INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Análise Comparativa da Tolerância a Falhas em Elixir e Outras Linguagens Distribuídas e Concorrentes

NUNO DINIS GONÇALVES RIBEIRO
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

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

Nuno Ribeiro

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

Advisor: Dr. Luís Nogueira

Porto, June 29, 2025

Statement of Integrity

I hereby declare that I have 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. The exceptions are explicitly recognised in the section "Ethical considerations" of the first chapter. This section also states how AI tools were used and for what purpose.

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

ISEP, Porto, June 29, 2025

Abstract

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

Nevertheless, other distributed and concurrent languages offer solutions, often leveraging similar concepts in different environments. For instance, Akka, inspired by "let it crash," provides a toolkit built for Scala. Similarly, Go, celebrated for its built-in concurrency primitives like goroutines and channels, can integrate with the Actor Model through the Proto.Actor framework.

Although all three languages are capable of implementing the Actor Model, Elixir adopts it as a first-class paradigm through its native integration with the Bogdan/Björn's Erlang Abstract Machine (BEAM) Virtual Machine and the Open Telecom Platform (OTP) toolkit. In contrast, Akka leverages the maturity and optimizations of the Java Virtual Machine (JVM), offering a powerful abstraction over traditional concurrency models. Proto.Actor, on the other hand, introduces a language-agnostic approach to the Actor Model, built upon Protocol Buffers, which promotes interoperability.

These modern tools bring significant fault-tolerant capabilities to software development. However, it is essential to understand how each compares, necessitating the execution of benchmarking. Thus, based on existing literature, the most effective methodology for benchmarking involves designing a generic application that simulates distributed and fault-tolerant operations. This approach led to a proposed test framework with a distributed, configurable architecture and test scenarios designed to yield measurable results.

Empirical benchmarks revealed Elixir consistently delivers high throughput and low variance under moderate to low fault rates, though reconnection latency increases at scale due to centralized coordination bottlenecks. Scala with Akka demonstrated the most stable fault recovery and reconnection latency across all scales, well-suited for large systems with rigorous stability. Go with Proto.Actor offered competitive throughput in small-scale deployments but exhibited scalability limitations in fault detection and recovery.

These findings show no single runtime dominate and each provides distinct trade-offs between throughput, fault recovery and detection. The results highlight how architecture-level decisions, such communication models, significantly impact runtime behavior under fault-prone conditions. Future work includes enhancing multi-node support, simulating more complex fault types, and exploring different actor system designs.

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

Resumo

A tolerância a falhas é um componente crítico nos sistemas distribuídos, considerando cada vez mais o crescimento exponencial da internet. As capacidades de resiliência e tolerância a falhas são essenciais para manter a fiabilidade e minimizar o *downtime*. Elixir, com as suas funcionalidades de tolerância a falhas e a sua base na filosofia "*let it crash*" de Erlang, destaca-se como uma ferramenta robusta para a construção de tais sistemas.

No entanto, outras linguagens distribuídas e concorrentes oferecem soluções, muitas vezes baseadas em conceitos semelhantes em ambientes diferentes. Por exemplo, Akka, inspirado em "*let it crash*", fornece um *toolkit* construído para Scala. De forma similar, Go, celebrado pelas suas primitivas de concorrência incorporadas, como *goroutines* e *channels*, pode integrar-se com o *Actor Model* através da *framework* Proto.Actor.

Embora as três linguagens sejam capazes de implementar o *Actor Model*, Elixir adota-o como um paradigma de primeira classe através da sua integração nativa com a máquina virtual Bogdan/Björn's Erlang Abstract Machine (BEAM) e o *toolkit Open Telecom Platform* (OTP). Em contraste, Akka faz uso da maturidade e das otimizações da *Java Virtual Machine* (JVM), oferecendo uma abstração poderosa sobre os modelos de concorrência tradicionais. Proto.Actor, por outro lado, introduz uma abordagem agnóstica sobre a linguagem de programção ao *Actor Model*, constituida por *Protocol Buffers*, o que promove a interoperabilidade.

Estas ferramentas modernas trazem capacidades significativas de tolerância a falhas para o desenvolvimento de *software*. Contudo, é essencial compreender como cada uma se compara, o que exige uma análise comparativa exaustiva através de *benchmarking*. Assim, com base na literatura existente, a metodologia mais eficaz envolve a conceção de uma aplicação genérica que simule operações distribuídas e tolerantes a falhas. Esta abordagem levou a uma proposta de uma *framework* de teste com uma arquitetura distribuída e configurável, e cenários de teste desenhados para produzir resultados mensuráveis.

Os *benchmarks* empíricos revelaram que Elixir é capaz de produzir alto nível de *throughput* com baixa variablidade sob qualquer tipo de injeção de erro, embora a latência de reconexão aumente em escala devido a *bottlenecks* de coordenação centralizada. Scala com Akka demonstrou uma recuperação de falhas e latência de reconexão mais estáveis em todas as escalas, sendo adequado para grandes sistemas com requisitos rigorosos de estabilidade. Go com Proto.Actor ofereceu *throughput* competitivo em pequena escala, mas exibiu limitações de escalabilidade na deteção e recuperação de falhas devido à gestão da comunicação remota.

Estas descobertas mostram que nenhuma solução domina isoladamente e que cada uma oferece compensações distintas entre *throughput*, recuperação e deteção de falhas. Os resultados destacam como as decisões ao nível da arquitetura, tais como os modelos de comunicação, impactam significativamente o comportamento em condições de falha. O trabalho futuro inclui a melhoria do suporte *multi-node*, a simulação de tipos de falha mais complexos e a exploração de projetos de sistemas de atores diferentes.

Agradecimentos

Ao concluir esta etapa, sinto o dever de expressar o meu sincero agradecimento a todos os que, de alguma forma, contribuíram para tornar este percurso mais agradável e leve.

Gostaria de manifestar a minha profunda gratidão a todos os familiares e amigos que, direta ou indiretamente, me apoiaram e ajudaram ao longo deste caminho. O vosso incentivo foi verdadeiramente fundamental, e por isso, o meu muito obrigado a todos.

Um agradecimento muito especial à minha Maria, pelo incondicional apoio, carinho, amor e riso. A tua ajuda foi, sem dúvida, um pilar essencial para a concretização deste trabalho.

Ao Professor Doutor Luís Nogueira, o meu muito obrigado pela boa orientação e profissionalismo. A disponibilidade prestada e os contributos foram cruciais durante todo este percurso.

Por fim, e acima de tudo, agradeço a Deus pela saúde e por todas as coisas.

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence.
API	Application Programming Interface.
BEAM	Bogdan/Björn's Erlang Abstract Machine.
CPU	Central Processing Unit.
CSP	Communicating Sequential Processes.
gRPC	Google Remote Procedure Call.
HTTP	Hypertext Transfer Protocol.
IoT	Internet of Things.
IPC	Interprocess Communication.
JVM	Java Virtual Machine.
LOC	Lines of code.
MTTF	Mean Time To Fail.
MTTR	Mean Time To Repair.
OS	Operating System.
OTP	Open Telecom Platform.
TCP	Transmission Control Protocol.
URL	Uniform Resource Locator.
VM	Virtual Machine.
WBS	Work Breakdown Structure.

Chapter 1

Introduction

1.1 Context and Problem

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

Elixir, a functional programming language built on the Erlang Virtual Machine (VM) called Bogdan/Björn's Erlang Abstract Machine (BEAM), has gained significant popularity for its "let it crash" paradigm, a philosophy that emphasizes process isolation, supervision trees, and fault recovery [3–5]. This design approach, which originates from Erlang's legacy in telecom systems, positions Elixir as a strong candidate for developing fault-tolerant systems. Concurrently, other distributed and concurrent programming languages also implement or adapt concepts similar to the "let it crash" philosophy through their own Actor Model implementations, making use of their unique concurrency and language primitives. This allows for a valuable comparison of how different language ecosystems implement and leverage these fault tolerance strategies.

Despite the recognized importance of fault tolerance in software systems, there is a lack of comprehensive, up-to-date research directly comparing the high-level fault tolerance and resilience aspects of Elixir with other prominent programming languages. Specifically, detailed empirical analyses, such as those concerning performance and resilience, are often absent. This gap in comparative analysis difficult software developers and architects from making informed decisions when selecting a language or framework for building fault-tolerant applications and from determining the most suitable language for a specific architectural design.

1.2 Objectives

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

- Comprehensively analyze the fault-tolerant mechanisms in Elixir, including its design paradigms, implementation strategies, and practical applications.
- Identify the most popular and relevant distributed and concurrent programming languages for comparison and investigate their fault-tolerant mechanisms.
- Compare Elixir's fault-tolerant capabilities with those of other languages to elucidate their respective strengths, weaknesses, and trade-offs.
- Conduct benchmarking experiments to empirically evaluate and compare the fault tolerance and resilience of Elixir against other distributed and concurrent programming languages, providing quantitative data to support the analysis.

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, encompassing data privacy, informed consent, and responsible disclosure of findings.

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

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 [7] and the Association for Computing Machinery Code (ACM) of Ethics and Professional Conduct [8], 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. Complementing, it is important to note that the theoretical foundation for this dissertation was utilized on the "Dissertation's Preparation" subject. However, all its content was reviewed and updated to reflect current insights and findings.

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.

Usage of Artificial Intelligence Tools In the execution of this dissertation, Artificial Intelligence (AI) tools were not utilized for content generation or the formulation of arguments. Their application was limited to assisting in some improvements in the linguistic quality of the text and grammatical correction.

1.4 Document Structure

The document is organized as follows:

- Introduction: This chapter provides an initial overview of the dissertation's scope, encompassing the context, the problem statement, and the primary goals the study aims to achieve. Additionally, it details the ethical considerations undertaken.
- Fundamentals of Distributed Systems and Fault-Tolerant Design: This chapter presents the foundational knowledge supporting the dissertation's background context. It is structured into three parts. The first part discusses general aspects of distributed systems, emphasizing their characteristics and the theoretical foundations. The second explores fault tolerance strategies and underlying principles. The third reviews distributed and concurrent programming languages, thereby justifying the selection of those analyzed in this work.
- Analysis of Fault-Tolerant Programming and Benchmarking: This chapter investigates the fault tolerance mechanisms of Elixir, Scala with Akka, and Go, as highlighted in the previous chapter. For each language, the main characteristics are detailed, along with an explanation of how each enables distributed and fault-tolerant features. The chapter concludes with an overview of benchmarking methodologies found in the literature, which will support the upcoming experimental work, by analyzing strategies and metrics present in the existing research.
- Architecture and Test Design: Introduces a high-level architectural proposal inspired by actor-based systems, along with the motivations and challenges it addresses. It then details the concrete architecture used in this study, describes the test scenario design, outlines the language-specific implementations, and highlights observed limitations.
- Experiments Results: This chapter presents the empirical results obtained from each test scenario, focusing on the throughput, mean reconnection time, and detection time across all targeted languages. All test case scenarios are accompanied by graphs, a detailed analysis, and summarized conclusions.
- Conclusion and Future Work: This chapter concludes the dissertation with a synthesis of the overall findings, relating them to the initially stated goals. It also presents potential directions for future work.

Chapter 2

Fundamentals of Distributed Systems and Fault-Tolerant Design

2.1 Distributed Systems

In the early days of computing, computers were large and expensive, operating as standalone machines without the ability to communicate with each other. As technology advanced, smaller and more affordable computers, such as smartphones and other devices, were developed, along with high-speed networking that allowed connectivity across a network [2]. These innovations made it possible to create systems distributed across nodes where tasks could be processed collectively to achieve a common goal [2]. Nodes in a distributed system may refer to physical devices or software processes [9].

To the end-user, distributed systems appear as a single, large virtual system, making the underlying logic transparent [9]. These systems achieve a shared objective by transmitting messages through various nodes and dividing computational tasks among them, increasing resilience and isolating business logic [9, 10]. Distributed systems can present heterogeneity, such as differing clocks, memory, programming languages, operating systems, or geographical locations, all of which must be abstracted from the end-user [2, 10].

2.1.1 Characteristics

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

Transparency

Transparency in distributed systems enables seamless user interaction by hiding the complexity of underlying operations [2, 11]. 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 the Uniform Resource Locators (URLs) [2, 12]. Replication transparency ensures reliability by masking data duplication, while failure transparency enables systems to handle faults without user disruption [2, 12]. 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 [13]. A highly reliable system maintains consistent, uninterrupted service over an extended period, minimizing disruptions for users [2]. On other hand, availability measures the probability that the system is operational and ready to respond correctly at any given moment, often expressed as a percentage of system up-time [2, 14].

Scalability

Designing and building a distributed system is complex, but also enables the creation of highly scalable systems, capable of expanding to meet increasing demands [2, 5, 9]. 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 [15]. Cloud services must support a large volume of simultaneous connections and interactions, making scalability a crucial factor [2].

Fault tolerance

Fault tolerance is a critical characteristic of distributed systems, closely linked to reliability, availability, and scalability. For a system to maintain these properties, it must be able to mask failures and continue operating despite the presence of errors [2]. Fault tolerance is especially vital in distributed environments where system failures can lead to significant disruptions and economic losses across sectors such as finance, telecommunications, and transportation [10].

The primary goal of a fault-tolerant system is to enable continuous operation by employing specific strategies and design patterns to mask the possible errors [1].

2.1.2 Communication

Communication is fundamental in distributed systems for coordination and data exchange. Nodes communicate over networks or via Interprocess Communication (IPC) when on the same machine [9]. Synchronous communication involves blocking operations where the sender waits for a response, suitable for scenarios requiring confirmation [2, 12]. In contrast, asynchronous communication allows non-blocking operations, enabling the sender to proceed without waiting. This approach, often supported by message queues, is a good suit for decoupled and heterogeneous systems [2].

2.1.3 Challenges

Distributed systems encounter numerous challenges, including scalability [13], managing software, network, and disk failures [16, 17], heterogeneity [12], coordination among nodes, and difficulties on debugging and testing [9, 17]. For the scope of this dissertation only the CAP theorem will be discussed.

CAP Theorem. The CAP theorem says that in a system where nodes are networked and share data, it is impossible to simultaneously achieve all three properties of Consistency, Availability, and Partition Tolerance [2, 9]. This theorem underlines a critical trade-off in

distributed systems: only two of these properties can be fully ensured at any given time [18]. A description of the properties can be given by:

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

According to the CAP theorem, when a network partition occurs, a distributed system must prioritize either consistency or availability, as achieving all three properties is not feasible in practice [2, 9, 18]. This concept is relevant to this dissertation, as fault tolerance strategies described 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 [10]. Fault tolerance can address a range of issues, including networking, hardware, software, and other dimensions, with various strategies designed to manage these different fault types [2, 19].

