ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Comparative Analysis of Fault Tolerance in Elixir, Scala with Akka, and Go

Nuno Ribeiro

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

Advisor: Dr. Luís Nogueira

Porto, December 15, 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 15, 2024

Contents

List of Figures vii

List of Tables ix

1 Introduction 1
1.1 Context and Problem . . . 1
1.2 Objectives . . . 1
1.3 Ethical Considerations . . . 1
1.4 Document structure . . . 2

2 Background 3
2.1 Distributed Systems . . . 3
2.1.1 Characteristics . . . 3
Transparency . . . 3
Reliability and Availability . . . 3
Scalability . . . 4
Fault tolerance . . . 4
2.1.2 Communication . . . 4
2.1.3 Challenges . . . 4
2.2 Fault Tolerance . . . 5
2.2.1 Fault Tolerance Taxonomy . . . 5
2.2.2 Strategies . . . 6
Retry Mechanism . . . 6
Circuit Breaker Pattern . . . 7
Replication and Redundancy . . . 8
Check-pointing and Message Logging . . . 11
Briefly Comparison Between Strategies . . . 12
2.3 Distributed and Concurrency Programming . . . 12
2.3.1 Models and Paradigms . . . 13
Actor Model . . . 13
Communicating Sequential Processes . . . 13
Microservices Architectures . . . 14
2.3.2 Distributed and Concurrent Programming Languages . . . 14
Analyses and Language Choice Justification . . . 15

3 State of the art 17
3.1 Research Questions . . . 17
3.1.1 Research Methodology . . . 17
3.2 Elixir Programming Language Analysis . . . 19
3.2.1 The Foundation of Erlang . . . 19
Concurrency in Erlang and BEAM . . . 19

Garbage Collection and Immutability 20
Hot-code swapping 21
3.2.2 Fault Tolerance Mechanism and Strategies 22
Let It Crash Philosophy and Actor Model 22
Tools and Support 23
Drawbacks and Real Applications 24
3.3 Scala Programming Language with Akka Toolkit Analysis 25
3.4 Go Programming Language Analysis 25
3.5 Benchmarking Strategies 25
3.6 Conclusions 25

4 Project Planning **27**
4.1 Project Charter 27
4.2 Work Breakdown Structure 28
4.3 Gantt Diagram 31
4.3.1 Project Management and Scheduling 31
4.3.2 Monitoring and Controlling Procedures 32
4.3.3 Meeting Sessions 33
4.3.4 Competencies Development Plan 33
4.4 Risk Management 35
4.4.1 Risk 1: Bugs in Third-Party Libraries 35
4.4.2 Risk 2: Integration Challenges Between Components 36

5 Appendix A Gantt Details **37**

List of Figures

2.1 Diagram illustrating the states of a Circuit Breaker 7
2.2 Diagram illustrating the states of a node in the Raft algorithm 11

3.1 Concurrency in the Erlang virtual machine 20
3.2 Elixir/BEAM processes vs JVM threads 21
3.3 Supervisor tree pattern 23

4.1 The Work Breakdown Structure (WBS) of the project. 28
4.2 Gantt timeline of the project 31
4.3 Monitoring and control procedures displayed on the Gantt chart. 33
4.4 Meeting sessions represented on the Gantt chart. 34
4.5 Competencies development plan represented on the Gantt chart. 34

5.1 Complete demonstration of the Gantt 38

List of Tables

2.1 Brief description of fauls types 5
2.2 Brief description of failure types 6
2.3 Characteristics of distributed and concurrent programming languages . . . 16

4.1 WBS dictionary 31

Chapter 1

Introduction

1.1 Context and Problem

- Lack of information focusing on fault tolerance comparison

- Old information available

- Increase usage of microservices / distributed systems

1.2 Objectives

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 [1]. 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 [2]. 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 [3] and the Association for Computing Machinery Code (ACM) of Ethics and Professional Conduct [4], 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

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 [5]. These innovations made it possible to create systems distributed across nodes where tasks could be processed collectively to achieve a common goal [5]. Nodes in a distributed system may refer to physical devices or software processes [6].

