collection on a per-process basis. The BEAM's process-level garbage collection minimizes the impact on overall system performance by isolating garbage collection events to individual processes [3].

Go's garbage collector, while global in nature, supports partial heap collection but still experiences "stop-the-world" pauses. Despite these challenges, it is designed to maintain pause times between 10 ms and 100 ms under heavy load, making it comparable to the G1 garbage collector in the JVM, which, as studied by Zhang et al. [56], can experience pause times ranging from 0 to 300 ms. In contrast, ZGC achieves significantly lower pause times, typically between 0 and 0.1 ms, making it an attractive option for latency-sensitive applications, although it is not compatible with Go's environment [53].

Each garbage collection approach has its own advantages and trade-offs. Elixir's garbage collection, rooted in the BEAM runtime, is particularly well-suited for distributed programming due to its strong emphasis on process isolation [27]. In contrast, Go supports a partial heap-targeted garbage collector, which pairs effectively with its lightweight goroutines, enhancing memory management in concurrent applications.

Distributed Communication

A notable limitation of Go is its lack of native support for distributed communication [52, 57]. While the combination of channels and goroutines serves as an excellent tool for managing concurrency and parallelism, it does not inherently extend to communication across physical machines [51]. This limitation has prompted efforts to extend Go's concurrency model to support distributed systems. For instance, Whitney et al. [57] proposed a novel protocol called Gluster, which provides a library to abstract cluster logic and facilitate distributed communication. However, Gluster has seen limited industrial adoption and is restricted to Linux environments, limiting its general applicability.

Nevertheless, Go it offers seamless integration with mature and optimized networking libraries [51]. Packages such as TCP, HTTP, and gRPC provide efficient mechanisms for enabling communication between distributed components [4, 53]. These libraries significantly reduce the overhead associated with managing low-level networking concerns. Furthermore, channel-based networking libraries allow the management of distributed interactions effectively, leveraging goroutines and channels to handle the inherent asynchronous aspects of the network calls [53].

While Go's native concurrency primitives do not align with the Actor Model, there are projects of the Actor Model available for the Go ecosystem. One mature example is Proto-Actor [43, 57], a library that abstracts the complexities of distribution through its API. Built on top of gRPC, Proto-Actor provides a remote facilities and location transparency of the Actor Model within Go [43].

Go's rising popularity in the industry is closely tied to its adoption in microservices architectures and cloud-native applications [54, 55]. Microservices, by their nature, represent distributed systems and facilitate communication through both asynchronous and synchronous methods, often utilizing discovery services to map all nodes. Another approach involves the use of message queues, which can provide location transparency for processes [54]. However, these strategies may lead to over-engineering, in some cases, resulting in additional overhead compared to the native approach of Elixir.

3.4.2 Fault Tolerance Mechanism and Strategies

Go's approach to fault tolerance is not a central feature of the language, particularly in contrast to Elixir's "let it crash" philosophy. Instead, Go has a more explicit error-handling strategy that emphasizes direct management of errors. Also, to achieve fault tolerance capabilities similar to those of Elixir and Akka, it is often necessary to rely on specific patterns and libraries.

Error Philosophy

Until now, the "let it crash" philosophy has been described, a core principle applied in both Elixir and Akka due to the inherent design of the Actor Model. This approach is based on the inevitability appearance of errors, allowing them to occur and relying on mechanisms like the supervisor pattern to detect and recover from them [27]. However, the error-handling philosophy in Go is fundamentally different, representing almost the opposite paradigm. In Go, the strategy emphasizes handling every error explicitly [4, 51]. Errors are treated as first-class citizens, returned as values, and must be actively managed by the program. Unlike languages such as Scala and even Elixir [41], Go does not include mechanisms like try-catch for error handling. Instead, it enforces a more explicit style that requires developers to check for and respond to errors immediately after an operation [52].

This philosophy aligns with Go's overall design principles of simplicity, clarity, and explicitness. It is supposed that requiring developers to handle errors explicitly, Go minimizes the risk of overlooking potential issues. While this approach can result in more verbose code, it aims to reduce the likelihood of unhandled exceptions and promote a clearer method of error management [4, 51].

Another important consideration in Go's error handling philosophy is its impact on code readability and maintainability. The explicit nature of error handling in Go often leads to repetitive code blocks, resulting in more boilerplate compared to the code styles of Elixir and Akka [4, 51]. However, this explicitness can facilitate tracing how errors are propagated and resolved within a program. In distributed systems, this approach can complement techniques such as logging and monitoring.

Fault Tolerance Mechanisms and Strategies

While Go was not primarily designed with built-in fault tolerance mechanisms, as it emphasizes efficiency and simplicity, it has become a fundamental component in distributed systems, such as Kubernetes [51, 53]. When combined with appropriate patterns, architectural approaches, and libraries, Go enables the development of fault tolerance capabilities within these systems.

A more suitable approach in Go combines the heartbeat pattern with panic/recover mechanisms [52]. In this pattern, a goroutine functions as a supervisor, monitoring other goroutines through periodic status updates known as heartbeats or pulse. If a monitored goroutine fails to send a notification within the expected timeframe, the supervisor can initiate recovery procedures to restore the failed component's state [4, 52]. This ensures that failures are detected and addressed promptly. Furthermore, the supervision mechanism can be enhanced by Go's panic/recover pattern, which enables the system to capture and handle critical errors. This approach is similar to the supervision trees in Elixir and Akka, but it operates on a more specific and internal level, rather than addressing distributed aspects if needed [4].

A notable case study for this dissertation is Go's implementation of the Actor Model [57]. Roger Johansson, the creator of Akka.NET, with his team created an innovative approach to implementing the Actor Model in Go. This implementation demonstrates that the Actor Model can be effectively combined with CSP, as these paradigms could be complementary rather than mutually exclusive [43]. Proto-Actor, positioned as a next-generation Actor Model framework, introduces the "Actor Standard Protocol," which establishes a language-agnostic protocol for communication across different programming languages [43].

This implementation incorporates fault tolerance through the "let it crash" philosophy and location transparency within the Go programming language [43]. It employs gRPC and HTTP/2 [43], representing a more modern approach compared to Akka's Artery protocol and Erlang's distributed protocol. While this design leverages Go's efficiency, it is a library-based solution, similar to Scala with Akka, which introduces some overhead. However, there are two notable distinctions in execution: Akka runs on the JVM, while Go applications compile directly to machine code, potentially leading to different performance characteristics. Although direct comparisons with Scala with Akka and the JVM are not available, benchmarking tests for Proto-Actor show that it outperforms Akka.NET. Nevertheless, this performance advantage is unrelated to the JVM.

In the context of microservices architecture, Go provides robust support through the mature Go-kit[6] library [58]. This library facilitates the development of microservices and distributed systems by implementing essential patterns such as circuit breakers, rate limiters, and distributed tracing capabilities [54, 58]. Additionally, this framework can be enhanced with the failsafe-go[7] library, which introduces additional aspects of fault tolerance, such as retry policies. Furthermore, integrating HashiCorp's Raft[8] implementation can provide strong consistency and leader election capabilities.

Many of these solutions can be viewed as generic strategies that are more architectural than native, relying on third-party libraries. This is similar to the practices observed in Elixir, which frequently utilizes third-party solutions for replication, as well as in Akka for Scala. Nevertheless, the approaches outlined are effective and can capitalize on Go's popularity and efficiency.

3.4.3 Challenges Compared With Akka and Elixir and Real Applications

After examining the Go language, it is clear that it does not lend itself to fault tolerance mechanisms as naturally as Scala with Akka or Elixir. However, similar to how Akka enhances Scala, Proto-Actor leverages the combination of CSP with the Actor Model in Go [43]. Just as Elixir relies on third-party libraries to implement Raft consensus, Go also requires external libraries to achieve fault tolerance capabilities. Nevertheless, it easy to observe that Elixir's environment is robust and natively implements these features, providing a distinct advantage and a more convenient approach.

According to the Proto-Actor benchmarking performance results [43], one test involved an initial actor spawning 10 new actors, each of which spawned another 10, continuing until a total of one million actors were created. Each actor returned its ordinal number, which was summed at the preceding level and sent back upstream to the root actor, resulting in a final sum in the range of 11 digits. In this test, Erlang outperformed the Actor Model implemented in Go, likely due to its optimized handling of short-lived processes.

In a different test, on the same source of Proto-Actor benchmarking performance results [43], two actors, one on each of two nodes, were used to exchange one million messages

[6]Go-kit: `https://gokit.io/` (accessed 4 December 2024)
[7]failsafe-go: `https://failsafe-go.dev/` (accessed 4 December 2024)
[8]HashiCorp's Raft in Go: `https://github.com/hashicorp/raft/` (accessed 4 December 2024)

back and forth. In this scenario, Go surpassed Erlang in throughput, a result attributed to Go's use of message references, which likely reduced overhead.

A study conducted by Marchuk et al. [59] revealed that Elixir outperformed Go in both requests per second and messages per second during a load test simulating a backend scenario. This test underscored Elixir's superior performance and efficiency.

In a similar vein, Valkov et al. [5] found that Go, likely due to its use of typed channels and the absence of a need for pattern matching, achieved higher throughput compared to Scala with Akka and Erlang. Notably, both Go and Erlang demonstrated the lowest message latency among the platforms evaluated. Furthermore, Go and Erlang exhibited more predictable scaling, with consistent increases in spawn time. In contrast, Scala with Akka experienced higher spawn times and less predictable scaling, particularly showing a significant performance spike when scaling from 10,000 to 20,000 processes.

Maintainability and Readability of the Code. One of the primary challenges when using Go for fault tolerance is the maintainability and readability of the code. While Go emphasizes simplicity and clarity, the absence of built-in fault tolerance mechanisms can lead to more complex code structures when implementing custom solutions [4]. In contrast, Elixir and Scala with Akka provide clear abstractions for fault tolerance, such as supervision trees and actor models, which inherently promote maintainability.

Built-in Libraries and Support. While Go has a growing ecosystem of libraries that facilitate fault tolerance, such as Go-kit and Proto-Actor, it lacks the extensive built-in support that Elixir and Akka offer. Elixir's OTP provides a rich set of libraries and tools specifically designed for building fault-tolerant systems, while Akka's actor model is deeply integrated into the framework.