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 and Srirama [20]. 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 [20, 21].

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 not consistent and potentially misleading [2].

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.

Table 2.1: Brief description of fauls types

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 [20].
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 [20].
Permanent Faults	Persistent issues that continue until the root cause is identified and addressed. Relatively straightforward to fix, typically related to complete component malfunction [2].
Byzantine Faults	Caused by internal system state corruption or incorrect network routing. Handling is complex and costly, often requiring multiple component replicas and voting mechanisms [20].

Table 2.2: Brief description of failure types

Type of Failure	**Description**
Crash Failure	The system halts and stops all operations entirely. Although it was functioning correctly before the halt, it does not resume operations or provide responses after the failure [2].
Omission Failure	The system fails to send or receive necessary messages, impacting communication and task coordination [20].
Timing Failure	The system's response occurs outside a specified time interval, either too early or too late, causing issues in time-sensitive operations [20].
Response Failure	The system provides incorrect outputs or deviates from expected state transitions, potentially leading to wrong results [2].
Arbitrary Failure	The system produces random or unpredictable responses at arbitrary times, potentially with incorrect data. This type of failure is challenging to diagnose and manage [2].

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 [11]. Despite its simplicity, this strategy is highly suitable in many scenarios, particularly when implementing more complex fault tolerance mechanisms would introduce unnecessary cost in environments with a high likelihood of transient faults. However, it is crucial to recognize that retrying in the case of a permanent error is pointless. Moreover, if the failure is caused by system overload, uncontrolled retries can aggravate the issue. To address these challenges, implementing a maximum retry limit and incorporating strategies such as exponential backoff, where retries are spaced out with increasing delays, becomes essential [1, 9].

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 [20]. 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 insignificant 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 [20]. In case of an unresolved situation it is created an omission failure due to the missing communication. Additionally, it is well-suited for database systems to handle transient locking or deadlock conditions, as well as in microservice architectures, where downstream services may temporarily become unresponsive but recover shortly thereafter [1].

This strategy aligns effectively with Actor Models due to their inherent monitoring capabilities, which detect errors and initiate retries automatically. Frameworks such as Akka have built-in support for this mechanism [20].

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 [9]. By actively monitoring the health of operations and selectively blocking problematic ones, circuit breakers act as safeguards against system overload and degradation [22].

Circuit breakers operate in three primary states: Closed, Open, and Half-Open [9, 22]. 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 [9, 22]. 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 [22].

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 [22]. 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 [9]. 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 [9]. In contrast, circuit breakers proactively block failing operations, reducing the risk of cascading failures and preserving overall system stability.

[1] From Oscar Blancarte Blog. https://www.oscarblancarteblog.com/2018/12/04/circuit-breaker-pattern/ (accessed 29 June 2025).

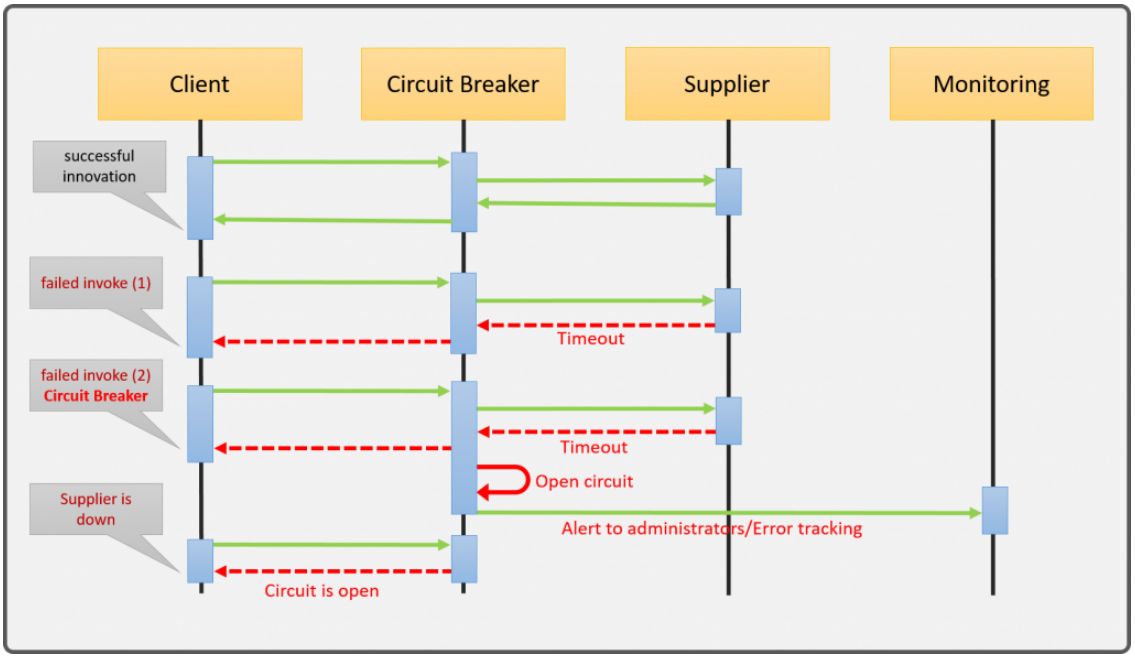

Figure 2.1: Diagram illustrating the states of a circuit breaker [1].

Replication and Redundancy

Replication is a fundamental strategy for achieving fault tolerance in distributed systems and is widely used across various domains [10]. By creating multiple replicas of data or processes, replication eliminates single points of failure, ensuring system reliability, availability, and transparency [12]. This approach allows a system to tolerate faults by introducing redundancy, which distributes operations across a group of replicas rather than relying on a single vulnerable node [2].

To effectively coordinate replicas and maintain consistency, replication mechanisms employ various strategies, which can be categorized as follows [20]:

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

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

Consensus Algorithms

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

Paxos

Paxos appeared in 1989 and has evolved over time, earning a reputation as a complex and difficult to understand algorithm [2]. Due to its challenges and the emergence of newer alternatives like Raft, which is described below, this explanation of Paxos will concentrate on its core concepts without going into exhaustive detail.

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, out of order, or lost, and no assumptions are made about message delivery timing [2, 23]. The algorithm revolves around three distinct roles [12, 23]:

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

With the roles defined, the Paxos algorithm progresses through the following phases to achieve consensus [2, 12, 23]:

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

While Paxos is robust in theory and guarantees consistency, its complexity and subtle behaviors make it difficult to implement correctly or faithfully to its original design [2]. Over time, new variations and extensions, such as Multi-Paxos, have been developed. Multi-Paxos enables the system to achieve consensus on multiple values, making it more practical for real-world applications, like Chubby lock service of Google, but in general it is not considered a highly adopted algorithm [12].

The inherent complexity of Paxos has also driven the creation of simpler consensus algorithms, such as Raft, which aim to provide the same guarantees while being easier to understand and implement [2, 23].

Raft

Raft is a consensus protocol designed to enable fault-tolerant operations in distributed systems. It ensures that a process will eventually detect if another process has failed and take appropriate corrective action. Raft was developed as a more comprehensible and practical alternative to Paxos, addressing its complexity and promoting clarity [2, 24].

Each process in Raft maintains a log of operations, which may include both committed and uncommitted entries. The primary goal of Raft is to ensure that these logs remain consistent across all servers, such that committed operations appear in the same order and position in every log [2]. To achieve this, Raft uses a leader-based approach, where one server assumes the role of leader while the remaining servers act as followers. The leader is responsible for determining the sequence of operations and ensuring their consistent replication [9]. The typical number of nodes used is five [24].

When an operation request is submitted, the leader appends the operation to its log as a tuple where it contains: the operation to be executed, the current term of the leader, and the index of the operation in the leader's logs [2]. The term is reset every time an election occurs, starting from zero [24]. This information is then propagated to the followers using a process inspired by the two-phase commit protocol, where it consists on [2, 9]:

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

This two-step process guarantees that committed entries are replicated on a majority of servers, preserving durability and consistency, even in case of server failures [2]. However, there are cases where the leader fails and an election starts among the followers. The followers acknowledge the leader's failure through the heartbeat strategy, where after a certain time without receiving any signal sent by the leader, the follower starts an election , like represented on Figure 2.2 by the change of state from follower to candidate [9, 24]. To prevent multiple followers from initiating elections simultaneously, heartbeat timeouts are randomized [2, 9].

The change of state of the node is displayed on the Figure 2.2 and the process consists on the following steps [9, 24]:

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.

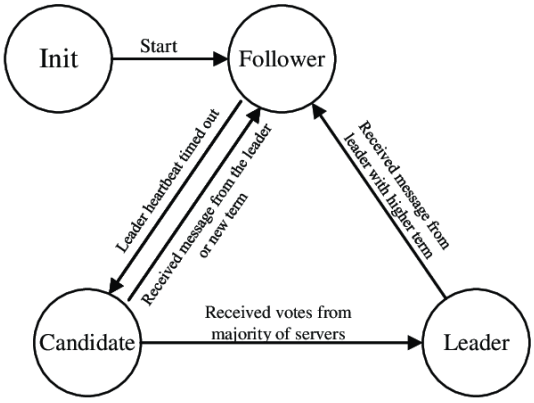

Figure 2.2: Diagram illustrating the states of a node in the Raft algorithm [2].

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

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

Comparison of Paxos and Raft

Both Paxos and Raft are leader-based distributed consensus algorithms designed to ensure consistency in replicated state machines [23]. The key difference lies in their approach to leader election and log replication. Raft prioritizes simplicity and readability, requiring candidates to have up-to-date logs before becoming leaders, avoiding complex log synchronization steps common in Paxos. Paxos, while general and flexible, allows for greater complexity, such as out-of-order log replication and term adjustments during leadership changes [2].

Raft is widely used in distributed systems requiring high availability and consistent state management. Notable applications use adaptations of Raft, including etcd for distributed key-value storage, Consul for service discovery, and CockroachDB for scalable, consistent databases [9, 23]. Paxos, on the other hand, is often found in legacy systems and specialized applications like Google's Chubby service [23].

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 state without error. Recovery strategies are generally classified as either backward recovery or forward recovery [2].

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

[2]Adapted from Jinjie Xu and colleagues. `https://doi.org/10.3390/sym14061122/` (accessed 29 June 2025).

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

Check-pointing is a backward recovery technique. It periodically saves the state of a process, enabling it to restart from the last saved state in the event of a failure. However, this approach is computationally expensive and introduces performance overhead [2, 20]. Furthermore, some operations are inherently irreversible, which limits the effectiveness of this strategy [2, 11].

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 [11, 20]. Although checkpoints are still required to avoid replaying all messages from the beginning, the overhead is significantly reduced compared to traditional check-pointing [20]. For this strategy to work effectively, system operations must be deterministic, ensuring that replaying messages reproduces the correct state, which is also a limitation or even an impossibility in some cases [2].

Apache Flink employs check-pointing to recover state and stream positions, ensuring that the application maintains consistent semantics even in the event of failures, as outlined in the project's documentation [25]. However, these strategies can be resource-intensive, and it is crucial to carefully evaluate their suitability.

Overview of Fault Tolerance 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 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 plays an important aspect in building resilient and fault-tolerant systems [26]. 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 [27],

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 [26]. This has driven the evolution of new paradigms, languages, frameworks, and libraries aimed at reducing development complexity in distributed and concurrent systems [5].

2.3.1 Models and Paradigms

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

Actor Model

The Actor Model, a conceptual framework for concurrent and distributed computing, was introduced by Carl Hewitt in 1973 [28]. 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 [29]. Each actor is equipped with its own mailbox, which receives messages and processes them sequentially [30].

A core principle of the Actor Model is isolation, maintaining their own internal state that is inaccessible and immutable by others [30]. This eliminates the need for shared memory, reducing complexity and potential data races [5].

The Actor Model also introduces the concept of supervision, where actors can monitor the behavior of other actors and take corrective actions in the event of a failure. This supervisory mechanism significantly enhances fault tolerance, enabling systems to recover gracefully from errors without compromising overall reliability [29].

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 [31]. Additionally, the model has been extended to other languages through frameworks and libraries. For instance, Akka brings actor-based concurrency to Java, Scala, C# and F# while Kilim provides similar functionality specifically for Java [29]. 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 [32].

CSP offers an abstract and formal framework for modeling interactions between concurrent processes through channels, which serve as the communication medium between them [33]. 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 [32]. While similar in some respects to the Actor Model, CSP distinguishes itself through its emphasis on direct coupling via channels and synchronization.

The CSP model influenced on programming languages and frameworks. For example, Go integrates CSP concepts in its implementation of goroutines and channels [4, 5, 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, promoting 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 [5, 33]. These approaches often introduce complexity, increasing the probability of deadlocks or race conditions. This has driven the need for languages and frameworks that abstract away these challenges, offering safer and more developer-friendly concurrency models [5].

One widely adopted paradigm for mitigating concurrency issues is the Actor Model. By avoiding shared state and using message passing for communication, the Actor Model reduces risks inherent in traditional concurrency mechanisms such as mutexes and locks [5]. Erlang, for instance, is renowned for its fault tolerance and "let-it-crash" philosophy [26]. Supervising actors monitor and recover from failures, makes Erlang highly suitable for building robust distributed systems [26]. Building on Erlang's foundation, Elixir introduces modern syntax and developer tooling while retaining Erlang's strengths for creating large-scale, fault-tolerant systems. These features make Elixir a popular choice for modern distributed systems development [3].

Haskell, a pure functional programming language, provides a deterministic approach to concurrency, ensuring consistent results regardless of execution order [5]. Its extension Cloud Haskell[3], builds upon the Actor Model, drawing inspiration from Erlang, to allow distributed computation through message passing.

[3] Cloud Haskell: `https://haskell-distributed.github.io/` (accessed 29 June 2025)

Similarly, Akka, a framework built for Scala, adopts the Actor Model to support distributed and concurrent applications. Akka combines Scala's strengths in functional and object-oriented programming, enabling developers to merge these paradigms effectively [5]. Unlike Erlang, Akka operates on the Java Virtual Machine (JVM), providing seamless interoperability with Java-based systems [37].

Go, developed by Google, simplifies concurrent programming through its lightweight goroutines and channels, inspired by the CSP paradigm. This abstraction simplifies the classic 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 [4]. Furthermore, the Proto.Actor[4] framework enables the integration of the Actor Model with CSP principles as the same moment that brings interoperability.

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 [38].

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 2025[5] and Tiobe June 2025[6].