To the end-user, distributed systems appear as a single, large virtual system, making the underlying logic transparent [6]. These systems achieve a shared objective by transmitting messages through various nodes and dividing computational tasks among them, increasing resilience and isolating business logic [6, 7]. 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 [5, 7].

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 [5, 8]. 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) [5, 9]. Replication transparency ensures reliability by masking data duplication, while failure transparency enables systems to handle faults without user disruption [5, 9]. 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 [10]. A highly reliable system maintains consistent, uninterrupted service over an extended period, minimizing disruptions for users [5], 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 [5, 11].

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 [5, 6, 12]. 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 [13]. Cloud services must support a large volume of simultaneous connections and interactions, making scalability a crucial factor [5].

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 [5]. 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 [7].

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 [14].

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 [6]. Synchronous communication involves blocking operations where the sender waits for a response, suitable for scenarios requiring confirmation [5, 9]. 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 [5].

2.1.3 Challenges

Distributed systems encounter numerous challenges, including scalability [10], managing software, network, and disk failures [15, 16], heterogeneity [9], coordination among nodes, and difficulties on debugging and testing [6, 16]. 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 [5, 6]. This theorem underlines a critical trade-off in distributed systems: only two of these properties can be fully ensured at any given time [17]. 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 [5, 6, 17]. 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 [7]. 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 [5, 18].

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. [19]. 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 [19, 20].

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 [19].
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 [19].
Permanent Faults	Persistent issues that continue until the root cause is identified and addressed. Relatively straightforward to fix, typically related to complete component malfunction [5].
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 [19].

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 [5].

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. [5]
Omission Failure	The system fails to send or receive necessary messages, impacting communication and task coordination. [19]
Timing Failure	The system's response occurs outside a specified time interval, either too early or too late, causing issues in time-sensitive operations. [19]
Response Failure	The system provides incorrect outputs or deviates from expected state transitions, potentially leading to wrong results. [5]
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. [5]

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 [8]. 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 [6, 14].

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 [19]. 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 [14].

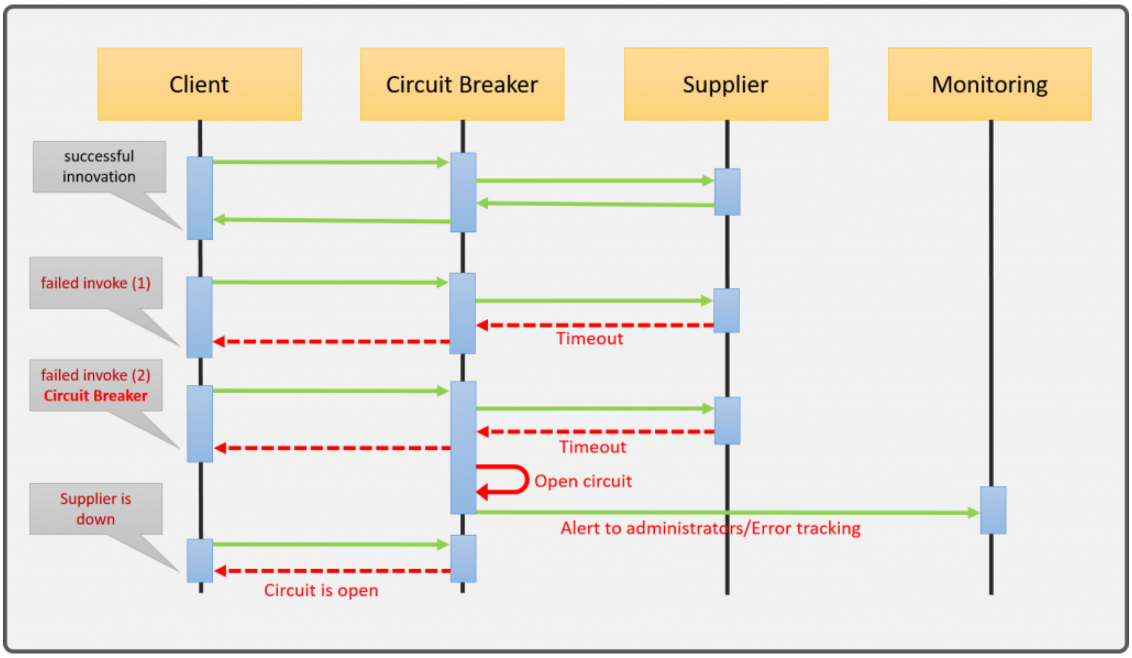

Figure 2.1: Diagram illustrating the states of a Circuit Breaker [1].