Real-World Applications. Go is extensively used in cloud applications and high-performance systems [55], particularly within Google, where it was originally developed. It plays a vital role in platforms such as Docker and Kubernetes, and companies like Dropbox have successfully transitioned from Python to Go to enhance efficiency [4]. Another notable example is Cockroach Labs, which has praised Go's garbage collection and performance as well-suited to their requirements [4]. However, challenges do exist. For instance, Discord initially implemented Go but later migrated to Rust due to issues with Go's garbage collection, which resulted in significant latency spikes. This led to the conclusion that the garbage collector was a contributing factor to performance degradation [60].

3.5 Benchmarking Analyses

According to Almeida et al. [61], the primary objectives of benchmarking are to *"provide a practical way to characterize and compare systems or components according to specific characteristics (e.g., performance, dependability)"*. Benchmarking delivers insights within a specific domain by quantifying key metrics, enabling practical comparisons. For results to be valid and meaningful, it is critical to conduct repeatable experiments. Benchmarking serves as an experimental approach that derives value from measurable outcomes, yielding consistent results under identical conditions, or statistically analyzed [61, 62]. Deterministic benchmarks are especially valuable as they ensure reproducibility when the same assumptions are applied.

Non-deterministic approaches are typically associated with chaos engineering, which focuses on testing system resilience by intentionally introducing random faults [32]. However, the insights gained from benchmarking in this context are relative and applicable only to the specific conditions under which the tests were conducted, due to the inherent randomness of the process. Consequently, these tests often lack reproducibility and may not offer comprehensive coverage across all system sizes [61].

On a overview, benchmarking is generally divided into two categories [5, 61–63]:

- **Macro Benchmarking:** This approach assesses the overall performance of an application or system. It evaluates the application as a whole, which can make it difficult to isolate and analyze specific components. Macro benchmarking is particularly useful when it is important to observe the interactions among components in their entirety.
- **Micro Benchmarking:** This method focuses on individual components, functions, or metrics, allowing for detailed and targeted analyses. Micro benchmarking is especially beneficial for studies that require an in-depth examination of specific aspects of the application, enabling developers to identify performance bottlenecks and optimize accordingly.

The well-known Computer Language Benchmarks Game[9] supports various algorithmic benchmarking tests, evaluating runtime, memory usage, and related performance metrics [62]. However, this benchmarkings does not support newer languages like Elixir, Scala, and Scala with Akka, nor does it address resilience-focused benchmarks. This limitation highlights the need for dedicated tools and frameworks tailored to resilience benchmarking in modern programming environments.

3.5.1 Fault-Tolerant and Distributed Benchmarking

Benchmarking fault tolerance in distributed systems presents specific challenges, requiring evaluative strategies that extend beyond conventional performance testing. Key considerations include not only throughput and latency under nominal conditions, or how the computational occurs under an algorithm execution, but also the ability to keep executing, to detect and recover from faults, including node failures, communication interruptions, and state inconsistencies [32, 61, 62].

The objective is to develop benchmarks that accurately simulate real-world errors. These benchmarks should closely reflect actual software behavior, taking into account factors such as overload conditions and software faults. In fault tolerance benchmarks, it is crucial to incorporate components that introduce faults in order to achieve dynamic accuracy [61]. This can be accomplished through random fault injection methods, such as Chaos Monkey, developed by Netflix, which randomly destabilizes the system, or through deterministic fault injections, where specific errors are deliberately introduced into the system and it is known what happened in order to corelate the metrics with the faults [32].

Distributed benchmarks present some challenges, particularly in maintaining effective communication between components. These challenges can lead to increased latency and may involve physical limitations, such as managing multiple machines, dealing with diverse hardware configurations, and relying on network connectivity.

[9]Computer Language Benchmarks Game: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/` (accessed 4 December 2024)

Strategies

The Computer Language Benchmarks Game, as previously introduced, provides a foundational collection of micro benchmarks designed to compare the performance of programming languages. Although it does not include distributed systems and lacks support for some newer languages, it has a set of algorithms that have served as the basis for other benchmarks, such as the work by Cardoso et al. [64]. This particular benchmark focuses on the agent and actor model, leveraging three algorithms from the collection: Fibonacci numbers, token-ring[10], and chameneos-redux[11]. These algorithms, each tailored to distinct computational tasks, offer a valuable resource for designing performance tests in a variety of contexts [32, 64]. They can be used in conjunction with resilience benchmarks to simulate processing, for example.

For instance, Savina, developed by Imam and Sarkar [63], has emerged as the de facto benchmark for evaluating actor model performance. It utilizes micro benchmarks alongside concurrency and parallelism techniques to assess the behavior of actor-oriented programs in compute-intensive applications. While Savina provides valuable performance insights, its scope is restricted to JVM-based languages, such as Akka with Java, thereby excluding languages like Elixir, Go, and Scala with Akka. Additionally, its focus is limited to single-node environments, offering no support for evaluating distributed systems.

Savina employs 28 benchmarking strategies, categorized into three groups: 7 for micro-benchmarking, 8 for concurrency, and 13 for parallelism. These benchmarks evaluate critical aspects such as communication efficiency, node creation and termination, mailbox contention, IPC resource allocation, among others [62, 63]. However, Savina's focus is confined to actor-based systems and localized performance, excluding non-actor-based and distributed communication scenarios.

Blessing et al. [62] highlighted significant limitations in the micro benchmarking Savina, particularly their narrow focus on isolated performance metrics. To address these issues, they proposed an application-oriented benchmarking approach that simulates real-world scenarios, such as a chat application like Facebook or WhatsApp. This approach shifts away from isolated micro benchmarks, such as evaluating latency in message-passing programs, to consider broader application-level metrics.

Their proposal centers on creating a comprehensive application that encompasses multiple scenarios. This approach aims to make benchmarks more relatable to real-world applications, encouraging developers to optimize the all system rather than focusing on discrete, independent cases [62]. To enhance flexibility, they propose the addition of tunable options, such as varying the number of load balancers, if applicable, to simulate bottlenecks or adjusting message sizes to test system behavior under different conditions, for example.

Despite its innovative approach, the implementation by Blessing et al. [62] had some shortcomings. It did not support stateful clients, distributed environments spanning virtual machines, or resilience testing. These limitations highlight opportunities to design advanced application-based benchmarks that incorporate fault tolerance techniques and cater to broader, more complex distributed systems.

[10]The token-ring algorithm simulates a network of nodes passing tokens in a circular manner, which is useful for evaluating message-passing and synchronization in concurrent systems [64].

[11]Chameneos-redux is a concurrency benchmark that models a group of agents (chameneos) interacting with each other based on color, showcasing the complexities of state management and communication in concurrent programming [64].

Randtoul and Trinder [32] present a study focused on evaluating the resilience of actor-based systems under fault scenarios through fault injection techniques. The authors did not used chaos engineering due to its inherent unpredictability, opting instead for a deterministic approach. This choice ensures reproducibility and controlled experimentation. Being deterministic, also facilitates a clear relation between fault inputs and the resulting metrics.

The benchmark developed in the study operates within a single physical machine, deploying multiple nodes without relying on separate physical machines or cloud-based resources [32]. It employs the supervisor pattern, a mechanism designed to recover from errors affecting supervised actors. The system allows for customization of the supervisor-to-supervisee ratio, with configurations ranging from 1:1 to 1:128. Fault injection tests are designed to simulate errors that closely mirror real-world scenarios, using the following fault patterns [32]:

- **Uniform Failure Pattern:** Distributes failures evenly, by a spontaneous injection, simulating periodic failures typical of web servers handling distributed requests.
- **Burst Failure Pattern:** Simulates sequential or simultaneous actor failures, mimicking hardware or network errors that propagate, potentially causing widespread system disruptions.
- **Random Failure Pattern:** Combines characteristics of burst and isolated failures, representing a realistic mix of spontaneous single failures and clusters of failing processes.
- **Progressive Permanent Failures:** Gradually terminates actors or processes at fixed intervals, such as every five seconds, for example, to simulate irrecoverable scenarios. This contrasts with the, before detailed, recovery patterns by emphasizing the inability to restore normal function.

The benchmark was implemented using Erlang and Scala with Akka [32]. The authors noted a limitation of the benchmark, that is the inability to utilize stateful actors, which presents certain challenges. Consequently, all tests were conducted using stateless actors.

After reviewing the most recent and relevant benchmark studies suitable, for this case, it is concluded that there are gaps in the current benchmarks available. However, these gaps present an opportunity to propose a benchmarking approach. This approach could employ a hybrid methodology, integrating various aspects discussed in the reviewed studies. For example, it could incorporate the fault injection strategies detailed in the work of [32], while adopting the generic application framework proposed by [62], which involves implementing a chat application. To address distributed communication, all nodes could be deployed on a single physical machine across different VMs, as similar of the work described on [32]. This setup minimizes the complexities associated with infrastructure management and network latency, as the nodes are distributed across virtual machines on the same host.

Metrics

Metrics are a fundamental aspect of the benchmarking process, providing a quantitative basis for evaluating system characteristics. They represent measurable outcomes of system execution and are essential for drawing meaningful conclusions [61]. The choice of metrics is critical, as it defines the evaluation scope and ensures the relevance of the results [1, 61].

In the observed context, metrics are generally categorized into two main types: performance-focused metrics and fault tolerance metrics [5, 32, 61]. Performance metrics assess system efficiency and responsiveness, either independently or in conjunction with fault tolerance, to evaluate how the system performs under normal and faulty conditions [5]. In contrast, fault tolerance metrics specifically examine a system's ability to manage and recover from failures, including metrics such as recovery time and fault handling capacity. Combining these perspectives provides a comprehensive evaluation, offering deeper insights into system performance and reliability [32].

For this study, metrics such as those employed by Valkov et al. [5], process creation time and latency among concurrent, are useful to assess server performance. Additionally, performance metrics from the Savina benchmarking suite [63], which focuses on message passing and actor-based performance aspects, are considered a valuable. Fault tolerance metrics add further depth by measuring specific attributes related to failure management [32], where it has information about how system can recover.