Conversely, some relatively recent languages have gained attention. Unison[7] employs content-addressed programming using hash references to improve code management and distribution. Gleam[8] compiles to Erlang and offers its own type-safe implementation of Open Telecom Platform (OTP), Erlang's toolkit. Pony[9], 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 2025 and Tiobe June 2025 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 June 2025 and IEEE Spectrum August 2024[10], the most recent to moment's date, are included to provide an overview of their popularity and adoption.

Analyses and Language Choice Justification

The focus of this dissertation is on Elixir as the central language for comparison. Elixir is chosen due to its modern syntax, developer-friendly tooling, and robust foundation on the BEAM also know as Erlang VM [3]. Since Elixir inherits all the strengths of Erlang [5], including fault tolerance and the Actor Model, a direct comparison with Erlang is unnecessary as they share the same core runtime and strategies. Such a comparison would likely yield redundant results and add little value to the research.

[4] Proto.Actor: `https://proto.actor/` (accessed 29 June 2025)

[5] RedMonk January 2025: `https://redmonk.com/sogrady/2025/06/18/language-rankings-1-25/` (accessed 29 June 2025)

[6] Tiobe June 2025: `https://www.tiobe.com/tiobe-index//` (accessed 29 June 2025)

[7] Unison: `https://www.unisonweb.org/` (accessed 29 June 2025)

[8] Gleam: `https://gleam.run/` (accessed 29 June 2025)

[9] Pony: `https://www.ponylang.io/` (accessed 29 June 2025)

[10] IEEE Spectrum 2024: `https://spectrum.ieee.org/top-programming-languages-2024/` (accessed 29 June 2025)

Table 2.3: Characteristics of distributed and concurrent programming languages

Name	**Concurrency Strategy**	**Model**	**Tiobe June 2025**	**IEEE Spectrum 2024**
Java	Explicit	Object-Oriented	4	2
Rust	Explicit	Procedural	18	11
C (PThreads)	Explicit	Procedural	2	9
Erlang	Actor Model	Functional	50	48
Elixir	Actor Model	Functional	46	35
Haskell	Evaluation Strategy	Functional	29	38
Scala (Akka)	Actor Model	Functional	31	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

On the other hand, comparing Elixir with low-level languages like Java, Rust, and C would also be less effective. These languages require explicit management of concurrency and fault tolerance [5], introducing complexities that diverge significantly from Elixir's high-level abstractions. A comparison in this context might be not realistic 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 VM differ: the BEAM for Elixir and the JVM for Akka [37]. Additionally, Scala with Akka is notable for its community acceptance [5]. This comparison is valuable because it explores how different implementations of the same paradigm can influence fault tolerance strategies and performance.

Furthermore, too recent or older languages with minimal popularity, such as Emerald, Oz, Unison and Gleam are excluded from this study. These languages lack widespread adoption, and insights derived from them would have limited applicability for the majority of developers, as demonstrated in Table 2.3 with a non-appearance in the Tiobe and IEEE Spectum rankings.

From another perspective, the inclusion of Go in this study adds an interesting dimension to the comparison. Go, unlike Elixir and Akka, lacks native, built-in support for distributed systems. However, its increasing popularity and industry adoption make it a strong candidate for exploration [34]. The combination of CSP and Proto.Actor within Go could offer valuable insights to the community; maintaining a consistent paradigm, the Actor Model, across all implementations contributes to a more comparable benchmarking process.

Chapter 3

Analysis of Fault-Tolerant Programming and Benchmarking

3.1 Elixir Programming Language Analysis

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

3.1.1 The Foundation of Erlang

Elixir is built on top of Erlang, making it essential to first understand Erlang's core principles and environment to move into Elixir's capabilities. Elixir leverages Erlang's foundation for constructing fault-tolerant and distributed systems, benefiting from its mature ecosystem and proven reliability [3, 26].

Erlang, developed in the mids of 1980s by Ericsson, was specifically designed to support systems that are highly reliable, responsive, scalable, and available [3, 26]. 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 [3, 39]. 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 a state-of-the-art concurrent programming model [40].

Concurrency in BEAM

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

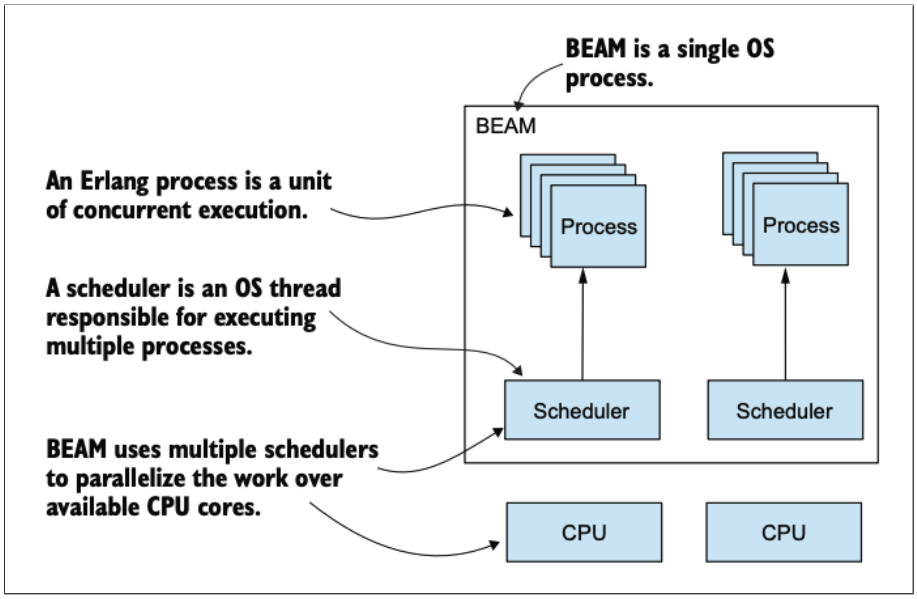

Figure 3.1: Concurrency in the Erlang virtual machine [3].

are heavy. However, virtual threads, introduced on Java 21, brings more capabilities to the JVM allowing a more fined scheduler like BEAM does [40].

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

The BEAM scheduler is considered preemptive, meaning that assigns short execution time slices to each process. This ensures that long-running tasks do not monopolize system resources, promoting fairness and responsiveness [26]. Also, it promotes fault tolerance characteristic by stopping processes carried with permanent faults, where on a non-preemptive scheduler could harm the overall system. Processes that are blocked due to I/O operations or waiting for messages are efficiently managed by separate threads or a kernel polling service, preventing unnecessary CPU usage and ensuring that waiting processes do not stop the execution of others [3, 40].

In a direct comparison of Elixir's processes running on the BEAM with the two threading techniques of the JVM, as illustrated in Figure 3.2, notable differences emerge [40]. The author makes it clear that the benchmarking focuses solely on concurrency, and although the exact type of code is not specified, the emphasis on concurrency is evident. Under low-load conditions, all three strategies, Elixir's processes, the JVM's platform threads, and virtual threads, perform effectively. However, as the number of concurrent units increases, Elixir continues to scale, handling up to approximately 200,000 concurrent processes, while the JVM is constrained by its architecture and struggles to maintain similar scalability. It is also important to note that the author explicitly limits the scope of this comparison to concurrency alone, because in terms of raw speed the BEAM is not necessarily the strongest

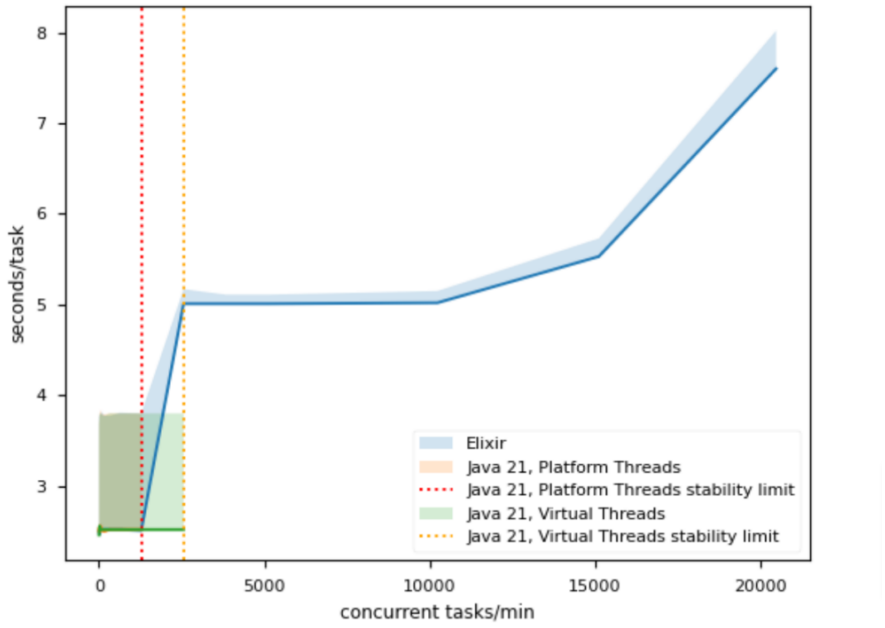

Figure 3.2: Elixir/BEAM processes vs JVM threads [40].

performer, as the author notes that Erjang, the version of Erlang running on the JVM, was observed to be up to 5000% faster than the BEAM in certain cases.

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, the BEAM is designed to manage a large number of lightweight processes efficiently. The JVM threading model is better suited for low-concurrency scenarios involving long-lived threads, or it requires a high-level abstraction over the underlying threads to support high levels of concurrency. In contrast, Elixir/BEAM native processes excel in high-concurrency situations with short-lived processes [5, 40].

Garbage Collection and Immutability

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

Because processes are completely isolated and do not share memory, the 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. Additionally, BEAM optimizes garbage collection by focusing on individual schedulers, further enhancing its efficiency [3, 26].

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 [26].

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].

3.1.2 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, and the optimal approach is not to prevent them entirely but to design systems that can recover autonomously and gracefully [1, 26]. 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 [3].

The Actor Model plays a central role in achieving this resilience. In Elixir, lightweight processes act as independent actors that do not share memory and communicate exclusively through message-passing. When a process encounters an unrecoverable error, it is allowed to fail and terminate. This termination is both deliberate and beneficial, as it enables easy fault detection and ensures that failures do not propagate, preserving the integrity of the overall system [3, 26]. 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 and Proto.Actor also use it. Elixir leverage this pattern to build fault-tolerant systems [5]. In this approach, processes are classified into two types [3]:

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

Supervisors are organized into a hierarchical supervision tree, as illustrated in Figure 3.3. This tree defines the relationships between supervisors and workers, with each supervisor

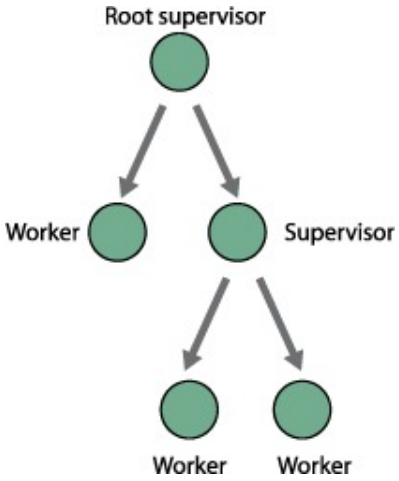

Figure 3.3: Supervisor tree pattern [3].

manage a group of processes. This structure provides modularity and ensures that fault recovery is localized, reducing the impact of failures [26].

Supervisors in Elixir, as well as in the supervisor pattern used in other frameworks, employ restart strategies to manage failures effectively. The options provided by the OTP supervisors, which are among the most commonly used, include the following [3, 26, 39]:

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

Each restart strategy addresses specific use cases. Additionally, supervisors can enforce restrictions on the restart process through a restart frequency configuration. This mechanism monitors the frequency of the worker process failures within a specified time frame. If a worker process fails repeatedly and exceeds the configured threshold, the supervisor itself terminates to avoid harming the system or entering an infinite restart loop [26].

The One-for-One strategy is best suited for independent processes [26]. 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 [26]. 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 [39]. 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 [41]. Additionally, in other paradigms such as Go, there are libraries capable of unifying CSP with the Actor Model, such as the Proto.Actor library [42]. Both are described at the flow of this document.

Tools and Support

Elixir's strengths in the fault tolerance area is related to the integration with the Erlang ecosystem, the BEAM, the Actor Model, and the "let it crash" philosophy. These elements are further enhanced by Elixir's compatibility with the OTP, which provides a suite of design principles and tools for building fault-tolerant and distributed systems. This integration allows Elixir inherit and extend the mechanisms that have been tested and proven in real case scenarios [3, 26].

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 [39].

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 [39, 43].

Beyond the core features of OTP, Elixir also includes Mix, a build tool that simplifies dependency management, testing, project configuration, and documentation generation. Mix integrates into the Elixir ecosystem, simplifying development workflows and contributing to the reliability of applications by ensuring consistent builds [39, 43].

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. [44], and Ra[1], developed by the RabbitMQ team, provide an implementation of the Raft consensus algorithm. Similarly, the Fuse[2] library, a widely-used implementation of the circuit breaker pattern, developed in Erlang, is also compatible with Elixir.

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

Lastly, it's important to mention the Elixir environment, which includes frameworks that enhance software development. Phoenix[3] is a popular framework for building scalable and fault-tolerant web applications. It inherits Elixir's fault tolerance, allowing applications to handle errors gracefully and maintain uptime. Phoenix also supports real-time features through channels for live updates [3]. Nerves[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 while addressing fault tolerance concerns.

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

[1] Ra: `https://github.com/rabbitmq/ra/` (accessed 29 June 2025)
[2] Fuse: `https://github.com/jlouis/fuse/` (accessed 29 June 2025)
[3] Phoenix: `https://phoenixframework.org/` (accessed 29 June 2025)
[4] Nerves: `https://nerves-project.org/` (accessed 29 June 2025)

3.1.3 Drawbacks and Real Applications

The benefits of Elixir are closely tied to the powerful features of the BEAM, as mentioned earlier. However, there are some drawbacks to consider. One major limitation is the lack of third-party libraries, despite OTP providing good support, it is currently challenging for Elixir to compete with more popular languages in this regard, such as Java. Additionally, although the BEAM has a distributed nature, its per-process garbage collection, while efficient for concurrent processes, might make it less suitable for fault-tolerant applications in critical systems that require extremely low-level control and predictable real-time performance.

A potential drawback of Elixir is that it is a dynamically-typed programming language, which can result in errors from type mismatches or programming mistakes [45]. 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. [45]. Despite these efforts, the proposed type system has yet to gain widespread industry adoption.