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 [19].

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 [6]. By actively monitoring the health of operations and selectively blocking problematic ones, circuit breakers act as safeguards against system overload and degradation [21].

Circuit breakers operate in three primary states: Closed, Open, and Half-Open [6, 21]. 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 [6, 21]. 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 [21].

[1] Adapted 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 [21]. 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 [6]. 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 [6]. 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 [7]. By creating multiple replicas of data or processes, replication eliminates single points of failure, ensuring system reliability, availability, and transparency [9]. 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 [5].

To effectively coordinate replicas and maintain consistency, replication mechanisms employ various strategies, which can be categorized as follows [19]:

- **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 [14].

Consensus Algorithms

Achieving consensus is essential in distributed systems to ensure that a group of processes operates cohesively as a single entity [5]. 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 [5].

Paxos

Paxos appeared in 1989 and has evolved over time, earning a reputation as a complex and difficult to understand algorithm [5]. 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 [5, 22]. The algorithm revolves around three distinct roles [9, 22]:

- **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 [5, 9, 22]:

- **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 [5]. 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 [9], 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 [5, 22].

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 [5, 23]. 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 [5]. 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 [6]. The typical number of nodes is five, which theoretically allows for two failures [23].

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 [5]. The term is reset every time an election occurs, starting from zero [23]. This information is then propagated to the followers using a process inspired by the two-phase commit protocol, where it consists on [5, 6]:

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 [5]. 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 [6, 23], 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 [5, 6].

The change of state of the node is displayed on the Figure 2.2 and the process consists on the following steps [6, 23]:

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.

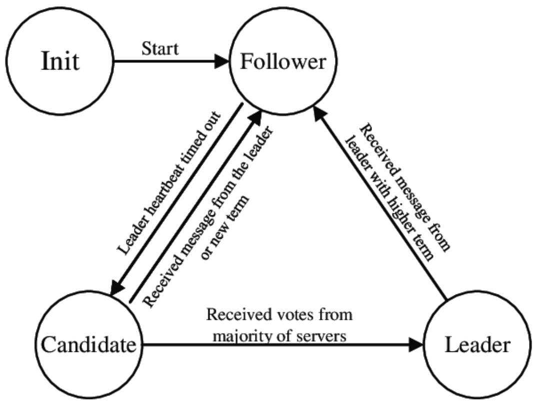

Figure 2.2: Diagram illustrating the states of a node in the Raft algorithm [2].

Once elected, the new leader reconciles any inconsistencies by broadcasting missing log entries to followers during subsequent operations, ensuring consistency across the cluster [5].

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 [22]. 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 [5].

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 [6, 22]. Paxos, on the other hand, is often found in legacy systems and specialized applications like Google's Chubby service [22].

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 [5].

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 [5]. 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 [5].

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 [5, 19]. Furthermore, some operations are inherently irreversible, which limits the effectiveness of this strategy [5]. 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 [8, 19].

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 [8, 19]. Although checkpoints are still required to avoid replaying all messages from the beginning, the overhead is significantly reduced compared to traditional check-pointing [19]. 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 [5].

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 [24]. 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 [25]. 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 [26], 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 [25]. This has driven the evolution of new paradigms, languages, frameworks, and libraries aimed at reducing development complexity in distributed and concurrent systems [12].

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 [27]. 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 [28]. Each actor is equipped with its own mailbox, which receives messages and processes them sequentially [29].

A core principle of the Actor Model is isolation, maintaining their own internal state that is inaccessible and immutable by others [29]. This eliminates the need for shared memory, reducing complexity and potential data races [12].