Code metrics, despite not being used on the studies, are a valuable tool that provides insights into the nature of the code. Considering the variety of programming languages and paradigms available, it is useful to make comparisons. For instance, one might examine whether the Lines of code (LOC) significantly differs between languages to gain insights into code complexity [2].

3.6 Conclusions

After the research about the topics of study, it is now possibility to answear to the research questions and also state what are the goals to the upcomming work of this dissertation with the future work.

3.6.1 Research Questions Answers

RQ1: How do the programming languages Elixir, Scala with Akka, and Go implement fault tolerance mechanisms in distributed systems, and what are the comparative strengths, weaknesses, and trade-offs of each approach?

Elixir, built on the foundation of Erlang and its BEAM VM, stands as a paradigm of fault tolerance in distributed systems. The BEAM's concurrency model, combined with immutable data structures and garbage collection, provides a robust environment for concurrent operations [3]. More importantly, Elixir inherits Erlang's "let it crash" philosophy, which embraces failure as a natural part of system operation [27]. This principle is reinforced by the supervisor pattern, where supervisors manage hierarchies of processes, ensuring that when failures occur, they are isolated and handled by restarting failed processes in a controlled manner [3]. This native implementation makes fault tolerance intrinsic to Elixir's design, requiring minimal overhead from developers and allowing systems to recover gracefully.

In contrast, Scala with the Akka toolkit extends the JVM's capabilities to include actor-based concurrency and fault tolerance. While Scala does not natively provide fault tolerance at the language level as Elixir does, the Akka it offers a powerful toolkit for building resilient systems [38, 42]. Features such as Akka Persistence, Akka Clustering, and Akka Circuit Breaker significantly enhance its capabilities, enabling systems to support state recovery, fault isolation, and resilience in distributed environments. However, Akka's reliance on the JVM introduces certain complexities and potential performance overhead when compared to BEAM's purpose-built design. Additionally, the possibility of mutable programming in Scala can pose risks [42].

Go, with its different approach to fault tolerance, offers a different paradigm. The language emphasizes error handling explicitly, being different from the "let it crash" philosophy found in Elixir or Akka [4]. Instead, Go encourages developers to take a proactive role in managing errors and system behavior. Concurrency in Go is achieved through goroutines and channels, inspired by CSP [43]. While Go lacks a native actor model, libraries such as Proto-Actor merge CSP concepts with actor patterns, offering a mature option. Additionally, Go's efficient network communication libraries and tools enable developers to build distributed systems that can integrate with service discovery and orchestration tools, to achieve fault tolerance [54].

Elixir's BEAM VM, with its preemptive scheduler, handles lightweight, short-lived processes efficiently, ensuring fair CPU distribution even under high concurrency [41]. Go's cooperative scheduler is simpler and resource-efficient but can suffer from CPU monopolization if goroutines fail to yield [4]. Akka, using the JVM's dispatcher with work-stealing algorithms, balances threads effectively for longer-running tasks but incurs higher overhead compared to BEAM's lightweight processes [42]. Additionally, Elixir's hot-code swapping supports live updates without downtime, a capability not supported on the same level in Akka in Go [3].

Fault tolerance is deeply ingrained in Elixir's OTP, providing native tools like supervision trees, process isolation, and fault recovery mechanisms designed specifically for distributed systems [41]. Akka, while not native to Scala, offers a powerful suite of tools like Clustering, Persistence, and Circuit Breakers but requires some configuration and runs atop the JVM, introducing additional complexity [42]. Go's explicit error handling prioritizes simplicity but shifts the burden of fault recovery onto developers [4]. While Go supports distributed systems via libraries like Proto-Actor and efficient network communication tools, it lacks the seamless fault tolerance integration of OTP or Akka's abstractions. Elixir's immutable state and BEAM's process isolation enhance safety, whereas Akka and Go, with their reliance on mutable constructs, require extra vigilance to avoid shared-state corruption [4, 42]. Finally, in ecosystem maturity, Elixir benefits from OTP's decades of optimization, while Akka and Go rely on frameworks layered atop general-purpose runtime environments, offering flexibility but at the cost of tight native integration.

RQ2: What are the most effective benchmarking strategies for distributed environments focusing on fault tolerance aspects?

Benchmarking fault tolerance, a key aspect of system resilience, in distributed environments, presents a significant challenge due to the diverse factors involved. Although some benchmarking techniques, particularly those designed for actor-based systems, are available, there is currently no formal standard. Each technique introduces its own methodologies and metrics, complicating direct comparisons of results across different systems. Nevertheless, a common theme among these benchmarks is the simultaneous evaluation of fault tolerance and performance metrics.

Error injection, a core element of these strategies, is typically implemented using two main approaches: chaos engineering or deterministic error injection. Chaos engineering emphasizes introducing random failures into a system to evaluate its resilience under unpredictable conditions, mimicking real-world faults [61]. On the other hand, deterministic error injection follows a structured approach, simulating specific failure scenarios to predictably test system responses [32]. Deterministic error injection offers greater reproducibility and precision, making it a more reliable option for controlled benchmarking efforts.

There are strategies focused on micro benchmarking, which involve targeted tests, as well as broader benchmarking approaches that encompass various tests within a more wide system. Given the diversity of available strategies, a hybrid approach that combines generic application simulations with deterministic error injection can offer a balanced and comprehensive perspective [32]. This strategy should also integrate performance monitoring, resilience assessment, and static code analysis to address fault tolerance in a holistic manner [5, 32, 63, 64].

In terms of metrics, effective benchmarking should take into account both performance and resilience indicators. Common performance metrics include latency, throughput, response time, and resource utilization [5, 63]. Resilience-specific metrics may include time to recovery, error rate, and time to detect errors, among others, due to the fact that it is dependable of the benchmarking design [32, 62]. Additionally, static code analysis can provide valuable insights into the development effort associated with each programming language, like the LOC metric.

3.6.2 Future Work

For future work, the goal is to build a chat application based on the conclusions drawn, creating use cases to explore and compare how the selected languages and the fault tolerance patterns and designs handle fault tolerance and resilience in distributed environments.

This chat application will be implemented in all three languages. For Go, two separate implementations will be developed: one using the Proto-Actor library to provide a direct comparison of actor models across Elixir, Akka, and Proto-Actor, and another leveraging Go's native features combined with libraries that mimic distributed communication and fault tolerance.

The primary objective of this application is not to evaluate scalability, such as handling a high number of requests by scaling up to additional VMs. Instead, the setup will use a fixed number of three VMs to simulate distributed communication, focusing on fault tolerance and performance. Scalability-related considerations will not be employed. Furthermore, as mentioned, the distribution simulation it will be implemented on a single physical machine though inter-VM communication. This approach assumes that the primary difference between communication across different physical machine and one machine it is the latency.

Design and Architecture

The application will be designed taking inspiration from the work by Randtoul and Trinder [32] with increments and adjustments. The application will have five main responsibilities:

- **Clients:** Represent the users, acting as the client side. Each user will be represented by a client, which will have the capability to send and receive messages. It will manage connections to directories and chats. Also, it will the responsible to simulate failures from the client side.
- **Discovery:** Acts as the registry and discovery server to store and provide client addresses. This serves as the source of truth for client locations and will be used by the external injector for action orchestration.
- **Chats:** Represents group conversations and supports multiple clients. Handles operations related to group chats, such as join, leave, and message forwarding.

- **External Injector:** Injects actions into the system, including fault injection and operational commands. This orchestrates application behavior, dictating client activities and fault simulations such as node crashes. Maintains constant communication with the discovery server to stay updated on client addresses. However, this part it is not related to the system itself, so it do not have to be protected against fault.
- **External Logger:** Responsible for tracking the traceability of actions and generating logs for metrics and analysis. Similar to the external injector, this component do not have to be fault-tolerant, if a error happens all the benchmark need to abort and be restarted.

There exists an "inner world" and an "external world" within the system architecture. The inner world comprises the core components of the real system, such as the client, discovery, and chat functionalities. In contrast, the external world consists of supporting components that facilitate benchmarking by issuing commands and collecting metrics. It is important to note that the external world is not safeguarded by fault-tolerant mechanisms, and because of that, in the event of a failure, it must be aborted to prevent interference with the test.

This architecture is intentionally designed to be generic, avoiding assumptions about the Actor Model or microservices paradigms, thereby allowing for customization based on specific system properties.

Implementation and Functioning

The primary goal of this architecture is to facilitate general-purpose fault tolerance testing, which encompasses the evaluation of performance and system design. For instance, this architecture can be implemented using the Actor Model across all three programming languages, incorporating the supervisor pattern. This approach enables a comparison of recovery mechanisms and the effectiveness of the "let it crash" philosophy.

Furthermore, the design allows for the evaluation of various fault tolerance mechanisms, such as the Akka Circuit Breaker in comparison to Elixir's supervision strategy and Go's explicit error handling. By measuring downtime and system recovery time, we can assess the suitability of these different approaches.

The implementation will leverage the Actor Model for both Elixir and Scala using Akka, utilizing their native supervisor patterns. For Go, two distinct implementations will be developed: one will employ the Proto-Actor library to mimic the Actor Model approach used in Elixir and Akka, while the other will be a "vanilla" implementation utilizing goroutines and channels. This second implementation will simulate a lightweight microservices architecture, with communication between components facilitated via gRPC and inter-node communication managed through goroutines and channels.

Configuration

In accordance with the principles outlined by Randtoul and Trinder [32], the benchmark should incorporate various configuration aspects, referred to as tuning knobs:

- **Message Characteristics:** This allows for the configuration of message size, enabling representation of both small texts and large objects. Additionally, the type of message can be specified, whether it involves CPU-intensive or memory-intensive operations, with adjustable algorithms. The intention is for the message to execute the algorithm either upon receipt or prior to being sent.