Another point to consider is that Erlang's foundations lie in the telecommunications industry, where reliability and concurrency capacity were prioritized over raw speed. Consequently, the BEAM might exhibit lower performance in terms of execution speed compared to some other languages [40].

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.

3.2 Scala Programming Language with Akka Toolkit Analysis

The success of the Actor Model, particularly through its implementation in Erlang inspired other programming languages to replicate its concepts [3, 37]. Among these were languages running on the JVM, such as Scala and Java. However, significant differences emerged due to the JVM's inherent concurrency challenges. Unlike BEAM, which was built with lightweight process isolation and message passing at its core, the JVM relied on low-level thread management and shared memory, necessitating careful synchronization through locks and other mechanisms [5, 37]. Although this languages offered APIs for concurrency management, these general-purpose languages placed much of the responsibility on the developer, going against the philosophy of Erlang/Elixir, that it was created for easy concurrency programming [37, 41].

This gap in the JVM ecosystem led to the creation of the Akka toolkit, designed to bring the actor programming model to its environment. Inspired by Erlang, Akka offers a runtime and comprehensive tools to support actor-based programming, enabling developers to leverage the JVM while benefiting from a more structured approach to concurrency [41]. By abstracting thread management and offering a framework for distributed communication and fault tolerance, Akka provides a robust solution for building scalable and distributed systems. This capability is comparable to what Elixir offers [37].

3.2.1 How Akka Handles the Actor Model

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

Location Transparency and Communication

The Actor Model, like described before, defines a concurrent paradigm where actors operate independently, maintaining their own mailboxes, and communicate exclusively via message passing. This communication should ideally respect the location transparent characteristic of distributed systems, allowing actors to interact easily regardless of their physical location [26]. On the JVM, threading enables scaling within a single machine by utilizing additional CPUs and memory due to the shared heap memory and concurrency model [37]. However, native support for scaling across distributed systems is lacking, a contrast to BEAM's distributed aspect.

Elixir facilitates communication natively, with the Erlang distribution protocol [43]. Akka addresses this limitation through its latest remoting protocol, Artery, which builds upon the older remoting mechanisms making improvements [37, 41]. Artery is designed for high-throughput and low-latency communication, utilizing either Transmission Control Protocol (TCP) or Aeron (UDP) for message transfer. This re-implementation of the old remoting module enhances performance and stability while remaining mostly source compatible. Artery also provides a separate subchannel for large messages and compresses actor paths to minimize overhead [41].

Akka facilitates scalability and communication with the discovery module. The discovery module allows actors to be registered with a specific name. In this context, actors can be registered using a designated key, and other actor can communicate with that name without knowing the specific address [37]. Furthermore, communication with these actors occurs through their mailboxes, following a standard First In First Out (FIFO) protocol, equals to the behavior in Elixir [3, 46].

Actors Isolation

Actor isolation is a foundational principle of the Actor Model, where actors are designed to operate independently and avoid shared state [26]. In Java, this independence can be implemented using low-level concurrency mechanisms, while in Scala it is often achieved through immutability. However, Akka significantly simplifies the process by providing an Actor API that inherently enforces isolation [37, 47]. Initially, Akka introduced the Classic Actors API, which supported untyped actor logic. In this model, messages were transmitted without type safety and processed using pattern matching, similar to the approach employed in Elixir [41]. In contrast, the modern standard is the Typed Actor API, which provides a more robust and type-safe solution. The Typed Actor API enforces type safety through, ensuring that only messages of the defined type can be sent to an actor [37, 41]. This differs from Elixir, where actor communication is dynamically typed and does not provide type guarantees.

Unlike Elixir, Scala permits mutable programming, which introduces the potential for shared mutable data to be passed between actors, a practice strongly discouraged in Akka's documentation. Despite these guidelines, the use of mutable data within Akka is technically possible. If misused, this could reintroduce well-known concurrency issues in the JVM, such as race conditions and thread interference [41]. This limitation stands in contrast to Elixir's approach, where the VM guarantees strict process isolation, effectively eliminating such risks [3, 5].

To achieve efficient performance, actors in Akka often share underlying threads, as individual threads are resource-intensive [46]. Akka actors mimic the lightweight processes of the BEAM by managing multiple actors within a single thread. This approach significantly reduces memory consumption compared to the heavyweight JVM threads. For instance, approximately 2.7 million Akka actors can fit within 1 GB of memory, a considerable contrast to the 4,096 threads that would occupy the same space [37]. This makes the minimum size of an actor in Akka, on average, approximately 400 bytes.

Randtoul and Trinder [31] examined the effectiveness of actor isolation in Erlang and Scala with Akka. Their findings revealed that server throughput is affected by the termination of server actors. Specifically, they observed that the throughput for both Erlang and Scala with Akka decreases only in proportion to the percentage of processes that fail. This leads to the conclusion that both Erlang and Scala with Akka offer robust process isolation.

3.2.2 Fault Tolerance Mechanism and Strategies

Akka, built upon the Actor Model, inherits its fault tolerance philosophy from Erlang's design principles, as described in Elixir's section. This forms the foundation of its fault tolerance aspects. At its core, the Actor Model facilitates fault tolerance through the supervisor pattern by defining strategies for handling failures, such as restarting, stopping, or resuming child actors, supervisors ensure that errors are contained and localized, preventing system-wide failures [3, 37, 41]. Like it was previously detailed.

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

Akka Clustering

Akka Cluster is a module that enables peer-to-peer communication among a group of nodes, allowing them to function as a unified distributed system [41, 48]. Designed to enhance fault tolerance, it implements replication and redundancy strategies while achieving location transparency through the Artery protocol, enabling nodes to communicate without knowing the physical locations of others [37].

The module provides an API for managing cluster operations, including managing nodes, designating a leader node, and obtaining cluster information. Although, Akka Cluster is autonomous, capable of redistributing workloads and managing actor states without manual intervention via API [37]. To ensure consistency and reliability, Akka Cluster employs a gossip protocol, a decentralized communication mechanism that facilitates the propagation of information across nodes [2, 41]. This protocol enables nodes to maintain a shared, consistent view of the cluster state [2].

For job processing, Akka Cluster employs a divide-and-conquer architecture through the use of master and worker actors. The master actor decomposes large tasks into smaller subtasks, distributing these among worker actors for parallel processing. Once processing is complete, the workers return their results to the master actor, which aggregates them to produce the final output. This design enhances performance and fault tolerance, as the master actor can reassign tasks from failed workers to other available nodes [41].

Leader election is a important aspect of a cluster management, and it is efficiently managed within Akka Cluster. In contrast to the Raft consensus, algorithm Akka Cluster adopts a simpler method by automatically assigning the leadership role to the node with the lowest address. This approach minimizes the overhead associated with the election process [24, 41].

Furthermore, Akka Cluster supports advanced features such as cluster sharding, which distributes actors across the cluster while preserving their identity and state. This facilitates load balancing and enhances performance by ensuring that workloads are distributed evenly. The module also includes monitoring and health management capabilities, enabling to verify the performance and status of nodes and actors [37].

Akka Circuit Breaker

Akka provides a dedicated module for implementing the circuit breaker pattern, a technique for improve system stability by isolating faults and preventing cascading failures. As presented earlier, the circuit breaker pattern temporarily suspends operations to a failing component, allowing it time to recover while safeguarding the overall system [22].

Akka's circuit breaker module offers a straightforward integration, making it possible to configure the way the circuit it will activate, where it can be defined failure thresholds, timeouts, and recovery intervals. This module is effective in monitoring interactions between actors and external services, ensuring that errors are contained and managed without disrupting the broader system [41].

Akka Persistence and Event Sourcing

Akka offers support for distributed persistence, similar to the functionality provided by Mnesia in Elixir and Erlang [39, 41]. It also features event sourcing, which contributes to fault tolerance through mechanisms such as message logging and check-pointing. Akka Persistence allows actors to recover their state after a failure. This is achieved by keeping an event log that records all changes to an actor's state in the order they occur. Upon restart, typically initiated by a supervisor, the actor replays the logged events to reconstruct its previous state, allowing it to resume operations from the point prior to the failure [41].

To optimize the recovery process, Akka Persistence also supports snapshots, supporting also the check-pointing strategy. Instead of replaying all events from the beginning of the event log, the actor can restore its state from the most recent snapshot and then replay only the events that occurred after that snapshot [37, 41].

3.2.3 Comparison with Elixir/BEAM and Real Applications

The Akka toolkit marks a significant step forward in extending the JVM to support modern concurrency models. As Valkov, Chechina, and Trinder [5] highlight, Akka improves Scala's performance by reducing communication latency. However, since Akka functions as an

abstraction layer on top of the JVM, the same study by observed that Erlang exhibits lower communication latency compared to Scala with Akka. This difference is likely attributed to the BEAM concurrency model [40].

However, Randtoul and Trinder [31] studied how Erlang and Scala with Akka manage server actors failures using a supervisor control pattern. They tested two supervisor-to-actor ratios (1:1020 and 1020:1) to see how throughput is impacted by different failure types. Their findings showed that both systems had similar throughput reductions during burst and random failures, especially with the 1:1 ratio. However, Akka outperformed Erlang in uniform failure scenarios with the 1:1 ratio. Despite of not being a directly Elixir comparison, the underline it is the same making it a valid comparison and showing the potential of Akka.

Garbage Collection A notable difference between Akka and the BEAM lies in their approach to garbage collection. Akka relies on the JVM's garbage collection strategy, which can introduce latency during "stop-the-world" events [37, 41]. These pauses can negatively affect the performance of highly concurrent systems, especially under heavy load. However, advancements in garbage collection technology, such as the Z Garbage Collector, have shown possibilities in reducing pause times significantly. As noted by Chaudhary et al. [49], Z Garbage Collector represents a state-of-the-art approach that is well-suited for applications requiring minimal pauses due to garbage collection.

Scheduling Model Scheduling represents a fundamental difference between Akka and the BEAM. The BEAM employs a preemptive scheduling model designed to efficiently handle numerous small, short-lived tasks, such as high-frequency message handling in highly concurrent systems. This approach ensures equitable CPU time distribution and mitigates process starvation [3, 39, 40]. However, frequent context switching can introduce overhead for long-running processes, potentially reducing efficiency in such scenarios. Akka, on the other hand, relies on the JVM's cooperative scheduling model and enhances it with its Message Dispatcher, which supports configurable thread pools like Fork-Join, that leverages a work-stealing algorithm, and Fixed Thread Pools, optimizing resource usage. This mechanism enables Akka to efficiently manage long-lived and computationally intensive tasks [37, 41].

Built-in Libraries and Support Both the Akka and Elixir/BEAM ecosystems offer comprehensive libraries to address common design patterns for concurrency and fault tolerance. Elixir's OTP framework, like stated before, provides a robust suite of built-in patterns, such as supervisors, specifically tailored for managing concurrency and ensuring system reliability [39, 40]. Akka, on the other hand, adopts a modular architecture with an easily pluggable library of features, including implementations for replication and circuit breakers. In the Elixir ecosystem, equivalent functionality is often achieved through third-party libraries maintained by the community. While these community-driven libraries are highly effective and widely used, their reliance on external maintenance and updates can present a potential drawback compared to Akka's more integrated and officially supported approach.

Real-World Applications Akka, just as Elixir, has demonstrated its capabilities in considerable large-scale applications, emphasizing its scalability, reliability, and performance. For instance, PayPal leverages Akka actors to manage over a billion financial transactions daily, ensuring high availability and robust fault tolerance [47]. Similarly, the Spark big data ecosystem depends on Akka for efficiently shuffling hundreds of terabytes of data across distributed

nodes. Other prominent companies, including Twitter, LinkedIn, and Walmart use Akka to solve concurrency and distributed system challenges [41, 47].

3.3 Go Programming Language Analysis

The Go programming language was designed to facilitate rapid software development while ensuring high execution speed [50, 51]. It addresses the drawbacks of traditional low-level languages like C, which, while performant, can be complex for modern development. At the same time, Go offers a solution to the performance limitations of scripting languages such as Python, which prioritize ease of use but often fails in execution speed [50]. As a statically typed, compiled language, Go enforces type safety, setting it apart from dynamic and immutable languages like Elixir [4].

While Go is not explicitly classified as a distributed or fault-tolerant programming language, it has gained considerable popularity in areas such as microservices, cloud applications, and high-concurrency systems. This popularity can be attributed to its simplicity, speed, and robust concurrency model [52, 53]. In contrast to Elixir, which is designed with immutability and a "let it crash" fault tolerance philosophy, Go does not inherently prioritize fault tolerance or the "let it crash" approach [51]. Nevertheless, Go can be a good candidate for integration into distributed architectures when paired with complementary technologies and libraries. By take advantage of Go's concurrency capabilities, it is possible to replicate the fault-tolerant strategies typical of Elixir-based systems, making it a valuable language to explore in this context.

3.3.1 Concurrency and Distribution

Go adopts a concurrency model rooted in the CSP paradigm, distinguishing itself from other approaches such as the Actor Model. While both paradigms prioritize concurrent communication, they are slightly different in their focus [51]. In CSP, channels are treated as first-class entities, emphasizing the communication mechanism itself, whereas the Actor Model considers processes to be first-class, focusing on the entities performing the computation, as described earlier. Additionally, the Actor Model enforces strict isolation between processes, with no shared memory, while CSP organizes concurrent processes to interact explicitly through channels rather than directly access to a single shared memory object [51].

Go is one of the first programming languages of success to integrate CSP directly into its design, emphasizing data sharing through channels rather than passing references to shared memory among its lightweight threads, known as goroutines [51]. This design choice minimizes potential synchronization complexities and aligns with Go's guiding principle: *"Do not communicate by sharing memory; instead, share memory by communicating"* [4]. This philosophy contrasts with shared-memory concurrency models, where processes directly access and modify shared state. In Go, by design, only one goroutine can access a value at any given time, effectively eliminating the possibility of data races [4, 50]. Nevertheless, Go also provides manual synchronization mechanisms, such as explicit mutexes, for situations where they are necessary [51].