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 [28].

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 [30]. 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 [28]. 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 [31].

CSP offers an abstract and formal framework for modeling interactions between concurrent processes through channels, which serve as the communication medium between them [32].

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 [31]. 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 [12, 32, 33]. In addition, the language Occam attempts to offer a direct implementation of CSP principles with its focus on critical projects such as satellites [34].

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 [35]. 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 [35].

Microservices architectures allow general purpose programming languages to participate in distributed computing paradigms by leveraging frameworks, libraries, and microservices principles [36].

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 [12, 32]. 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 [12].

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 [12]. Erlang, for instance, is renowned for its fault tolerance and “let-it-crash” philosophy [25]. Supervising actors monitor and recover from failures, making Erlang highly suitable for building robust distributed systems [25]. 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 [37].

Haskell, a pure functional programming language, provides a deterministic approach to concurrency, ensuring consistent results regardless of execution order [12]. 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 [12]. 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 [34]. 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 [33].

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 Bogdan/Björn's Erlang Abstract Machine (BEAM) also know as Erlang Virtual Machine (VM) [37]. Since Elixir inherits all the strengths of Erlang [12], including fault tolerance and the Actor Model, a direct comparison with Erlang is unnecessary as they share the same

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

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

[8] Official website of 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

core runtime and strategies. Such a comparison would likely yield redundant results and add little value to the research.

On the other hand, comparing Elixir with low-level languages like Java, Rust, and C would also be less effective. These languages require explicit management of concurrency and fault tolerance [12], 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 [12]. 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 [34]. By examining how Go can achieve fault tolerance using libraries and abstractions under a microservices strategy, the study can assess whether an external abstraction layer can match or exceed the capabilities of languages with native support. This investigation could reveal whether the flexibility of a non-native distributed model can compensate for the lack of built-in features, providing valuable insights for developers operating in modern cloud-native environments.

Chapter 3

State of the art

3.1 Research Questions

For the literature review and stat 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, such as microservices, and identifies the metrics necessary for accurately measuring each case.

- **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 simulating distributed environments, and which metrics are most relevant for evaluating fault tolerance in these scenarios?

3.1.1 Research Methodology

This section outlines the research methodology adopted in this dissertation. It is important to note that this study addresses 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.

However, relevant groundwork can be found in the study by Valkov et al. [12], which compared Erlang, Go, and Scala in terms of inter-process communication latency, process creation time, the maximum number of supported processes, and throughput. This work serves as a valuable reference, but it does not delve into fault-tolerance aspects, which this dissertation seeks to address.

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.

IEEE Xplore Query:

```
"query": {
    ("Document Title": "Elixir" OR "Document Title": "Erlang") AND
    ("Document Title": "Go" OR "Document Title": "Golang") AND
    ("Document Title": "Akka" OR "Document Title": "Scala")
}
```

ACM Digital Library Query:

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

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 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:

```
"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, the methodology employed covered grey literature, particularly books, due to their mature, structured content. 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 simulating distributed environments, and which metrics are most relevant for evaluating fault tolerance in these scenarios?

...

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.

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 [25, 37].

Erlang, developed in the mids of 1980s by Ericsson, was specifically designed to support systems that are highly reliable, responsive, scalable, and always available [25, 37]. 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 more developer-friendly syntax, powerful metaprogramming capabilities with macros, and improved tooling, all while maintaining full compatibility with the Erlang runtime [37]. 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 Erlang and 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 [12, 37]. 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 operating system for management and lack the flexibility needed for fine-grained optimization. For instance, in the JVM, platform threads are a thin abstraction over OS threads, limiting control and optimization. However, virtual threads, introduced on Java 21, brings more capabilities to the JVM, where the Akka framework take advantage of [40].

In contrast, BEAM employs a concurrency-oriented programming model in which one thread per CPU core manages numerous lightweight processes. This architecture enables BEAM to handle parallelism by letting one scheduler per CPU handling multiple lightweight processes. This strategy is illustrated in Figure 3.1, where this architecture enables fault tolerance aspects over isolating the processes [37].

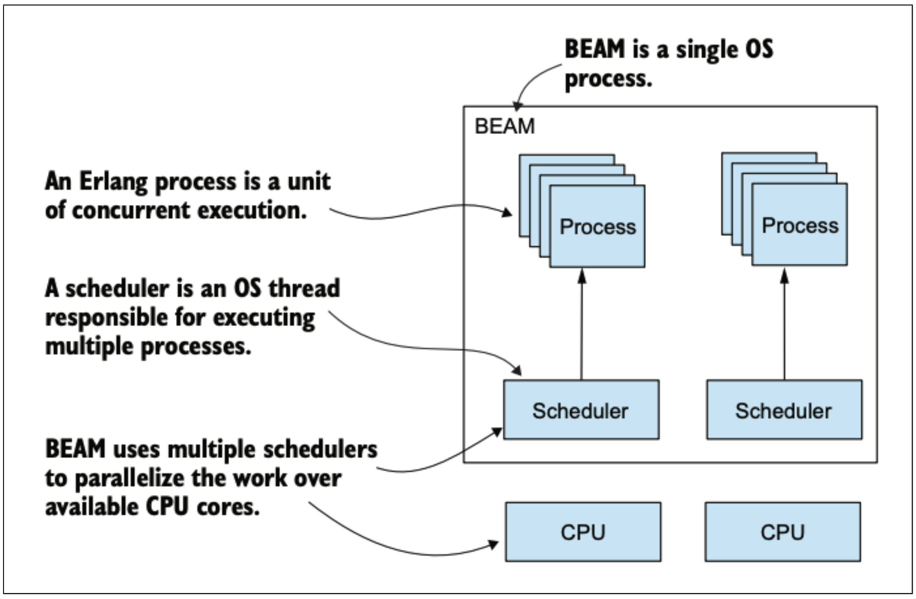

Figure 3.1: Concurrency in the Erlang virtual machine. Adapted from [37].

The BEAM scheduler is considered preemptive, meaning that assigns short execution time slices to each process. This ensures that long-running tasks do not monopolize system resources, promoting fairness and responsiveness [25]. 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 hinder the execution of others [37, 40].

In a direct comparison of Elixir's processes running on BEAM with the two threading techniques of the JVM, 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 threads, 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 [37].

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 [12, 40].

Garbage Collection and Immutability

Erlang and Elixir enforce immutability as a fundamental principle, ensuring that all data remains unchangeable. This eliminates many concurrency issues common in systems with