- **Failure Scenarios:** This feature enables the simulation of various failure bursts, such as the simultaneous crashing of all clients in a chat, targeted failures (e.g., a single client or chat crashing), or cascading failures that propagate through the system.
- **Client Behavior:** Allows for customization of client activity levels, such as the frequency of sending messages, joining or leaving chats, and interacting with other clients. Parameters can control the ratio of active to idle clients, simulating various system loads.
- **Discovery Server Load:** Enables adjustment of the number of clients handled per discovery server, testing the impact of load balancing and potential bottlenecks. This can be used to evaluate the performance of the discovery mechanism under high demand.
- **Chat Size and Connectivity:** Configures the number of clients per chat and the probability of clients being invited to new chats. This can simulate low-connectivity systems versus highly interconnected networks.
- **Supervision Strategies:** For Actor Model implementations, allows selection of different supervision strategies (e.g., one-for-one, one-for-all) and configuration of supervisor tree depth and breadth. This helps analyze recovery performance and bottlenecks in supervision hierarchies.

Test Scenarios and Metrics

The benchmark will evaluate fault tolerance through the following test scenarios:

- **Client Crash Recovery:**
  Description: Simulate a client crashing and being recovered. The crash must be fatal and non-fatal. For example, one scenario could simulate a complete outage for some seconds, and another could simulate an internal error, such as an exception. This allows comparison of the "let it crash" philosophy with the explicit error handling in Go, specifically in scenarios where exceptions can be controlled.
  Metrics: Time taken to recover, consistency of the system state post-recovery, and LOC of different strategies or the supervisor implementation in the different languages.
- **Chat Crash Recovery:**
  Description: Simulate the chat system crashing and recovering. Upon recovery, all clients must also reconnect successfully. The test registers the time taken for the system to stabilize, including reconnecting all clients. This scenario also tests replication, where each language will implement replication using the Raft algorithm, having a replicated chat for fault tolerance.
  Metrics: Time taken to recover, time to leader election after a crash, time to recover all state, and consistency of the system state post-recovery.
- **Message Delivery Durability:**
  Description: The system must continue to deliver messages even during failures and during scenarios where processes run indefinitely. This test simulates a client sending a consistent stream of messages while inducing infinite message loops to evaluate how the different schedulers operate across languages.
  Metrics: Number of undelivered messages, time to recover message flow after failure, and consistency of delivered message order.

- **Message Throughput Durability:**
  Description: Evaluate the system's ability to maintain message throughput during fault injection. This test simulates various failures, such as client crashes, server crashes, or high resource utilization, while measuring the message throughput across the system. The objective is to assess how well the system sustains performance under stress and whether it degrades gracefully during failures.
  Metrics: Average and peak message throughput during faults, percentage drop in throughput compared to normal operation, recovery time to regain baseline throughput, and the impact of different fault tolerance mechanisms (e.g., "let it crash" vs. explicit error handling).

- **Network Test:**
  Description: Assess the system's resilience to network-related issues. This scenario compares the effectiveness of fault tolerance mechanisms like supervisor hierarchies in Elixir and Akka versus circuit breakers in Akka and Go. The test involves simulating network partitions and latency spikes to measure the system's ability to recover and maintain consistent communication.
  Metrics: Time to detect and recover from network partitions, latency impact on message delivery and system state synchronization, time to stabilize after network recovery, and a comparative analysis of recovery performance between supervisor-based and circuit breaker-based mechanisms.

Data Analysis

To obtain the results, we will utilize a logger that consolidates the final information. The logs are received by the logger in a disordered manner, necessitating organization. Each event will have a unique identifier, allowing for effective tracking and tracing.

Chapter 4

Project Planning

4.1 Project Charter

The project charter provides an overview of the stakeholders, benefits, and assumptions of the project to state the beginning of the project. This document serves as an formal source that captures the project's initial vision.

Stakeholder

Identification	**Power**	**Interest**
Students and developers in the areas of distributed systems and fault tolerance	Low	Medium
Administration of the master's program, responsible for dissertation evaluation	Medium	Medium
Advisor	High	High

The administration of the master's degree programme has a moderate level of influence and interest in the project, as they must ensure compliance with institutional requirements and rules. Additionally, they have interest in the project's outcome, given its association with the institution. In contrast, the advisor has a high level of influence and interest, as their guidance and approval are crucial to the project's success.

On the other hand, students and developers in the field of software engineering have an interest in the project's outcome, as it will provide guidance on selecting languages and strategies related to fault tolerance. However, they have limited power to influence the project's direction.

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 to already existing ones.
- **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.

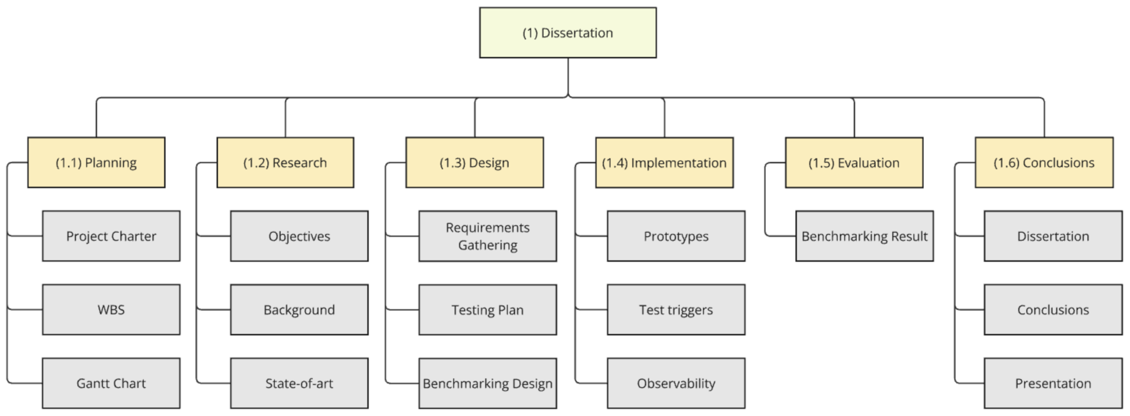


Figure 4.1: The WBS of the project.

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 advisor 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.

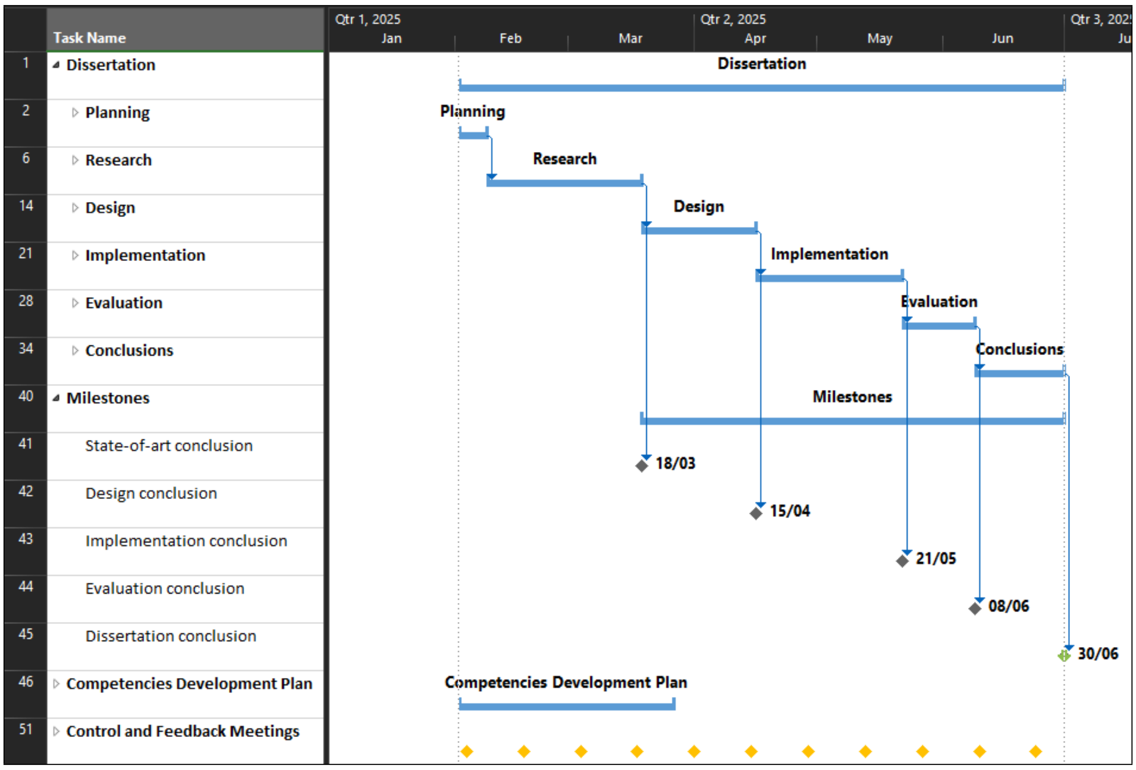
4.2 Work Breakdown Structure

The objective of the Work Breakdown Structure (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. As shown in Figure 4.1, the main 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, because 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.

Due to the extension of the WBS dictionary the table it is allocated on the Appendix A Planning Detailing by the Table 5.1. This table details each phase in order to be defined what are the goals and the acceptance criteria in a concise way.



	Task Name
1	◢ Dissertation
2	▷ Planning
6	▷ Research
14	▷ Design
21	▷ Implementation
28	▷ Evaluation
34	▷ Conclusions
40	◢ Milestones
41	State-of-art conclusion
42	Design conclusion
43	Implementation conclusion
44	Evaluation conclusion
45	Dissertation conclusion
46	▷ Competencies Development Plan
51	▷ Control and Feedback Meetings

Figure 4.2: Gantt timeline of the project

4.3 Gantt Diagram

This section provides details about the Gantt diagram, created using Microsoft Project. The timeline is illustrated in Figure 4.2, where only the main phases, milestones, and the periodic schedule of control and feedback meetings are shown for simplicity. Additionally, the tasks covering the development of competencies are highlighted. Detailed information is provided below, offering an overview that demonstrates how the project's organization aligns with the Work Breakdown Structure (WBS). For more in-depth details, refer to Appendix A Planning Detailing for a representation of all tasks related.

4.3.1 Project Management and Scheduling

The project's start date was set for 02/02/2025, following the first semester exams, with the end date aligned to the predefined project formalization deadline of 30/06/2025, as illustrated in Figure 4.2. The scheduling strategy employed automatic mode, enabling the calculation of dates based on task durations and their interdependencies. The schedule assumes a calendar without restrictions on working days, meaning all days, including weekends, are treated as working days to ensure continuous progress.