In contrast to the Actor Model, where actors encapsulate state and communicate through asynchronous messages, Go's CSP-based model prioritizes structured communication patterns via channels. This distinction encourages developers to design systems by focusing on

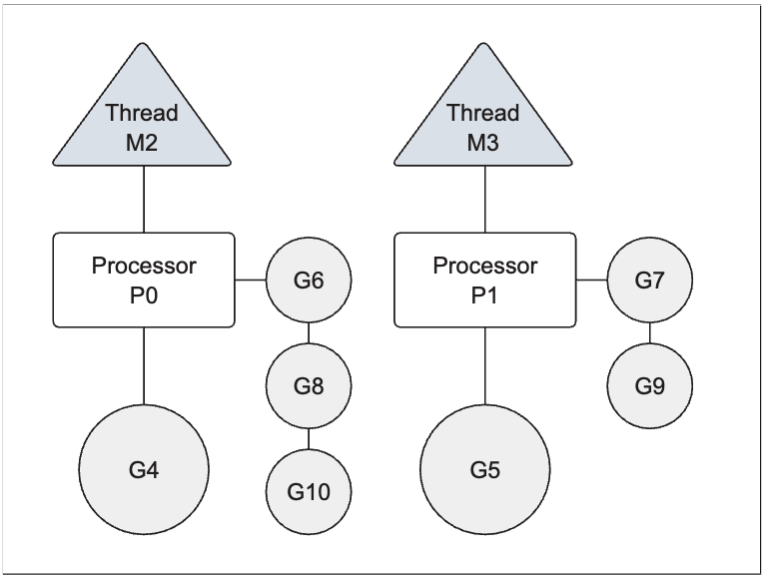

Figure 3.4: Go's scheduler logic of distributing goroutine by the logical processors. Adapted from [50].

the flow of data and the relationships between processes, rather than the individual behavior of the computational units [4].

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

Goroutines

Due to the overhead associated with OS threads, Go enhances efficiency by implementing a multiplexing logic that allows multiple processes to run on the same OS thread, similar to the approaches used in Elixir and Akka [51, 52]. Go achieves this through goroutines, which are lightweight abstractions that enable concurrent and parallel execution. By utilizing goroutines, Go can efficiently manage numerous tasks without significant resource consumption [4].

The OS is responsible for scheduling threads to run on physical processors, whereas Go handles the scheduling of lightweight goroutines onto logical processors, which are subsequently bound to OS threads [50]. As shown in Figure 3.4, the example illustrate two OS threads (M2 and M3), with goroutines identified by the prefix *G*. The scheduling process involves allocating goroutines to logical processors via a local run queue. However, initially, goroutines are placed in the global scheduler run queue, and only afterward are they assigned to the local queues of logical processors [50, 51]. The same Figure 3.4, also illustrates how Go achieves parallelism. Goroutines can be distributed across multiple CPU cores if the hardware supports parallel execution [50].

Channels

Building on the CSP model, Go integrates channels as a core element of its concurrency paradigm. These data structures facilitate safe and efficient communication between goroutines, enhancing synchronization while addressing common challenges associated with shared

memory access [50]. By following the principle that only one goroutine should modify a piece of data at any given time, channels help ensure data integrity, reducing the risk of concurrent modification and giving predictable behavior in concurrent programs [51].

However, channels in Go do not inherently enforce data access protection features such as immutability and isolation, which are fundamental to the concurrency model found in Elixir and Scala when well implemented [50]. These languages are specifically designed to facilitate effective concurrency management, like analyzed earlier. Nevertheless, it is possible to adopt a strategy of using immutable data within Go channels, where all information is a copy of the original data [51]. This approach aligns more closely with the methodologies employed by Elixir and Scala, even though it is not the primary purpose of channels in Go.

Garbage Collector

In Go, the garbage collection strategy is based on a concurrent, non-generational mark-and-sweep algorithm, which operates globally on the heap [4, 54]. This approach differs significantly from the BEAM's garbage collector, which performs garbage collection on a per-process basis. The BEAM's process-level garbage collection minimizes the impact on overall system performance by isolating garbage collection events to individual processes [3].

Go's garbage collector, while global in nature, supports partial heap collection but still experiences "stop-the-world" pauses. Despite these challenges, it is designed to maintain pause times between 10 ms and 100 ms under heavy load, making it comparable to the G1 garbage collector in the JVM, which, as studied by Zhang et al. [55], can experience pause times ranging from 0 to 300 ms. In contrast, ZGC achieves significantly lower pause times, typically between 0 and 0.1 ms, making it an attractive option for latency-sensitive applications, although it is not compatible with Go's environment [52].

Each garbage collection approach has its own advantages and trade-offs. Elixir's garbage collection, rooted in the BEAM runtime, is particularly well-suited for concurrent programming due to its strong emphasis on process isolation [26]. In contrast, Go supports a partial heap-targeted garbage collector, which pairs effectively with its lightweight goroutines, enhancing memory management in concurrent applications.

Distributed Communication

A notable limitation of Go is its lack of native support for distributed communication [51, 56]. While the combination of channels and goroutines serves as an excellent tool for managing concurrency and parallelism, it does not inherently extend to communication across physical machines [50]. This limitation has prompted efforts to extend Go's concurrency model to support distributed systems. For instance, Whitney et al. [56] proposed a novel protocol called Gluster, which provides a library to abstract cluster logic and facilitate distributed communication. However, Gluster has seen limited industrial adoption and is restricted to Linux environments, limiting its general applicability.

Nevertheless, Go it offers seamless integration with mature and optimized networking libraries [50]. Packages such as TCP, HTTP, and gRPC provide efficient mechanisms for enabling communication between distributed components [4, 52]. These libraries significantly reduce the overhead associated with managing low-level networking concerns. Furthermore,

channel-based networking libraries allow the management of distributed interactions effectively, leveraging goroutines and channels to handle the inherent asynchronous aspects of the network calls [52].

While Go's native concurrency primitives do not align with the Actor Model, there are projects of the Actor Model available for the Go ecosystem. One mature example is Proto.Actor, a library that abstracts the complexities of distribution through its API [42, 56]. Built on top of gRPC, Proto.Actor provides a remote facilities and location transparency of the Actor Model within Go [42].

Go's rising popularity in the industry is closely tied to its adoption in microservices architectures and cloud-native applications [53, 54]. Microservices, by their nature, represent distributed systems and facilitate communication through both asynchronous and synchronous methods, often utilizing discovery services to map all nodes. Another approach involves the use of message queues, which can provide location transparency for processes [53].

3.3.2 Fault Tolerance Mechanism and Strategies

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

Error Philosophy

Until now, the “let it crash” philosophy has been described, a core principle applied in both Elixir and Akka due to the inherent design of the Actor Model, and even in Proto.Actor. This approach is based on the inevitability of errors, allowing them to occur and relying on mechanisms like the supervisor pattern to detect and recover from them [26]. However, the error-handling philosophy in Go is fundamentally different, representing almost the opposite paradigm. In Go, the strategy emphasizes handling every error explicitly [4, 50]. Errors are treated as first-class citizens, returned as values, and must be actively managed by the program. Unlike languages such as Scala and even Elixir, Go does not include mechanisms like try-catch for error handling [39]. Instead, it enforces a more explicit style that requires developers to check for and respond to errors immediately after an operation [51].

This philosophy aligns with Go's overall design principles of simplicity, clarity, and explicitness. It is supposed that requiring developers to handle errors explicitly, Go minimizes the risk of overlooking potential issues. While this approach can result in more verbose code, it aims to reduce the likelihood of unhandled exceptions and promote a clearer method of error management [4, 50].

Another important consideration in Go's error handling philosophy is its impact on code readability and maintainability. The explicit nature of error handling in Go often leads to repetitive code blocks, resulting in more boilerplate compared to the code styles of Elixir and Akka [4, 50]. However, this explicitness can facilitate tracing how errors are propagated and resolved within a program.

Fault Tolerance Mechanisms and Strategies

While Go was not primarily designed with built-in fault tolerance mechanisms, as it emphasizes efficiency and simplicity, it has become a fundamental component in distributed systems, such as Kubernetes [50, 52]. When combined with appropriate patterns, architectural approaches, and libraries, Go enables the development of fault tolerance capabilities within these systems.

A more suitable approach in Go combines the heartbeat pattern with panic/recover mechanisms [51]. In this pattern, a goroutine functions as a supervisor, monitoring other goroutines through periodic status updates known as heartbeats. If a monitored goroutine fails to send a notification within the expected timeframe, the supervisor can initiate recovery procedures to restore the failed component's state [4, 51]. This ensures that failures are detected and addressed quickly. Furthermore, the supervision mechanism can be enhanced by Go's panic/recover pattern, which enables the system to capture and handle critical errors [4].

In the context of microservices architecture, Go provides robust support through the mature Go-kit[5] library [57]. This library facilitates the development of microservices and distributed systems by implementing essential patterns such as circuit breakers, rate limiters, and distributed tracing capabilities [53, 57]. Additionally, this framework can be enhanced with the failsafe-go[6] library, which introduces additional aspects of fault tolerance, such as retry policies. Furthermore, integrating HashiCorp's Raft[7] implementation can provide strong consistency and leader election capabilities.

Many of these solutions can be viewed as generic strategies that are more architectural than native, relying on third-party libraries. This is similar to the practices observed in Elixir, which frequently utilizes third-party solutions for replication, as well as in Akka for Scala.

3.3.3 Go with Proto.Actor

As previously introduced, Go does not provide native capabilities for distributed or fault-tolerant systems. Unlike Elixir, which is built on the BEAM with built-in support for concurrency and distribution, or Scala with its Akka framework, Go requires external abstractions to achieve similar behavior. Proto.Actor serves this purpose by providing a cross-platform actor framework designed to bring structured concurrency and resilience to Go applications [42].

Roger Johansson, the creator of Akka.NET, and his team developed an innovative approach to implementing the Actor Model in Go. This implementation demonstrates that the Actor Model can be effectively combined with CSP, suggesting these paradigms are complementary rather than mutually exclusive [42]. As previously introduced, Proto.Actor, positioned as a next-generation Actor Model framework, introduces the "Actor Standard Protocol." This protocol establishes a language-agnostic communication standard across different programming languages, while maintaining fault-tolerant capabilities [42]. This design allows the Actor Model to be extensible to any programming language and runtime, avoiding the tight coupling seen in the BEAM ecosystem, where communication often necessitates the same underlying system.

[5] Go-kit: `https://gokit.io/` (accessed 29 June 2025)
[6] failsafe-go: `https://failsafe-go.dev/` (accessed 29 June 2025)
[7] HashiCorp's Raft in Go: `https://github.com/hashicorp/raft/` (accessed 29 June 2025)

Remote Communication Proto.Actor abstracts remote communication to achieve location transparency. At its core, it utilizes gRPC streaming for inter-node messaging, with Protocol Buffers handling serialization. Because gRPC is built on HTTP/2 and is language-agnostic, this choice ensures compatibility with a wide range of platforms and programming languages, making Proto.Actor a flexible option for building distributed systems.

Clustering Proto.Actor also supports clustering features, including node discovery, load balancing, and cluster membership management. These capabilities allow the construction of scalable systems composed of multiple cooperating nodes.

Fault Tolerance Proto.Actor incorporates all the Actor Model fault tolerance mechanisms such as supervision trees, restart and retry strategies, and isolation of failure. These mechanisms mirror those found in Akka and Elixir, providing structured recovery and enabling developers to build resilient systems in Go, which otherwise lacks built-in supervision semantics.

Although direct comparisons between Proto.Actor and Scala with Akka on the JVM are not widely available in benchmarks, existing tests for Proto.Actor indicate that it outperforms Akka.NET. Nevertheless, this particular performance advantage is unrelated to the JVM.

3.3.4 Challenges Compared With Akka and Elixir and Real Applications

After examining the Go language, it is clear that it does not lend itself to fault tolerance mechanisms so naturally. However, similar to how Akka enhances Scala, Proto.Actor leverages the combination of CSP with the Actor Model in Go [42]. Just as Elixir relies on third-party libraries, for instance, to implement Raft consensus, Go also requires external libraries to achieve fault tolerance capabilities.

According to the Proto.Actor benchmarking performance results [42], one test involved an initial actor spawning 10 new actors, each of which spawned another 10, continuing until a total of one million actors were created. Each actor returned its ordinal number, which was summed at the preceding level and sent back upstream to the root actor, resulting in a final sum in the range of 11 digits. In this test, Erlang outperformed the Actor Model implemented in Go.

In a different test, on the same source of Proto.Actor benchmarking performance results [42], two actors, one on each of two nodes, were used to exchange one million messages back and forth. In this scenario, Go surpassed Erlang in throughput, a result attributed to Go's use of message references, which likely reduced overhead.

A study conducted by Marchuk et al. [58] revealed that Elixir outperformed Go in both requests per second and messages per second during a load test simulating a backend scenario. This test underscored Elixir's superior performance and efficiency.

In a similar vein, Valkov et al. [5] found that Go, likely due to its use of typed channels and the absence of a need for pattern matching, achieved higher throughput compared to Scala with Akka and Erlang.

Real-World Applications Go is extensively used in cloud applications and high-performance systems, particularly within Google, where it was originally developed [54]. It plays a vital

role in platforms such as Docker and Kubernetes, and companies like Dropbox have successfully transitioned from Python to Go to enhance efficiency [4]. Another notable example is Cockroach Labs, which has praised Go's garbage collection and performance as well-suited to their requirements [4]. However, challenges do exist. For instance, Discord initially implemented Go but later migrated to Rust due to issues with Go's garbage collection, which resulted in significant latency spikes. This led to the conclusion that the garbage collector was a contributing factor to performance degradation [59].

3.4 Benchmarking Analyses

According to Almeida et al. [60], the primary objectives of benchmarking are to *"provide a practical way to characterize and compare systems or components according to specific characteristics (e.g., performance, dependability)*". Benchmarking delivers insights within a specific domain by quantifying key metrics, enabling practical comparisons. For results to be valid and meaningful, it is critical to conduct repeatable experiments. Benchmarking serves as an experimental approach that derives value from measurable outcomes, yielding consistent results under identical conditions, or statistically analyzed [60, 61]. Deterministic benchmarks are especially valuable as they ensure reproducibility when the same assumptions are applied.

Non-deterministic approaches are typically associated with chaos engineering, which focuses on testing system resilience by intentionally introducing random faults [31]. However, the insights gained from benchmarking in this context are relative and applicable only to the specific conditions under which the tests were conducted, due to the inherent randomness of the process. Consequently, these tests often lack reproducibility and may not offer comprehensive coverage across all system sizes [60].

On a overview, benchmarking, among others, can be generally divided into two categories [5, 60–62]:

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

The well-known Computer Language Benchmarks Game[8] supports various algorithmic benchmarks tests, evaluating runtime, memory usage, and related performance metrics [61]. However, this benchmarks does not support newer languages like Elixir, Scala, and Scala with Akka, nor does it address resilience focused benchmarks. This limitation highlights the need for strategies tailored to resilience benchmarks.