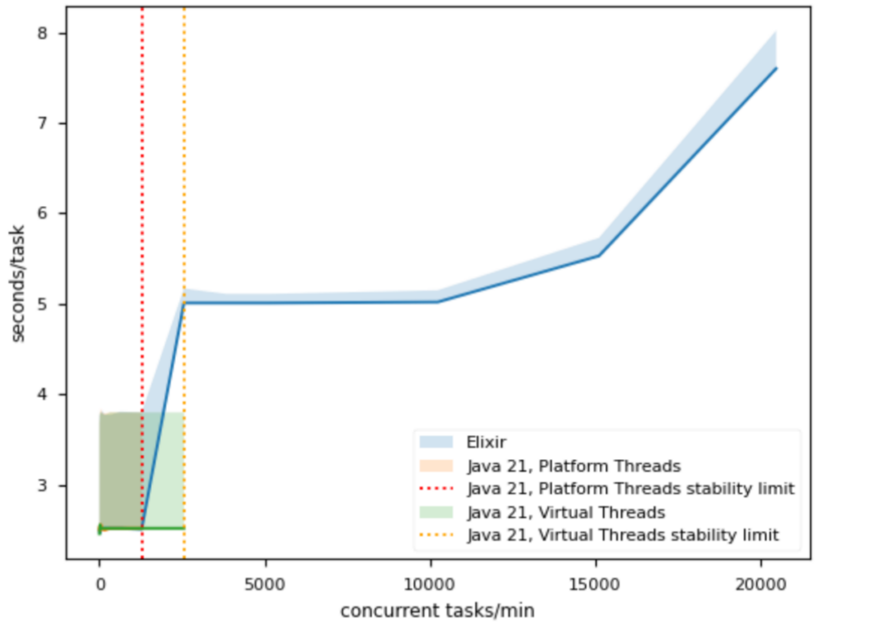

Figure 3.2: Elixir/BEAM processes vs JVM threads. Adapted from [40].

shared memory, such as race conditions. 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 [37, 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 [25, 37].

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 [12, 37].

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 [25].

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 [33].

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 [14, 25]. Instead of defensive 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 [37].

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 [25, 37]. 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 leverages this pattern to build fault-tolerant systems [12]. In this approach, processes are classified into two types [37]:

- **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 managing a group of processes. This structure provides modularity and ensures that fault recovery is localized, reducing the impact of failures [25].

Supervisors in Elixir operate using restart strategies to determine how to handle failures. The options given by the OTP Supervisors and the most common ones are the following [25, 37, 41]:

- **One-for-One:** If a single worker process fails, the supervisor restarts only that process.

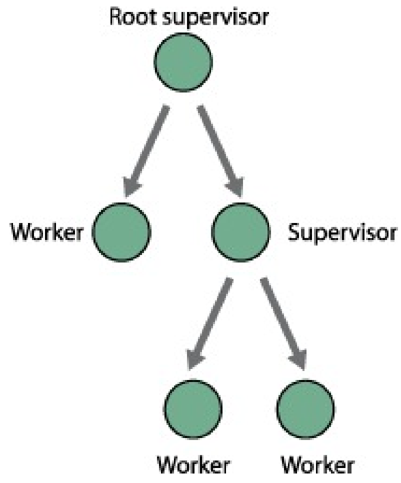

Figure 3.3: Supervisor tree pattern. Adapted from [37].

- **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 [25].

The One-for-One strategy is best suited for independent processes [25]. 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 [25]. 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. Additionally, in other paradigms such as Go, there are libraries capable of unifying CSP with the Actor Model, such as the proto.actor library.

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 [25, 37].

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, 42].

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, streamlining development workflows and contributing to the reliability of applications by ensuring consistent builds[41, 42].

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 et al. [43] in 2019, and Ra, developed by the RabbitMq team [1] , provide an implementation of the Raft consensus algorithm. Similarly, the Fuse library [2], 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 [37].

Lastly, it's important to mention the Elixir environment, which includes frameworks that enhance software development. Phoenix [3] 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 [37]. Nerves [4] 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.

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.

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. 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 [37].Another potential drawback of Elixir is that it is a dynamically-typed programming