Estimation Rationale. Task durations were estimated by evaluating the complexity and importance of each project phase like illustrated by the column *"Duration"* in Figure 4.3. The background and state-of-the-art have a considerable slice, given their foundational role in the research. These stages demand research and analysis, making them time-consuming. Following this, the design phase received significant attention, as it involves defining system requirements and developing a detailed testing plan, which are important steps for the implementation.

The implementation phase was planned taking in considerations the necessity of prototyping, testing, and possible adjustments, making it the most time-consuming task. The evaluation phase, particularly benchmarking, was allocated less time compared to implementation, but enough time for a thorough analysis and interpretation of results. Finally, the conclusion phase was planned with a related similarly time to the state-of-art to synthesize findings, document outcomes, and finalize the dissertation.

Given the dependency between phases, this project is difficult to parallelize. The implementation phase relies heavily on the preceding design phase, and the evaluation phase depends on a functional implementation. While test-driven development (TDD) methodologies could be considered in some projects, they are unsuitable here due to the need for a fully realized implementation to generate reliable metric data for evaluation.

Resource Allocation. A resource column was included in the project plan to track resource allocation, focusing on the student and advisor as the primary stakeholders as illustrated in Figure 4.2. The advisor's role is focused on controlling and monitoring tasks, providing feedback and guidance throughout the project.

Cost Management. The cost component was excluded from the project plan, as it is not applicable to this type of academic project.

4.3.2 Monitoring and Controlling Procedures

The strategy defined to have control over the progress of each task was to mark its progress by an estimated percentage of completion. This strategy ensures that the progress of the task is clearly visible and measurable, giving both the student and the advisor a view of progress, like it is possible to observe the *"% Completed"* column on the Figure 4.3.

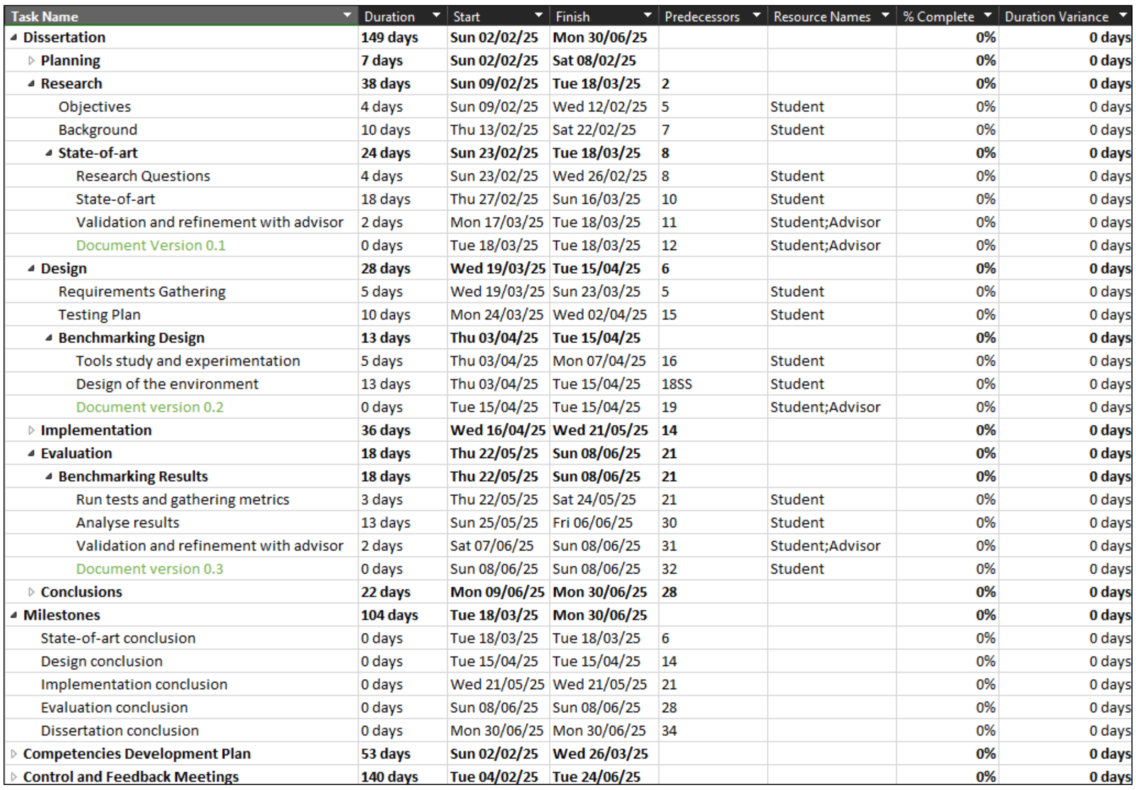
To improve this process, specific tasks dedicated to monitoring and control have been incorporated, such as *"Validation and Refinement with the Advisor"*, like illustrated in Figure 4.3. These tasks have two main objectives, firstly, to make the student responsible for presenting the partial document ready to be assessed. Secondly, to ensure that the advisor is warned in advance of the need for closer and more active feedback on these pre-defined days. Feedback can be given asynchronously via messages or synchronously during scheduled meetings. After the feedback time is up, the goal is to be able to present a refined partial document, referred to as the *"Document version X"* task, also possible to observe in Figure 4.3.

Additionally, milestones have been defined to mark the completion of each significant project phase. These milestones serve as checkpoints to ensure progress is on track and can be observed in Figure 4.3 under the main task *"Milestones"*.

To manage potential delays, a baseline has been established. This baseline records the initial schedule, allowing deviations to be tracked throughout the project. This mechanism provides an overview of delays and their impact on the schedule. The column *"Duration Variance"* column in Figure 4.3 illustrates this feature, allowing a visualization of changes between planned and actual progress.

4.3.3 Meeting Sessions

To ensure consistent communication and effective progress monitoring with the advisor, a series of biweekly meetings has been scheduled on Wednesdays, with each session expected to last between 30 minutes and 1 hour. While the schedule includes a predefined list of sessions,



Task Name	Duration	Start	Finish	Predecessors	Resource Names	% Complete	Duration Variance
Dissertation	**149 days**	**Sun 02/02/25**	**Mon 30/06/25**			**0%**	**0 days**
Planning	**7 days**	**Sun 02/02/25**	**Sat 08/02/25**			**0%**	**0 days**
Research	**38 days**	**Sun 09/02/25**	**Tue 18/03/25**	**2**		**0%**	**0 days**
Objectives	4 days	Sun 09/02/25	Wed 12/02/25	5	Student	0%	0 days
Background	10 days	Thu 13/02/25	Sat 22/02/25	7	Student	0%	0 days
State-of-art	**24 days**	**Sun 23/02/25**	**Tue 18/03/25**	**8**		**0%**	**0 days**
Research Questions	4 days	Sun 23/02/25	Wed 26/02/25	8	Student	0%	0 days
State-of-art	18 days	Thu 27/02/25	Sun 16/03/25	10	Student	0%	0 days
Validation and refinement with advisor	2 days	Mon 17/03/25	Tue 18/03/25	11	Student;Advisor	0%	0 days
Document Version 0.1	0 days	Tue 18/03/25	Tue 18/03/25	12	Student;Advisor	0%	0 days
Design	**28 days**	**Wed 19/03/25**	**Tue 15/04/25**	**6**		**0%**	**0 days**
Requirements Gathering	5 days	Wed 19/03/25	Sun 23/03/25	5	Student	0%	0 days
Testing Plan	10 days	Mon 24/03/25	Wed 02/04/25	15	Student	0%	0 days
Benchmarking Design	**13 days**	**Thu 03/04/25**	**Tue 15/04/25**			**0%**	**0 days**
Tools study and experimentation	5 days	Thu 03/04/25	Mon 07/04/25	16	Student	0%	0 days
Design of the environment	13 days	Thu 03/04/25	Tue 15/04/25	18SS	Student	0%	0 days
Document version 0.2	0 days	Tue 15/04/25	Tue 15/04/25	19	Student;Advisor	0%	0 days
Implementation	**36 days**	**Wed 16/04/25**	**Wed 21/05/25**	**14**		**0%**	**0 days**
Evaluation	**18 days**	**Thu 22/05/25**	**Sun 08/06/25**	**21**		**0%**	**0 days**
Benchmarking Results	**18 days**	**Thu 22/05/25**	**Sun 08/06/25**	**21**		**0%**	**0 days**
Run tests and gathering metrics	3 days	Thu 22/05/25	Sat 24/05/25	21	Student	0%	0 days
Analyse results	13 days	Sun 25/05/25	Fri 06/06/25	30	Student	0%	0 days
Validation and refinement with advisor	2 days	Sat 07/06/25	Sun 08/06/25	31	Student;Advisor	0%	0 days
Document version 0.3	0 days	Sun 08/06/25	Sun 08/06/25	32	Student	0%	0 days
Conclusions	**22 days**	**Mon 09/06/25**	**Mon 30/06/25**	**28**		**0%**	**0 days**
Milestones	**104 days**	**Tue 18/03/25**	**Mon 30/06/25**			**0%**	**0 days**
State-of-art conclusion	0 days	Tue 18/03/25	Tue 18/03/25	6		0%	0 days
Design conclusion	0 days	Tue 15/04/25	Tue 15/04/25	14		0%	0 days
Implementation conclusion	0 days	Wed 21/05/25	Wed 21/05/25	21		0%	0 days
Evaluation conclusion	0 days	Sun 08/06/25	Sun 08/06/25	28		0%	0 days
Dissertation conclusion	0 days	Mon 30/06/25	Mon 30/06/25	34		0%	0 days
Competencies Development Plan	**53 days**	**Sun 02/02/25**	**Wed 26/03/25**			**0%**	**0 days**
Control and Feedback Meetings	**140 days**	**Tue 04/02/25**	**Tue 24/06/25**			**0%**	**0 days**

Figure 4.3: Monitoring and control procedures displayed on the Gantt chart.

it remains flexible, allowing adjustments to the frequency of meetings as the project evolves. For instance, the number of meetings may increase during the final stages of the project, at which point the Gantt chart should be updated accordingly.