[8] Computer Language Benchmarks Game: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/` (accessed 29 June 2025)

3.4.1 Fault-Tolerant and Distributed Benchmarking

Benchmarking fault tolerance in distributed systems presents specific challenges, requiring evaluative strategies that extend beyond conventional performance testing. Key considerations include not only throughput and latency under normal conditions, or how the computational occurs under an algorithm execution, but also the ability to keep executing, to detect and recover from faults, including node failures, communication interruptions, and state inconsistencies [31, 60, 61].

The objective is to develop benchmarks that tries to simulate real-world errors. These benchmarks should try to closely reflect actual software behavior, taking into account factors such as overload conditions and software faults. In fault tolerance benchmarks, it is crucial to incorporate components that introduce faults in order to achieve dynamic accuracy [60]. This can be accomplished through random fault injection methods, such as Chaos Monkey, developed by Netflix, which randomly destabilizes the system. Another strategy is through deterministic fault injections, where specific errors are deliberately introduced into the system and it is known what happened in order to corelate the metrics with the faults [31].

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

Strategies

The Computer Language Benchmarks Game, as previously introduced, provides a foundational collection of micro-benchmarks designed to compare the performance of programming languages. Although it does not include distributed systems and lacks support for some newer languages, it has a set of algorithms that have served as the basis for other benchmarks, such as the work by Cardoso et al. [63]. This particular benchmark focuses on the agent and Actor Model, leveraging three algorithms from the collection: Fibonacci numbers, token-ring[9], and chameneos-redux[10]. These algorithms, each tailored to distinct computational tasks, offer a valuable resource for designing performance tests in a variety of contexts [31, 63]. They can be used in conjunction with resilience benchmarks to simulate processing, for example.

For instance, Savina, developed by Imam and Sarkar [62], has emerged as the de facto benchmark for evaluating Actor Model performance [61]. It utilizes micro-benchmarks alongside concurrency and parallelism techniques to assess the behavior of actor-oriented programs in compute intensive applications. While Savina provides valuable performance insights, its scope is restricted to JVM-based languages, such as Akka with Java, thereby excluding languages like Elixir, Go, and Scala with Akka.

Savina employs 28 benchmarking strategies, categorized into three groups: 7 for micro-benchmarking, 8 for concurrency, and 13 for parallelism. These benchmarks evaluate critical aspects such as communication efficiency, node creation and termination, mailbox contention, IPC resource allocation, among others [61, 62]. However, Savina's focus is confined

[9] The token-ring algorithm simulates a network of nodes passing tokens in a circular manner, which is useful for evaluating message-passing and synchronization in concurrent systems [63].

[10] Chameneos-redux is a concurrency benchmark that models a group of agents (chameneos) interacting with each other based on color, showcasing the complexities of state management and communication in concurrent programming [63].

to actor-based systems and localized performance, excluding non-actor-based and distributed communication scenarios.

Blessing et al. [61] highlighted significant limitations in the micro-benchmarking Savina, particularly their narrow focus on isolated performance metrics. To address these issues, they proposed an application-oriented benchmarking approach that simulates real-world scenarios, such as a simpler version of a chat application like Facebook. This approach moves away from isolated micro-benchmarks to focus on a more application-level perspective.

Their proposal centers on creating a comprehensive application that encompasses multiple scenarios. This approach aims to make benchmarks more relatable to real-world applications, encouraging developers to optimize the all system rather than focusing on discrete, independent cases [61]. To enhance flexibility, they propose the addition of tunable options, such as varying the number of load balancers, if applicable.

Despite its innovative approach, the implementation by Blessing et al. [61] had some shortcomings such as not supporting resilience testing. These limitations highlight opportunities to design advanced application-based benchmarks that incorporate fault tolerance techniques and aims to broader systems.

Randtoul and Trinder [31] present a study focused on evaluating the resilience of actor-based systems under fault scenarios through fault injection techniques. The authors did not used chaos engineering due to its inherent unpredictability, opting instead for a deterministic approach. This choice ensures reproducibility and controlled experimentation. Being deterministic, also facilitates a clear relation between fault inputs and the resulting metrics. This particular benchmark operates and is designed solely to run on a single physical machine and lacks distributed semantics.

Metrics

Metrics are a fundamental aspect of the benchmarking process, providing a quantitative basis for evaluating system characteristics. They represent measurable outcomes of system execution and are essential for drawing meaningful conclusions [60].

In the observed context, metrics are generally categorized into two main types: performance and resilience metrics [5, 31, 60]. Performance metrics assess system efficiency and responsiveness, either independently or in conjunction with fault tolerance, to evaluate how the system performs under normal and faulty conditions [5]. In contrast, fault tolerance metrics specifically examine a system's ability to manage and recover from failures, including metrics such as recovery and detection time. Combining these perspectives provides a comprehensive evaluation, offering deeper insights into system performance and reliability [31].

Chapter 4

Architecture and Test Design

4.1 High-Level Architecture Proposal

Building upon existing literature, this section proposes a benchmarking architecture designed to evaluate fault tolerance and performance characteristics across multiple paradigms, using micro-benchmarking within a generic real-world application. This concept aligns with suggestions by Blessing et al. [61] and Randtoul and Trinder [31], who both proposed chat-based systems as effective case studies. Inspired by their approach, the proposed architecture emphasizes strong configurability, enabling adaptations without requiring code modifications.

The use of a chat application as the core model provides extensive flexibility and simplifies the integration of new features [61]. It also reflects realistic usage patterns, particularly the client-server communication model. A key enhancement in this architecture is the inclusion of distributed capabilities, supporting extensibility and addressing limitations in prior implementations [31, 62, 63].

Another fundamental design principle is the separation of metric calculation logic from the core benchmarking implementation. In contrast to Randtoul and Trinder [31], where metrics were computed within each language runtime, this architecture delegates all metric-related processing to a dedicated external component. This separation is enabled via asynchronous event communication, typically using a message queue. As a result, new language implementations can focus exclusively on the benchmark logic, reducing integration complexity and ensuring consistent, fair measurement across heterogeneous environments. It also minimizes measurement overhead, which can otherwise bias results.

As shown in Figure 4.1, the architecture consists of server groups and client groups. Each server group contains a supervisor and its associated chat instances. These groups may be deployed across multiple nodes, allowing configurable supervisor-to-instance ratios, a strategy inspired by the benchmarking approach described by Randtoul and Trinder [31]. Chat instances may be either stateful or stateless, supporting experiments on how state affects fault tolerance and performance.

The chat component could support conversations between clients, including the exchange of media, and allow for the persistent historical storage of conversations. With stateful chats, it is possible to observe how recovery can be influenced by the presence of state. The media referred to can be random bytes mimicking a photo or video.

The client group include the clients, fault injectors, and a publisher component. Clients and fault injectors communicate directly with chat instances. Clients should be capable of sending messages of varying types and can exhibit different behaviors upon receiving responses. As previously introduced, clients can send mock data simulating video or images,

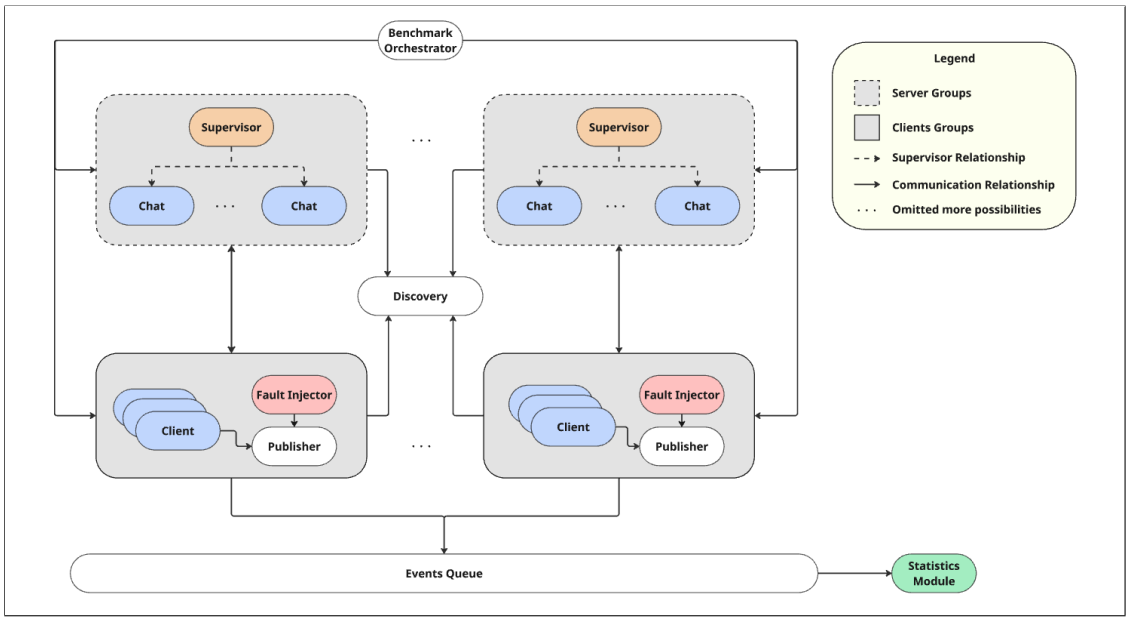

Figure 4.1: High level test benchmark architecture design proposal

and they can also mimic interaction patterns from the Savina benchmark suite, such as the Thread Ring scenario, Ping-Pong, among others [62]. This allows for focusing on throughput, concurrency, parallelism, and other relevant message types to produce specific metrics.

Fault injectors are responsible for introducing failures at configurable intervals. These components are designed for extensibility and may simulate various types of faults, including resource exhaustion through high-complexity tasks, for instance, large Fibonacci calculations or matrix operations, abrupt termination of services, or injection of malformed or oversized messages to stress memory and network subsystems [60, 62].

The publisher component has the mission of abstracting communication with the external world, in this case, the statistical component. The publisher must incorporate an asynchronous mechanism to quickly free the client, allowing the client to remain focused on its benchmarking logic. Upon the client's order, the publisher is responsible for guaranteeing that the event reaches an intermediary, from which the statistical module can then fetch it.

To enable dynamic communication and coordination, a centralized discovery server maintains a registry of active chats and supplies connection information to clients. This supports scalable and decoupled service discovery in distributed deployments. This component could also be a target for replication, for example, through the use of the Raft consensus algorithm, which would allow for a new set of tests. These tests, focused on a consensus algorithm, could aim to find the mean time taken to elect a new leader, among other aspects, with the objective of comparing how each Raft implementation is optimized in each language.

An orchestrator component manages the benchmarking process. It handles remote deployment, test configuration, and ensures system correctness before benchmarks begin. The orchestrator is responsible for issuing commands to initiate execution and verifying all operational preconditions.

Finally, this architecture proposes a dedicated mechanism for metric computation. This design offloads metric calculation to an external system, ensuring uniform analysis across

implementations. Additionally, it supports deferred processing of metrics, allowing evaluations to take place after benchmark execution has completed and system components have shut down.

This architecture is proposed as a general and extensible model for standardized benchmarking. It is subject to refinement as new experimental needs arise or new use cases are explored.

4.2 Concrete Test Architecture and Design

Based on the previously described high-level proposal, this section details how each component is specifically designed for this benchmarking setup. Given current limitations in both time and resources, the implemented benchmarking is restricted to a single physical machine. However, the implementation incorporates extensibility, thereby enabling future expansion into a multi-node cluster environment. While the scope of this dissertation is limited to a single-node deployment, it is crucial to recognize that this operational constraint may influence the generalizability of the obtained results. This limitation will receive further elaboration regarding its potential negative aspects.

Client Functionality For the client functionality, among the possible design paths, the chosen approach is inspired by the Ping-Pong mechanism [31, 61, 62]. In this mechanism, two components exchange messages. In this specific benchmarking scenario, clients will send *ping* messages, and the server will broadcast *pong* messages. The client's primary responsibility is to connect to the chat instance, which requires querying the discovery server. After connection, depending on the specific test use case, it needs to send messages and be prepared to receive them. Concurrently, it will monitor the chat instance so that it can initiate a reconnection action when a disconnection is detected.

Fault Injector Regarding the fault injector, the selected approach is to abruptly crash the server at a specified time, causing the chat to terminate and all connected clients to be notified. This design decision specifically aims to measure the reconnection time and the detection time of errors, as well as the throughput under a faulty system. There will always be one fault injector per chat instance in the system.

Supervisor The supervisor component is responsible for managing a specified number of chat instances. It will follow a *one-for-one* supervision strategy, meaning that if one chat instance fails, it is restarted with a fresh instance. This forces the child instance to re-execute its setup logic from the beginning.

Chat Instance Each chat instance follows a linear sequence of operations. It first connects to the discovery server and then accepts incoming client connections to build its list of associated clients. Each instance is assigned a unique chat topic, which serves as a unique identifier. Registration with the discovery server is mandatory for a chat instance to accept messages. Upon receiving a message from any client, the chat instance broadcasts it to all its connected clients. Additionally, the chat needs to allow an abrupt disconnection whenever the fault injector triggers that operation.

Publisher Component This component has the sole responsibility of sending events to the message queue. This design frees all other components from this task, allowing them to implement a "fire-and-forget" logic for event publishing.

Statistical Component The statistical component is implemented as a separate project. All other components are only required to send relevant event data to a message queue, which will subsequently be read by the statistical module.

Orchestrator The orchestrator facilitates the creation and coordination of all components. It begins by instantiating the publisher component and the discovery server. Subsequently, the orchestrator proceeds to create all server groups and client groups, awaiting their successful connection. Finally, it sets the initialization and termination times for the benchmark and communicates these parameters to all participating components. All components incorporate a self-termination call triggered by the stop timestamp provided by the orchestrator.

4.2.1 Test Scenarios

Having described the architecture, it is now detailed the implemented and analyzed test scenarios. The primary focus is to evaluate how the languages could maintain throughput despite fault injection, and how effectively they detect and recover from faults, specifically transient faults.

To support the configurability aspect of these tests, a dedicated configuration file was created, as detailed in Table 4.1. This table outlines the specific values used across the various test scenarios, and this configuration file serves as the primary entry point for test execution.

Table 4.1: Test parameters

Name	Type	Range	Description
test_type	String	[detection_time, reconnection_time, throughput]	Specifies the type of test being conducted.
num_supervisor	Integer	≥ 1	The number of supervisors involved in the test.
chats_per_sup	Integer	≥ 0	The number of chats assigned to each supervisor.
clients_per_server	Integer	≥ 1	The number of clients supported by each chat server.
fault_pause_ms	Integer	≥ 0 ms	The duration of the pause, in milliseconds, after a fault occurs until another is triggered.
msg_type	String	[error, none]	The type of message being sent by fault injector.
test_duration_secs	Integer	≥ 1 sec	The total duration of the test in seconds.