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

language, which can result in errors from type mismatches or programming mistakes [44]. 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. [44].

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.

3.3 Scala Programming Language with Akka Toolkit Analysis

3.4 Go Programming Language Analysis

3.5 Benchmarking Strategies

3.6 Conclusions

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.

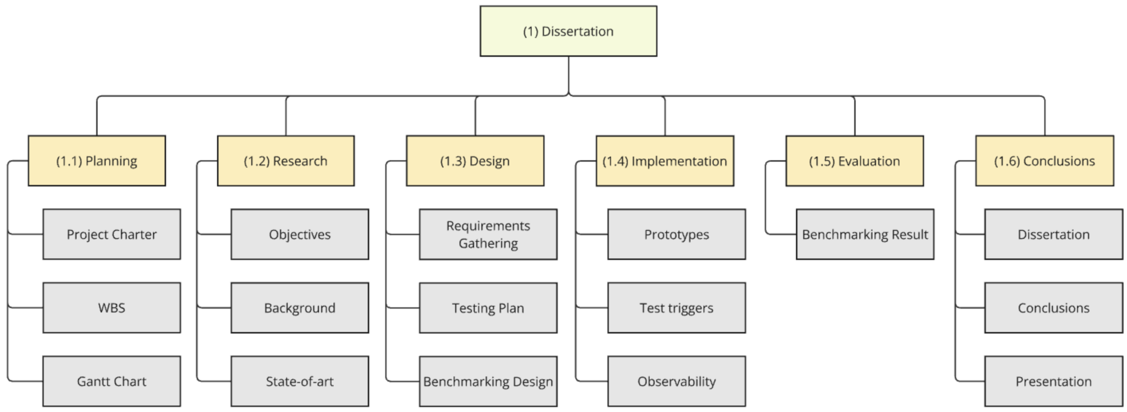

Figure 4.1: The WBS of the project.

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.

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

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.
(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.

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

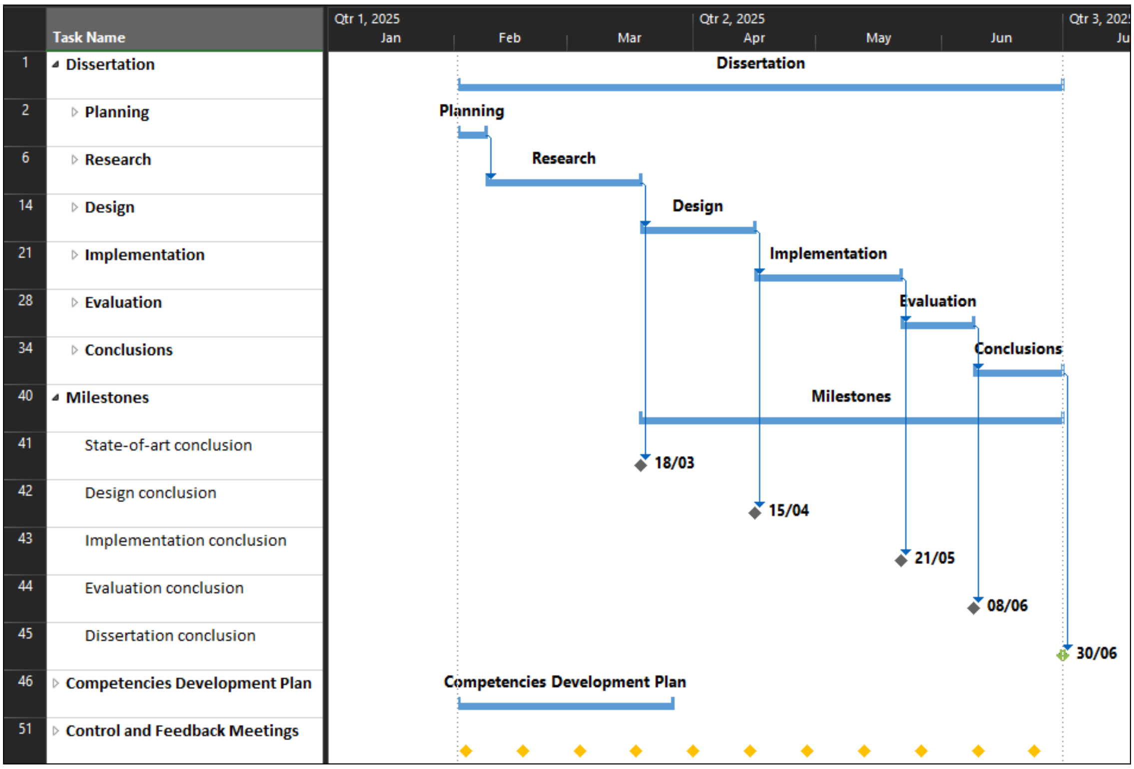

Figure 4.2: Gantt timeline of the project

Item Name	Type of Item	Additional Description / Acceptance Criteria
(1.6.3) Presentation	Deliverable	Prepare and deliver the final presentation to the evaluation committee. **Acceptance Criteria:** Presentation must be clear and precise.

Table 4.1: WBS dictionary

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 Gantt Details 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"*.

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.

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, 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.

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.

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].

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.

[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)

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

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.

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 Gantt Details

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.

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] 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.

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

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

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

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

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

[7] 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.

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

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

[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*. url: https://hexdocs.pm/elixir/introduction.html.

[42] *Elixir School - Official*. Accessed at 03.11.2024. 2023. url: https://elixirschool.com/.

[43] 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.

[44] 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.