The meeting schedule is illustrated in Figure 4.4, which includes eleven recurring tasks organized under the main task *"Meeting Sessions"*. These meetings are planned to take place online.

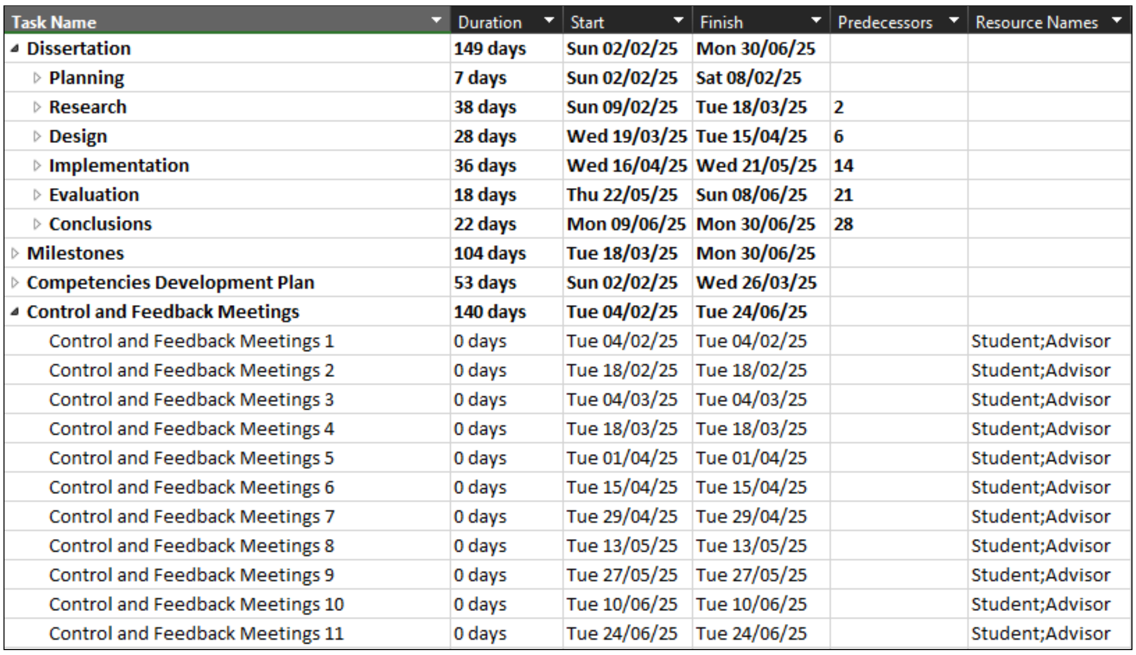
4.3.4 Competencies Development Plan

To address the competencies identified during the diagnosis of critical skills required for the completion of the dissertation, a dedicated section titled *"Competencies Development Plan"* has been incorporated into the project plan, as illustrated in Figure 4.5. This section outlines targeted tasks designed to address these areas for improvement.

For stress management and resilience, the plan includes attending a Stress Management course. [1]. To enhance self-discipline and time management, the initial task involves identifying and installing at least one productivity application, such as tools that restrict smartphone usage during certain periods, for example. Additionally, time management competence is further explored by taking the Time Management Fundamentals course [2].

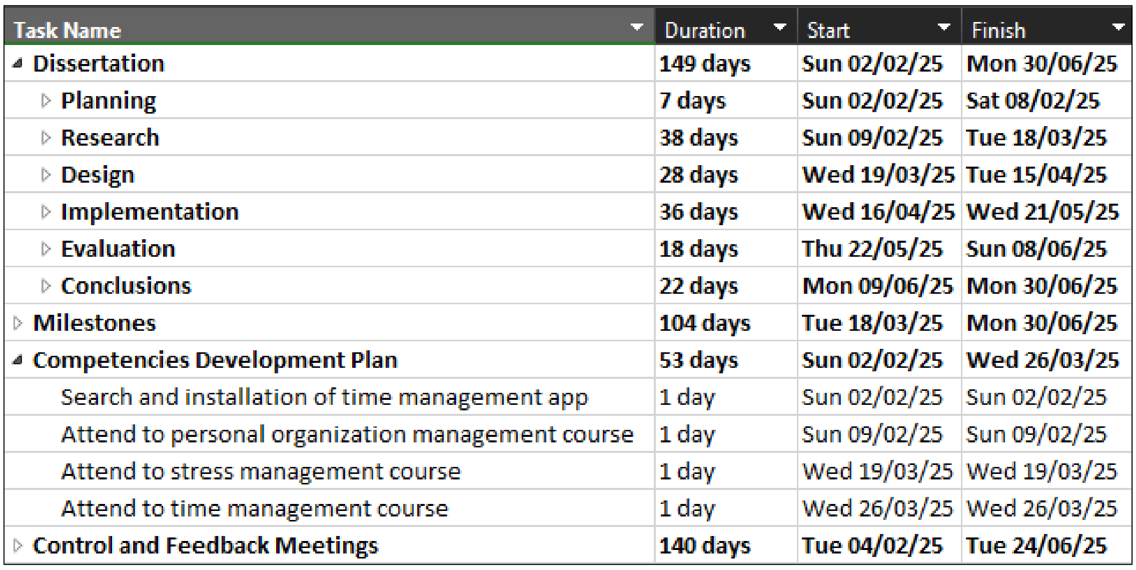
[1]Managing Stress Course: `https://www.linkedin.com/learning/managing-stress-2019/` (accessed 4 December 2024)

[2]Time Management Fundamentals Course: `https://www.linkedin.com/learning/time-management-fundamentals-14548057/the-power-of-managing-your-time/` (accessed 4 December 2024)



Task Name	Duration	Start	Finish	Predecessors	Resource Names
Dissertation	**149 days**	**Sun 02/02/25**	**Mon 30/06/25**		
Planning	**7 days**	**Sun 02/02/25**	**Sat 08/02/25**		
Research	**38 days**	**Sun 09/02/25**	**Tue 18/03/25**	**2**	
Design	**28 days**	**Wed 19/03/25**	**Tue 15/04/25**	**6**	
Implementation	**36 days**	**Wed 16/04/25**	**Wed 21/05/25**	**14**	
Evaluation	**18 days**	**Thu 22/05/25**	**Sun 08/06/25**	**21**	
Conclusions	**22 days**	**Mon 09/06/25**	**Mon 30/06/25**	**28**	
Milestones	**104 days**	**Tue 18/03/25**	**Mon 30/06/25**		
Competencies Development Plan	**53 days**	**Sun 02/02/25**	**Wed 26/03/25**		
Control and Feedback Meetings	**140 days**	**Tue 04/02/25**	**Tue 24/06/25**		
Control and Feedback Meetings 1	0 days	Tue 04/02/25	Tue 04/02/25		Student;Advisor
Control and Feedback Meetings 2	0 days	Tue 18/02/25	Tue 18/02/25		Student;Advisor
Control and Feedback Meetings 3	0 days	Tue 04/03/25	Tue 04/03/25		Student;Advisor
Control and Feedback Meetings 4	0 days	Tue 18/03/25	Tue 18/03/25		Student;Advisor
Control and Feedback Meetings 5	0 days	Tue 01/04/25	Tue 01/04/25		Student;Advisor
Control and Feedback Meetings 6	0 days	Tue 15/04/25	Tue 15/04/25		Student;Advisor
Control and Feedback Meetings 7	0 days	Tue 29/04/25	Tue 29/04/25		Student;Advisor
Control and Feedback Meetings 8	0 days	Tue 13/05/25	Tue 13/05/25		Student;Advisor
Control and Feedback Meetings 9	0 days	Tue 27/05/25	Tue 27/05/25		Student;Advisor
Control and Feedback Meetings 10	0 days	Tue 10/06/25	Tue 10/06/25		Student;Advisor
Control and Feedback Meetings 11	0 days	Tue 24/06/25	Tue 24/06/25		Student;Advisor

Figure 4.4: Meeting sessions represented on the Gantt chart.



Task Name	Duration	Start	Finish
Dissertation	**149 days**	**Sun 02/02/25**	**Mon 30/06/25**
Planning	**7 days**	**Sun 02/02/25**	**Sat 08/02/25**
Research	**38 days**	**Sun 09/02/25**	**Tue 18/03/25**
Design	**28 days**	**Wed 19/03/25**	**Tue 15/04/25**
Implementation	**36 days**	**Wed 16/04/25**	**Wed 21/05/25**
Evaluation	**18 days**	**Thu 22/05/25**	**Sun 08/06/25**
Conclusions	**22 days**	**Mon 09/06/25**	**Mon 30/06/25**
Milestones	**104 days**	**Tue 18/03/25**	**Mon 30/06/25**
Competencies Development Plan	**53 days**	**Sun 02/02/25**	**Wed 26/03/25**
Search and installation of time management app	1 day	Sun 02/02/25	Sun 02/02/25
Attend to personal organization management course	1 day	Sun 09/02/25	Sun 09/02/25
Attend to stress management course	1 day	Wed 19/03/25	Wed 19/03/25
Attend to time management course	1 day	Wed 26/03/25	Wed 26/03/25
Control and Feedback Meetings	**140 days**	**Tue 04/02/25**	**Tue 24/06/25**

Figure 4.5: Competencies development plan represented on the Gantt chart.

Communication, another key area of focus, will be developed by attending to the "Communicating with Confidence" course [3].

4.4 Risk Management

Effective risk management seeks to transform potential uncertainties into more predictable and controlled outcomes. To achieve this, the most significant risks identified are described next

4.4.1 Risk 1: Bugs in Third-Party Libraries

Description: There is a potential risk of encountering bugs in third-party libraries, which could compromise the viability of implementation and testing of the prototypes. Since the project relies on external libraries to implement fault-tolerant strategies, the stability and reliability of these libraries are important.

Cause: The cause of this risk is the need of trust on external software components.

Effect on the Project: Errors in the libraries can compromise the viability of the prototype's development and also the integrity of the results.

Risk Owner: Student.

Probability. 2. **Impact:** 4. **PI Score.** 8.