All tests reported in this document were conducted with a single supervisor. Although a dynamic supervisor-to-supervised ratio was explored, these specific results have been omitted. This is because, unlike the variations observed by Randtoul and Trinder [31] when investigating high supervisor responsibility, the tests performed for this document, covering ratios from

1:1024 to 1024:1, consistently yielded the same outcomes. Therefore, to avoid redundancy and duplicated conclusions, all presented tests utilized a single supervisor focusing solely on the server-clientx ratio.

Throughput Test Case

This test case evaluates how different language implementations maintain message throughput under stress conditions. Clients are configured to send messages to the server at a fixed interval. Upon receiving each message, the server broadcasts it to all connected clients. Each recipient client then asynchronously notifies the publisher component, enabling the message reception to be recorded for subsequent metric analysis.

To simulate adverse conditions, the fault injector periodically crashes the server. The frequency of these induced failures is configurable so it can simulate different levels of system instability. Each crash triggers the reconnection process for all affected clients and chat instances, adding pressure to the system's recovery mechanisms.

The primary objective of this test is to observe which language implementations can sustain the highest throughput despite transient failures. By applying a range of fault injection rates, from frequent disruptions to periods of stability, the benchmark aims to reveal the resilience and efficiency of each implementation under dynamic and potentially unstable conditions.

Reconnection Time Test Case

The focus of this test is to evaluate the time required for each language implementation to re-establish client-server communication following a transient server failure. The fault injector is configured to deliberately crash the server at fixed intervals, simulating realistic and time-bound fault conditions. Upon failure, each client must detect the disconnection and autonomously initiate the reconnection process. Simultaneously, the supervisor is responsible for restarting the failed server process, which must then re-register with the discovery component before it can resume accepting client requests. The measured metric is the elapsed time between the moment of disconnection and the full restoration of a stable and operational client-server communication channel.

The primary objective of this test is to analyze the mean time to recover under controlled transient fault conditions. This scenario highlights how each runtime handles short-lived disruptions and the overhead involved in restoring connectivity.

Detection Time Test Case

This test case closely mirrors the reconnection time scenario but focuses specifically on the time required for each client to detect a failure of a chat instance. The fault injector deliberately crashes the server, and clients continuously monitor the connection state to recognize disconnection events. The recorded metric is the elapsed time between the moment the server crashes and the moment the client detects the disconnection.

This test emphasizes the role of each language's supervision and fault monitoring mechanisms, with the scope of observing how quickly each architecture and approach can facilitate notification.

4.2.2 Practical Implementation

The benchmarking architecture has been implemented in three different programming languages: Elixir, Scala with Akka, and Go using Proto.Actor. These languages were selected for their support of the Actor Model and their relevance in concurrent and distributed systems development, as described earlier. The goal is to provide a fair and consistent comparison across implementations by adhering to the same architectural paradigm, despite the inherent differences in language syntax, runtime environments, and concurrency models.

To ensure fairness, all implementations avoid leveraging third-party optimization techniques or infrastructure-level enhancements, such as external caching layers. This approach allows the focus to remain strictly on the Actor Model implementations and their capabilities for concurrency, fault tolerance, and distribution.

A key requirement for each implementation is support for distribution, even in single-node environments. This ensures that the architecture remains extensible and applicable to real-world, distributed deployments. Therefore, the discovery server must be located and accessed using a generic distribution mechanism. All component instances must be capable of discovering the location of the discovery server without having prior knowledge of its address.

All implementations need to be able to send events to a message queue, with all events conforming to the same format and types. This ensures that the statistical component remains language-agnostic and can interpret the events in a standardized format.

Elixir Implementation Details

The Elixir implementation is built upon on the OTP, a mature and robust foundation for constructing concurrent, distributed, and fault-tolerant applications, like detailed. The architecture key components such as clients, chat servers, and the discovery service, all of them uses `GenServer` processes, while the process supervision and lifecycle management it is delegated to a `DynamicSupervisor`. Interactions between components rely on asynchronous message passing, which preserves actor isolation and supports failure transparency.

Service discovery is implemented using the `:global` module, which enables named process registration across distributed nodes. By registering the discovery server under a globally unique identifier, other components can resolve its Process Identifier (PID) dynamically at runtime using `:global.whereis_name`, eliminating the need for static configuration or hardcoded addresses. Internally, the `:global` module replicates registration metadata across nodes and employs distributed mutual exclusion to serialize updates and prevent name collisions [39]. While functionally analogous to the `Receptionist` in Akka, this approach does not provide event-based subscriptions or update notifications. Consequently, processes that require fault detection must explicitly establish monitors via `Process.monitor` to track failures and recover accordingly.

State management in both chat and discovery components relies on immutable `Map` structures, reflecting Elixir's functional programming paradigm. The implementation deliberately avoids shared-memory constructs, including Erlang Term Storage (ETS) tables or agent-based coordination, in order to maintain fairness with actor-based systems implemented on other runtimes. This design choice enforces fairness and consistency across implementations, and it avoids optimization artifacts specific to the BEAM VM.

Supervision is configured using a *one-for-one* strategy under `DynamicSupervisor`, ensuring that failures in individual chat processes trigger isolated restarts without cascading effects. This aligns with supervision semantics in actor-based models such as Akka and Proto.Actor, where failed actors are replaced by fresh instances with new identifiers and reinitialized state. This behavior facilitates client-side fault handling, allowing reconnection logic to be triggered upon termination events detected via monitors.

To simulate faults and test failure handling, abrupt process termination is induced using `Process.exit(pid, :kill)`. This bypasses normal termination routines and prevents message processing during shutdown. This method permits precise control over the fault injection timeline and avoids backpressure associated with mailbox exhaustion or blocking operations.

Temporal behaviors such as timeouts, retries, and periodic actions are implemented using `Process.send_after`, which enables non-blocking message scheduling within actors. This technique preserves system responsiveness and avoids the nondeterminism introduced by blocking operations such as `:timer.sleep`, which are intentionally excluded from the all implementations.

Client processes establish fault monitoring relationships with their respective chat servers via `Process.monitor`. Upon detecting a termination, clients receive a `:DOWN` message. In a justified test case, this message triggers communication with the `Publisher` component to register the occurrence.

All inter-process relationships are established via monitoring rather than linking due to the unilateral form of connections among components. Client reconnection logic is implemented as a bounded retry loop with fixed delays, there is no use of backoff. Internal coordination among components is strictly message-based, without reliance on pub-sub mechanisms. The system uses non-natively minimal test instrumentation, with failure simulation and delayed messaging exclusively used via native OTP primitives, such as `Process.exit` and `Process.send_after`, trying to preserve implementation uniformity across platforms.

Scala with Akka Implementation Details

The Scala implementation is built on the Akka Typed API, which enforces type safety through the `Behavior` interface [41]. Core system components, including chat instances, clients, and the discovery service, are implemented as typed actors, communicating exclusively through strongly-typed, immutable messages. This model promotes deterministic concurrency, fault isolation, and location transparency.

Service discovery is implemented via Akka's `Receptionist`, allowing actors to register under a `ServiceKey` and enabling others to subscribe for updates. The actors subscribes to `Discovery` listings using a message adapter, ensuring it dynamically receives the updates of discovery actor reference. This mechanism supports distributed dynamic updates:

```
val receptionistSubscriber: ActorRef[Receptionist.Listing] =
  context.messageAdapter {
    case Discovery.serviceKey.Listing(set) => ChangeDiscovery(set.headOption)
  }
context.system.receptionist ! Receptionist.Subscribe(
  Discovery.serviceKey, receptionistSubscriber
)
```

Unlike Elixir's `:global` registry, Akka's discovery model allows actors to respond to runtime changes in system topology without requiring a centralized lookup or explicit polling.

All actor state is immutable and updated through functional behavior transitions. For example, the set of connected clients in a `Chat` actor is updated by returning a new behavior with an updated client set. This design aligns with Akka's functional programming principles, ensuring clarity and concurrency safety under load. Actor lifecycle events are managed via Akka's `Watch` mechanism. In the discovery service, registered chat actors are monitored, and any termination triggers a cleanup of the internal mapping:

```
context.watch(ref)
...
.receiveSignal {
  case (_, Terminated(ref)) =>
    discovery(publisher, chats - ref.path.name)
}
```

To simulate hard node crashes during benchmarking, actor termination is triggered using Akka's classic API. This bypasses the normal message-handling lifecycle and avoids any graceful shutdown, ensuring behavior that more closely matches low-level process termination:

```
val classicActorRef = chat.toClassic
context.system.toClassic.stop(classicActorRef)
```

Although Akka supports a range of supervision strategies, this benchmark enforces termination and clean restart to maintain fairness with Elixir and Go implementations. Upon failure, actors are reinitialized, and watchers are notified through `Terminated` messages, ensuring consistent fault propagation and state recovery.

Scheduled behaviors, such as message emission or test shutdowns, are implemented using Akka's internal timer system through `scheduleOnce`. The objective is for an actor to send a message to itself after a specified delay with non-blocking delays, mimicking the behavior of Elixir's `Process.send_after`.

```
context.scheduleOnce(200.millis, context.self, ProcessMessage)
```

Clients interact with chat actors through a registration service, where each chat instance maintains a unique set of clients and rejects duplicate registrations. Upon connection, clients receive a reference to the chat actor. Broadcasts and reconnections are handled asynchronously, and all relevant events are forwarded to a publisher actor, which aggregates and relays the information to the external messaging system.

Go with Proto.Actor Implementation Details

The Go implementation uses the Proto.Actor framework, a cross-platform Actor Model system inspired by Akka [42]. All components implement the `Actor` interface, where each defines its message types and a `Receive` method to handle them.

It is necessary to utilize the `Proto.Remote` module to enforce distributed communication. This module provides a robust communication abstraction built upon gRPC streams, effectively managing inter-node interactions and actor instance communication. `Proto.Remote`

uses gRPC's bidirectional streaming capabilities as its foundational transport layer for distributed actor communication, ensuring high efficiency, scalability, and cross-language interoperability by using Protocol Buffers for message serialization. This design allows for transparent communication between actors regardless of their physical location.

Service discovery is also provided via the `Proto.Remote` module, which allows actors to be spawned under globally known names using `remote.SpawnNamed`. This abstraction mirrors the functionality of Elixir's `:global` and Akka's `Receptionist`, enabling name-based actor resolution across the system.

State is managed using native Go `Map` structures local to each actor. Although Go permits mutable state, Proto.Actor guarantees that each message is processed synchronously, thereby maintaining a consistent state without necessitating the use of explicit locks. Consequently, it is guaranteed that no more than one actor can access the same state concurrently.

Supervision is implemented through Proto.Actor's built-in supervisor strategies. Similar to Akka, child actors can be restarted internally without external awareness. However, the implementation explicitly forces complete termination and restart of failed actors to ensure monitoring systems detect the change and to trigger reconnection logic.

Timers and scheduling are implemented using Proto.Actor's support for sending delayed self-messages, such as `scheduler.SendOnce(time, ctx.Self(), &type)`. This enables periodic behavior like retries, timeouts, or simulated delays, implemented without blocking or shared timers.

Clients use the `Watch` feature to monitor their associated chat actors, similar to the other implementations. This guarantees that all monitoring processes are notified upon an abrupt crash of the monitored process.

Statistical Component

The statistical component is implemented in Python and consumes events through a RabbitMQ message queue, which acts as the intermediary for event delivery. Upon startup, the component receives a test identifier and processes only the events associated with that specific test until a new start. It also requires explicit start and termination timestamps for each test run, allowing it to determine the acceptable time range of events.

Each test type generates events with distinct structures and semantics:

- **Throughput Test**
  - `eventId`: Unique event identifier
  - `timestamp`: Time the message was received
  - `type`: `MessageThroughput`
  - `name`: Client's name
- **Reconnection**
  - `eventId`: Unique event identifier
  - `type`: `Reconnection`
  - `name`: Client's name

  - `value`: Time taken to reconnect in ms

- **Detection injection from the Injector**
  - `eventId`: Unique event identifier
  - `type`: `FaultInjection`
  - `timestamp`: Time when fault was injected
  - `name`: Injector's name
  - `value`: Uniquely identifier of the injection
- **Detection from the Client**
  - `eventId`: Unique event identifier
  - `type`: `Detection`
  - `timestamp`: Time when failure was detected
  - `name`: Client's name
  - `value`: Uniquely identifier of the injection

For throughput analysis, events are grouped into time buckets representing one-second intervals. This allows for rapid aggregation of messages over time. In contrast, reconnection events are merged into a single timeline and processed collectively due to the already calculated nature.

Detection analysis involves associating the fault injection timestamp with the detection times reported by each client. By computing the deltas between injection and detection timestamps, detection latency metrics across clients are derived, which are then aggregated for statistical evaluation.

Test Automation

To facilitate large-scale testing and ensure consistency across executions, a set of automation scripts was developed. These scripts are responsible for generating all intended combinations of configuration parameters and executing entire test suites in batches.

Before each test begins, the scripts attempt to ensure a clean and cold environment by terminating any previously running processes associated with the system under test. For instance, in the Elixir implementation, the BEAM VM is explicitly shut down prior to execution to avoid residual state or orphan processes from influencing the results. Similar precautions are taken for the Scala and Go environments to promote fair and reproducible benchmarks.

4.2.3 Limitations and Considerations

In addition to the constraints already discussed, it is important to acknowledge certain technical limitations that help contextualize the results and prevent misinterpretation. These limitations are not intended to discredit the results but to clarify their scope and encourage informed interpretation, especially when extending this work to broader or production-like environments.

Benchmark Variability Each scenario was executed multiple times to mitigate the influence of outliers. Nonetheless, shared system resources, such as CPU scheduling and RAM memory usage, can introduce variability between runs. Therefore, repeatability was employed as a strategy to account for this variability.

Garbage Collection Effects The garbage collection behavior was not explicitly measured, logged, or tuned in any implementation. Each language uses its own memory management strategy, and while garbage collection pauses could influence results, such effects fall outside the focus of this study.

No VM or Runtime Tuning All benchmarks were executed using default runtime, compilation and VM settings. No platform specific optimizations were applied. This choice supports fairness across implementations, though it may limit the performance of individual runtimes.

Local-Only Network Communication Despite all implementations leveraging the remote communication mechanisms provided by the language API, thus enforcing distributed semantics, all components were executed on a single physical machine. This setup does not fully reflect real-world distributed conditions and introduces limitations to the conclusions drawn from the results.