Risk Response Strategy: To mitigate this risk, alternative libraries will be identified for each strategy and language in advance. At least two or three libraries will be shortlisted and prioritized. If the primary library encounters significant bugs or issues, the next library on the list will be utilized.

Expected Result Without Action: If no action is taken, delays in prototypes development will occur.

Risk Response Type: Mitigation.

Response Description: A proactive approach will be taken to evaluate multiple libraries during the research phase.

4.4.2 Risk 2: Integration Challenges Between Components

Description: Integration issues could arise when combining multiple components, such as third-party libraries, testing frameworks, and custom implementations.

Cause: Differences in interfaces, versions, or dependencies among components used in the project.

Effect on the Project: These challenges could delay testing and result in compatibility issues that reduce productivity.

Risk Owner: Student.

Probability. 2. **Impact:** 4. **PI Score.** 8.

[3]Communicating with Confidence Course: `https://www.linkedin.com/learning/como-se-comunicar-com-confianca/` (accessed 4 December 2024)

Risk Response Strategy: To mitigate this risk, dependencies and versions will be carefully managed using dependency management tools (e.g., *mix* for Elixir, *go.mod* for Go, *sbt* for Scala). Component integration will also be tested incrementally to identify issues early.

Expected Result Without Action: Significant delays during the integration of components on the implementation phase.

Risk Response Type: Mitigation.

Response Description: Incremental integration practices will ensure smoother component interaction.

Chapter 5

Appendix A Planning Detailing

5.1 Gantt Chart

Figure 5.1 presents a detailed Gantt chart that outlines all project sub-tasks and their respective timelines, showcasing the relationships and dependencies between tasks. While parallelism could potentially provide more time for individual tasks, achieving it in this project is challenging, as noted in the main document. However, a Start-to-Start relationship is established between the task of studying a language and the implementation or environment design. This decision reflects the importance of dedicating time to the study phase while allowing for parallel progress to optimize the schedule.

For clarity, the variance column has been omitted from the chart, as its details are thoroughly addressed in the main document. Similarly, the baseline, set at the project's inception, has been excluded to avoid unnecessary visual complexity in the diagram.

5.2 Work Breakdown Structure Dictionary

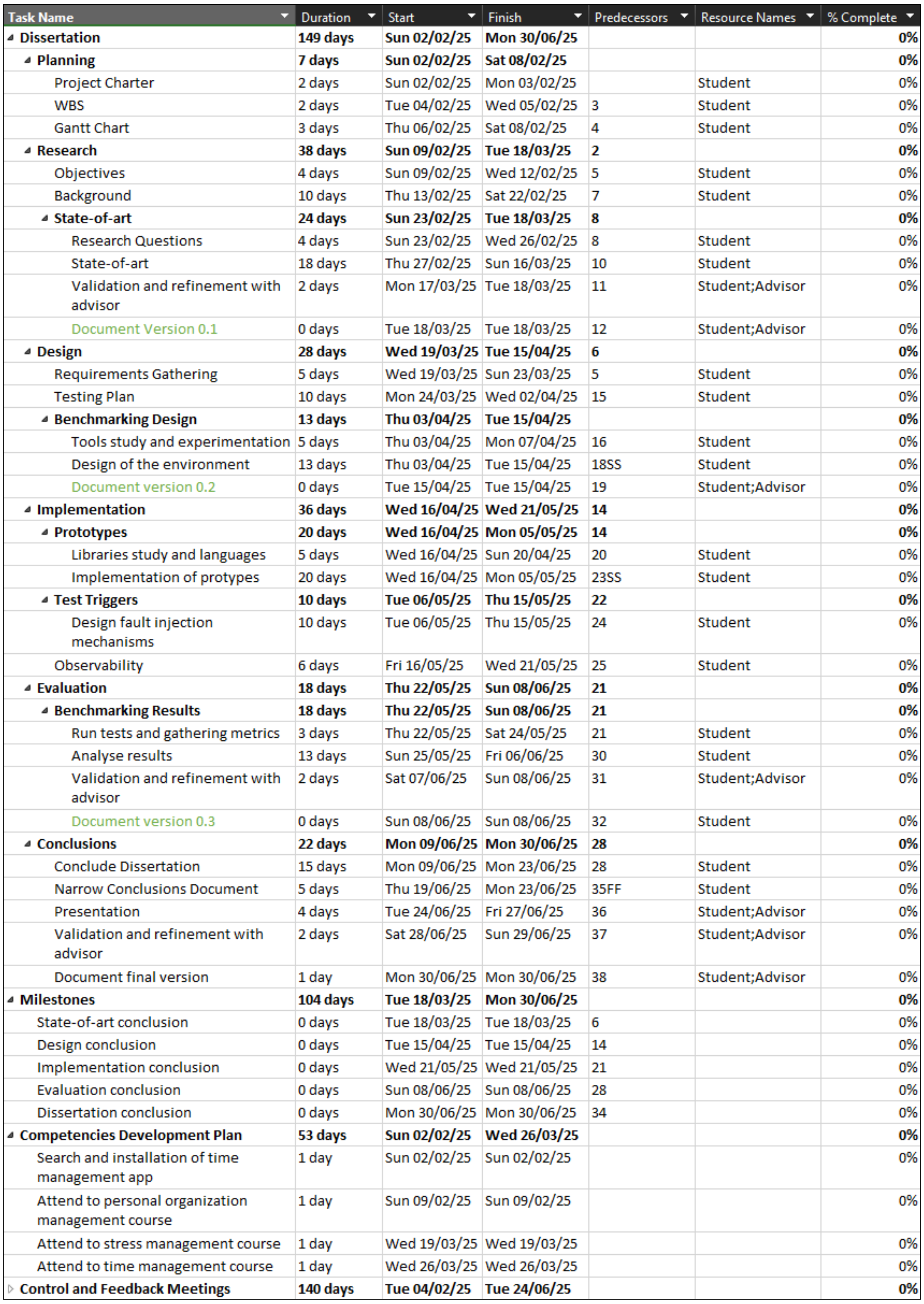
On the Table 5.1 it is represented in a detailed way the description of the WBS's deliverables, such as the work loads. For each item there is a concise descriptions and a acceptance criteria.

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 advisor.
(1.1.2) WBS	Deliverable	The WBS should break down the project into manageable components. **Acceptance Criteria:** The WBS should be validated by the advisor and include all project elements.
(1.1.3) Gantt Chart	Deliverable	A detailed timeline outlining tasks, dependencies, competence development plan, milestones, and the dissertation deadline. **Acceptance Criteria:** The Gantt chart must accurately reflect project phases and be reviewed by the advisor.

Item Name	Type of Item	Additional Description / Acceptance Criteria
(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 advisor.
(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 advisor, and must include a clear justification for the languages chosen.
(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 advisor.
(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 advisor.
(1.3.3) Benchmarking Design	Deliverable	Define the design for benchmarking environments. **Acceptance Criteria:** Benchmarking environments design must be validated by the advisor, 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.

Item Name	**Type of Item**	**Additional Description / Acceptance Criteria**
(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 advisor.
(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 academic 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.

Table 5.1: WBS dictionary



Task Name	Duration	Start	Finish	Predecessors	Resource Names	% Complete
Dissertation	**149 days**	**Sun 02/02/25**	**Mon 30/06/25**			**0%**
Planning	**7 days**	**Sun 02/02/25**	**Sat 08/02/25**			**0%**
Project Charter	2 days	Sun 02/02/25	Mon 03/02/25		Student	0%
WBS	2 days	Tue 04/02/25	Wed 05/02/25	3	Student	0%
Gantt Chart	3 days	Thu 06/02/25	Sat 08/02/25	4	Student	0%
Research	**38 days**	**Sun 09/02/25**	**Tue 18/03/25**	**2**		**0%**
Objectives	4 days	Sun 09/02/25	Wed 12/02/25	5	Student	0%
Background	10 days	Thu 13/02/25	Sat 22/02/25	7	Student	0%
State-of-art	**24 days**	**Sun 23/02/25**	**Tue 18/03/25**	**8**		**0%**
Research Questions	4 days	Sun 23/02/25	Wed 26/02/25	8	Student	0%
State-of-art	18 days	Thu 27/02/25	Sun 16/03/25	10	Student	0%
Validation and refinement with advisor	2 days	Mon 17/03/25	Tue 18/03/25	11	Student;Advisor	0%
Document Version 0.1	0 days	Tue 18/03/25	Tue 18/03/25	12	Student;Advisor	0%
Design	**28 days**	**Wed 19/03/25**	**Tue 15/04/25**	**6**		**0%**
Requirements Gathering	5 days	Wed 19/03/25	Sun 23/03/25	5	Student	0%
Testing Plan	10 days	Mon 24/03/25	Wed 02/04/25	15	Student	0%
Benchmarking Design	**13 days**	**Thu 03/04/25**	**Tue 15/04/25**			**0%**
Tools study and experimentation	5 days	Thu 03/04/25	Mon 07/04/25	16	Student	0%
Design of the environment	13 days	Thu 03/04/25	Tue 15/04/25	18SS	Student	0%
Document version 0.2	0 days	Tue 15/04/25	Tue 15/04/25	19	Student;Advisor	0%
Implementation	**36 days**	**Wed 16/04/25**	**Wed 21/05/25**	**14**		**0%**
Prototypes	**20 days**	**Wed 16/04/25**	**Mon 05/05/25**	**14**		**0%**
Libraries study and languages	5 days	Wed 16/04/25	Sun 20/04/25	20	Student	0%
Implementation of protypes	20 days	Wed 16/04/25	Mon 05/05/25	23SS	Student	0%
Test Triggers	**10 days**	**Tue 06/05/25**	**Thu 15/05/25**	**22**		**0%**
Design fault injection mechanisms	10 days	Tue 06/05/25	Thu 15/05/25	24	Student	0%
Observability	6 days	Fri 16/05/25	Wed 21/05/25	25	Student	0%
Evaluation	**18 days**	**Thu 22/05/25**	**Sun 08/06/25**	**21**		**0%**
Benchmarking Results	**18 days**	**Thu 22/05/25**	**Sun 08/06/25**	**21**		**0%**
Run tests and gathering metrics	3 days	Thu 22/05/25	Sat 24/05/25	21	Student	0%
Analyse results	13 days	Sun 25/05/25	Fri 06/06/25	30	Student	0%
Validation and refinement with advisor	2 days	Sat 07/06/25	Sun 08/06/25	31	Student;Advisor	0%
Document version 0.3	0 days	Sun 08/06/25	Sun 08/06/25	32	Student	0%
Conclusions	**22 days**	**Mon 09/06/25**	**Mon 30/06/25**	**28**		**0%**
Conclude Dissertation	15 days	Mon 09/06/25	Mon 23/06/25	28	Student	0%
Narrow Conclusions Document	5 days	Thu 19/06/25	Mon 23/06/25	35FF	Student	0%
Presentation	4 days	Tue 24/06/25	Fri 27/06/25	36	Student;Advisor	0%
Validation and refinement with advisor	2 days	Sat 28/06/25	Sun 29/06/25	37	Student;Advisor	0%
Document final version	1 day	Mon 30/06/25	Mon 30/06/25	38	Student;Advisor	0%
Milestones	**104 days**	**Tue 18/03/25**	**Mon 30/06/25**			**0%**
State-of-art conclusion	0 days	Tue 18/03/25	Tue 18/03/25	6		0%
Design conclusion	0 days	Tue 15/04/25	Tue 15/04/25	14		0%
Implementation conclusion	0 days	Wed 21/05/25	Wed 21/05/25	21		0%
Evaluation conclusion	0 days	Sun 08/06/25	Sun 08/06/25	28		0%
Dissertation conclusion	0 days	Mon 30/06/25	Mon 30/06/25	34		0%
Competencies Development Plan	**53 days**	**Sun 02/02/25**	**Wed 26/03/25**			**0%**
Search and installation of time management app	1 day	Sun 02/02/25	Sun 02/02/25			0%
Attend to personal organization management course	1 day	Sun 09/02/25	Sun 09/02/25			0%
Attend to stress management course	1 day	Wed 19/03/25	Wed 19/03/25			0%
Attend to time management course	1 day	Wed 26/03/25	Wed 26/03/25			0%
Control and Feedback Meetings	**140 days**	**Tue 04/02/25**	**Tue 24/06/25**			**0%**

Figure 5.1: Complete demonstration of the Gantt

Bibliography

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

[2] S. Andrew Tanenbaum and M. Maarten Van Steen. *Distributed systems*. 4th ed. Maarten Van Steen, 2023. isbn: 978-90-815406-3-6.

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

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

[5] 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`.

[6] Dipankar Deb, Rajeeb Dey, and Valentina E. Balas. *Engineering Research Methodology*. Dec. 2018. doi: `10.1007/978-981-13-2947-0`. url: `https://doi.org/10.1007/978-981-13-2947-0`.

[7] NSPE. *NSPE Code of Ethics for Engineers*. Accessed at 01.12.2024. url: `https://www.nspe.org/resources/ethics/code-ethics`.

[8] IEEE. *IEEE Code of Ethics*. Accessed at 01.12.2024. url: `https://www.ieee.org/about/corporate/governance/p7-8.html`.

[9] ACM. *ACM Code of Ethics and Professional Conduct*. Accessed at 01.12.2024. url: `https://www.acm.org/code-of-ethics/`.

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

[11] 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`.

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

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

[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] 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`.

[17] 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`.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] Martin Fowler. *Circuit Breaker*. 2014. url: https://martinfowler.com/bliki/CircuitBreaker.html.

[24] Heidi Howard and Richard Mortier. "Paxos vs Raft: Have we reached consensus on distributed consensus?" In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2020*. Association for Computing Machinery, Inc, Apr. 2020. isbn: 9781450375245. doi: 10.1145/3380787.3393681.

[25] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. isbn: 978-1-931971-10-2. url: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[26] *Apache Flink Documentation*. Accessed at 03.11.2024. url: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/fault-tolerance/checkpointing/.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] Attila Sragli. *Optimising for Concurrency: Comparing and contrasting the BEAM and JVM virtual machines*. Accessed at 03.11.2024. Nov. 2024. url: https://www.erlang-solutions.com/blog/optimising-for-concurrency-comparing-and-contrasting-the-beam-and-jvm-virtual-machines/.

[41] *Elixir Language Documentation*. Accessed at 03.11.2024. url: https://hexdocs.pm/elixir/introduction.html.

[42] *Akka Official Documentation*. Accessed at 03.11.2024. 2024. url: https://doc.akka.io/libraries/akka-core/current/typed/actors.html/.

[43] *Proto.Actor Documentation*. Accessed at 03.11.2024. 2024. url: https://proto.actor/docs/.

[44] *Elixir School - Official*. Accessed at 03.11.2024. 2024. url: https://elixirschool.com/.

[45] Matthew Alan Le Brun, Duncan Paul Attard, and Adrian Francalanza. “Graft: general purpose raft consensus in Elixir”. In: *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang* (Aug. 2021), pp. 2–14. doi: 10.1145/3471871.3472963.

[46] Mauricio Cassola et al. “A Gradual Type System for Elixir”. In: *ACM International Conference Proceeding Series*. Association for Computing Machinery, Oct. 2020, pp. 17–24. isbn: 9781450389433. doi: 10.1145/3427081.3427084.

[47] Ahmed Abdel Moamen, Dezhong Wang, and Nadeem Jamali. “Supporting Resource Control for Actor Systems in Akka”. In: *Proceedings - International Conference on Distributed Computing Systems*. Institute of Electrical and Electronics Engineers Inc., July 2017, pp. 2642–2645. isbn: 9781538617915. doi: 10.1109/ICDCS.2017.291.

[48] Mehdi Bagherzadeh et al. “Actor concurrency bugs: A comprehensive study on symptoms, root causes, API usages, and differences”. In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA Nov. 2020). issn: 24751421. doi: 10.1145/3428282.

[49] Mohsen Moradi Moghadam et al. “Akka: Mutation Testing for Actor Concurrency in Akka using Real-World Bugs”. In: *ESEC/FSE 2023 - Proceedings of the 31st ACM*

Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, Inc, Nov. 2023, pp. 262–274. isbn: 9798400703270. doi: 10.1145/3611643.3616362.

[50] Ishu Chaudhary et al. "Generational ZGC- An Improvement in Garbage Collector in Java 21". In: *Proceedings of International Conference on Communication, Computer Sciences and Engineering, IC3SE 2024*. Institute of Electrical and Electronics Engineers Inc., 2024, pp. 631–636. isbn: 9798350366846. doi: 10.1109/IC3SE62002.2024.10593533.

[51] Brian Kennedy. *Go in Action*. 2016. url: www.allitebooks.com.

[52] Katherine. Cox-Buday. *Concurrency in Go : tools and techniques for developers*. O'Reilly Media, 2017. isbn: 9781491941195.

[53] David Castro et al. "Distributed Programming using Role-Parametric Session Types in Go: Statically-typed endpoint APIs for dynamically-instantiated communication structures". In: *Proceedings of the ACM on Programming Languages* 3 (POPL Jan. 2019). issn: 24751421. doi: 10.1145/3290342.

[54] Alexander. Shuiskov. *Microservices With GO : Building Scalable and Reliable Go Microservices*. PACKT PUBLISHING LIMITED, 2022. isbn: 9781804617007.

[55] Junxian Zhao et al. "Let It Go: Relieving Garbage Collection Pain for Latency Critical Applications in Golang". In: *HPDC 2023 - Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. Association for Computing Machinery, Inc, Aug. 2023, pp. 169–180. isbn: 9798400701559. doi: 10.1145/3588195.3592998.

[56] Jiayi Zhang. "Performance Comparative Analysis on Garbage First Garbage Collector and Z Garbage Collector". In: *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer, ICFTIC 2021*. Institute of Electrical and Electronics Engineers Inc., 2021, pp. 733–740. isbn: 9781665406055. doi: 10.1109/ICFTIC54370.2021.9647167.

[57] James Whitney, Chandler Gifford, and Maria Pantoja. "Distributed execution of communicating sequential process-style concurrency: Golang case study". In: *Journal of Supercomputing* 75 (3 Mar. 2019), pp. 1396–1409. issn: 15730484. doi: 10.1007/s11227-018-2649-2.

[58] *Go-kit Documentation*. Accessed at 03.11.2024. 2024. url: https://gokit.io/.

[59] Yaroslav Marchuk, Bohdan Melnyk, and Nataliya Melnyk. "Analysis of the Speed of Execution of Business Logic in Applications Created in Different Software Environments". In: *Proceedings - International Conference on Advanced Computer Information Technologies, ACIT*. 2023, pp. 357–360. doi: 10.1109/ACIT58437.2023.10275631.

[60] *Discord blog - Why discord is switching from go to rust*. Accessed at 03.11.2024. 2024. url: https://discord.com/blog/why-discord-is-switching-from-go-to-rust/.

[61] Raquel Almeida and Marco Vieira. *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM Digital Library, 2013. isbn: 9781450305754. doi: 10.1145/1988008.1988035.

[62] Sebastian Blessing et al. "Run, actor, run towards cross-actor language benchmarking". In: *AGERE 2019 - Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, co-located with SPLASH 2019*. Association for Computing Machinery, Inc, Oct. 2019, pp. 41–50. isbn: 9781450369824. doi: 10.1145/3358499.3361224.

[63] Shams Imam and Vivek Sarkar. "Savina - An actor benchmark suite: Enabling empirical evaluation of actor libraries". In: *AGERE! 2014 - Proceedings of the 2014 ACM*

SIGPLAN Workshop on Programming Based on Actors, Agents, and Decentralized Control, Part of SPLASH 2014. Association for Computing Machinery, Oct. 2014, pp. 67–80. isbn: 9781450321891. doi: 10.1145/2687357.2687368.

[64] Rafael C. Cardoso et al. “Towards benchmarking actor- and agent-based programming languages”. In: *AGERE! 2013 - Proceedings of the 2013 ACM Workshop on Programming Based on Actors, Agents, and Decentralized Control*. Association for Computing Machinery, 2013, pp. 115–125. isbn: 9781450326025. doi: 10.1145/2541329.2541339.