Message Queue Co-Location The message queue, in this case a RabbitMQ instance, responsible for event logging, was deployed on the same host as the benchmarked components within a containerized image. Although statistical processing, which will read these messages, is executed after the test completes, real-time message transmission between the components and the message queue could contribute to destabilize the benchmark's results.

Chapter 5

Experiments Results

This section presents the execution results and corresponding analysis of the test cases introduced earlier. Each experiment was conducted over a fixed duration of 60 seconds. All tests were performed on a machine equipped with 16 GB of RAM and an 8-core Apple M3 CPU, running macOS Sequoia 15.0.1. The software environments were configured with Elixir 1.18.3 (compiled with Erlang/OTP 27), Go 1.23.4 using Proto.Actor 0.4.0, and Scala 3.3.5 with Akka 2.10.2.

The concurrency load limits for these tests were set at approximately 50,000 concurrent actors maximum. This decision was primarily driven by the physical limitations of the machine used for the experiments, which helped ensure a fair comparison across all implementations. Beyond this threshold, results became unpredictable, and system warnings, such as `{:alarm_handler, {:clear, :system_memory_high_watermark}}`, indicated that the BEAM was reaching high memory usage mark. To maintain the integrity and comparability of the data, results falling outside this defined range were excluded from the analysis.

For the mean time to reconnect and detection time measurements, each test was executed three times, and the mean was calculated for all values. For throughput, the same averaging process was not employed, as one of the goals is to observe variability, which is illustrated through the box plot graph. Nevertheless, multiple tests were conducted for throughput as well, and the presented results align with the overall observations.

5.1 Throughput Under Failure Test Results

Both Figure 5.1 and Figure 5.2 illustrate the throughput performance of the three actor-based implementations, Elixir, Scala with Akka, and Go with Proto.Actor, under varying fault injection intervals. The experimental setup simulates a workload consisting of 256 concurrent chats, each with 5 clients. This configuration was chosen to establish a balanced ratio between actors and clients, ensuring comparable reconnection and failure detection times across platforms, thus reducing external variability in the throughput measurements.

The X-axis in both figures represents the fault injection interval, ranging from 300 ms to the scenario with no injected faults, in this case marked with 60,000 ms. This interval determines the deterministic point at which transient failures are introduced. The Y-axis reports throughput in messages per second.

As illustrated in Figure 5.1, throughput consistently increases across all implementations as the fault injection interval becomes longer. With fewer interruptions, each system is able to recover more efficiently, allowing throughput to approach levels comparable to a no-fault environment. This trend is particularly evident at the 10,000 ms interval, where all

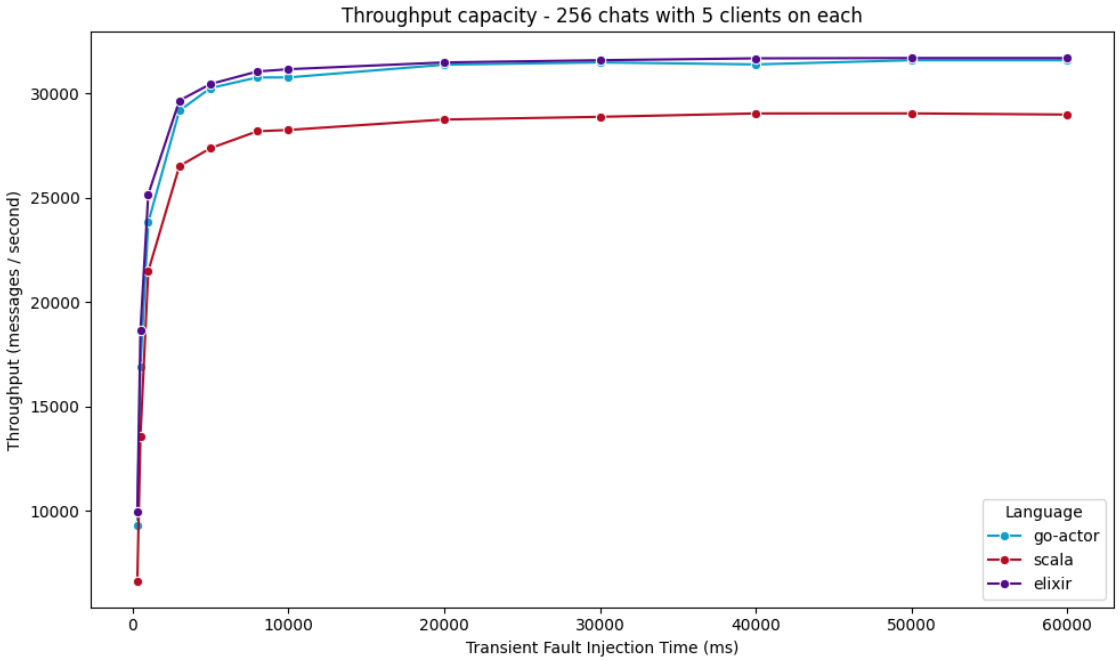

Figure 5.1: Throughput under failure line graph

three languages sustain consistent and nearly equivalent throughput, eventually converging with the performance observed under no-fault conditions. In contrast, within the 300 ms to 5,000 ms range, throughput it is significantly worse. This decline is primarily justified to the rapid and successive transient failures, which provoke reconnection procedures and stress the systems' ability to maintain stable communication flows.

From this observation, it is evident that Scala with Akka exhibits the lowest throughput performance under high fault rates when compared to Elixir and Go with Proto.Actor. Specifically, at the 300 ms fault injection interval, Scala's throughput is approximately ∼30% lower than the other two implementations. In contrast, Elixir and Go with Proto.Actor demonstrate comparable throughput levels under these high-frequency failure conditions, indicating a more robust handling of rapid transient faults.

Previous findings by Randtoul and Trinder [31], based on a Ping-Pong inspired benchmark, indicate that Erlang achieves approximately 10% higher throughput than Scala with Akka in fault-free conditions. These results are consistent with the observations in Figure 5.1, where Elixir outperforms Scala with Akka by approximately ∼30% at high failure injection rates. Under low-frequency fault injection, this lead narrows, with Elixir retaining a 10% advantage over Scala with Akka, while throughput becomes nearly equivalent to that of Go with Proto.Actor.

Throughput variability also distinguishes the implementations, as shown in Figure 5.2. Elixir maintains consistently high throughput with low variability across most intervals. From 1,000 ms onward, it demonstrates a narrow Interquartile Range (IQR), indicating stable and reliable performance even under moderate fault conditions.

Go with Proto.Actor also demonstrates relatively stable performance, but shows its highest variability between the 300 ms and 3,000 ms fault intervals. This fluctuation is likely attributable to the overhead of frequently establishing and tearing down gRPC bidirectional

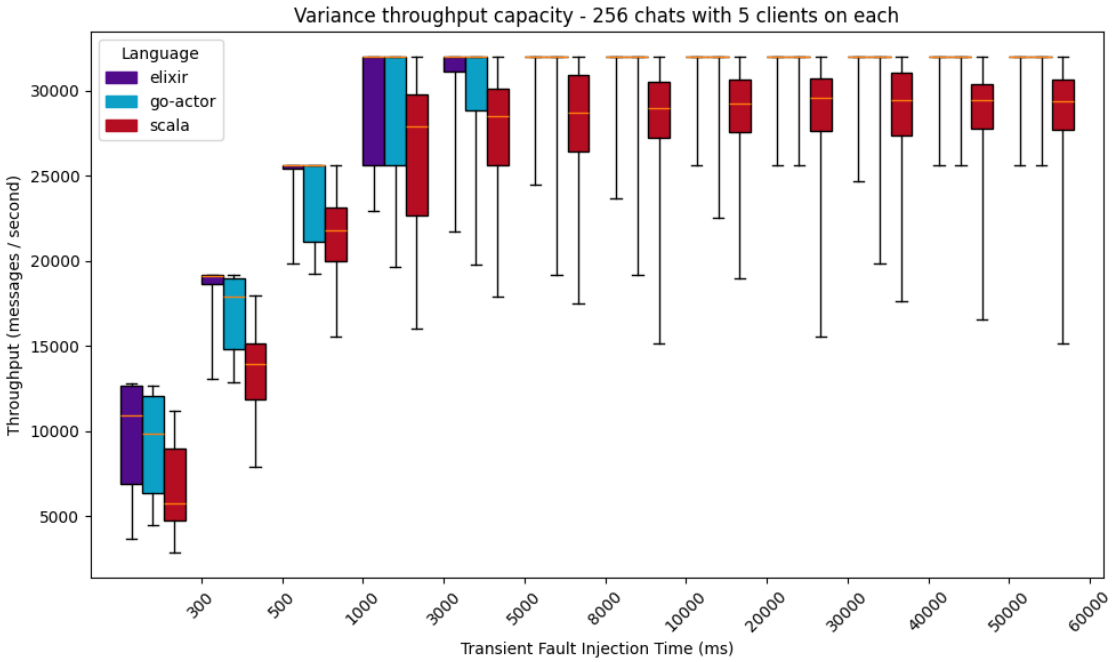

Figure 5.2: Throughput under failure variance on box-plot graph

streams during rapid failure and recovery cycles. As the fault interval increases, however, throughput consistently improves and stabilizes. Notably, from the 10,000 ms mark on forward, both Go and Elixir converge at a throughput of approximately 30,000 messages per second. This convergence suggests that Go's distributed runtime strategy, particularly its management of gRPC streams, supports predictable and scalable performance under low fault pressure. Although the study by Valkov, Chechina, and Trinder [5] did not consider distributed semantics, it similarly found that Go, using only local channels, outperformed Scala with Akka and Erlang in terms of raw throughput.

In contrast, Scala with Akka exhibits the highest variability across all fault injection intervals. While it occasionally reaches throughput peaks comparable to those of Elixir and Go, its wide IQRs and extended whiskers indicate significant inconsistency in performance. This fluctuation persists even at higher fault intervals, such as 5,000 ms and above, where more stable behavior would typically be expected. These inconsistencies may be influenced by JVM-related factors such as thread contention or garbage collection overhead, though further investigation is required to draw definitive conclusions. Similar observations are reported by Valkov, Chechina, and Trinder [5] and Randtoul and Trinder [31], who identify higher communication latency in Scala with Akka compared to Erlang and Go. The present benchmark, incorporating distributed semantics, further supports these findings. This performance gap also may be attributed to factors such as message serialization and deserialization overhead in Akka's Artery transport, though a detailed comparison of the underlying communication protocols is necessary to confirm this hypothesis.

In conclusion, while all three Actor Model implementations scale and recover effectively under decreasing fault frequency, their performance stability under stress varies significantly. Elixir consistently demonstrates superior resilience and throughput, even under moderate fault pressure, suggesting an optimized distribution protocol. Go offers a balanced trade-off between stability and performance. Scala, while capable of high throughput, suffers from

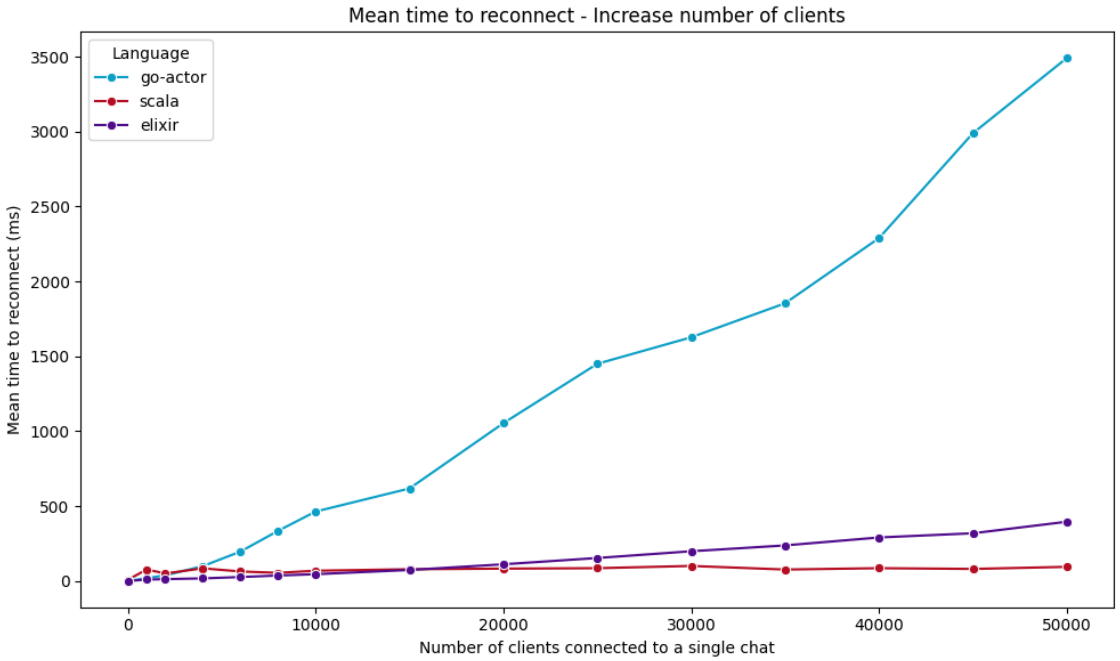

Figure 5.3: Mean time to reconnect after failure line graph over clients scalability

accentuated variability, being marked as less reliable for systems that demands predictable throughput stability.

5.2 Reconnection Time Test Results

Figures 5.3 and 5.4 illustrate the mean reconnection time of the clients on the three actor-based runtime implementations under two distinct scaling conditions. This analysis focuses on the recovery behavior of each system in response to transient fault injections. The measured reconnection time covers the duration, after a crash, for a chat instance to re-register with the discovery service and the subsequent time for its connected clients to reestablish their connections. In both configurations, the X-axis represents the system's scale, while the Y-axis indicates the average time in milliseconds required for each client to reestablish a connection after a fixed 5-second fault injection.

The benchmark is designed around two key scaling strategies. The first, illustrated in Figure 5.3, increases the number of clients connected to a single chat instance, ranging from 5 up to 50,000. The second, shown in Figure 5.4, increases the number of independent chat instances, each with 5 clients connected, scaling from 1 to 9,000 chats, trying to maintain a roughly comparable concurrency ratio in both scenarios.

Across both experiments, Go with Proto.Actor demonstrates the higher increase in reconnection time as system load scales. Initially, up to approximately 5,000 clients or 500 chats, Go performs comparably to the other two implementations. However, as the number of concurrent actors increases, reconnection performance degrades significantly. This degradation can likely be attributed to the overhead inherent in managing a large number of clients connected over one or multiple gRPC bidirectional streams. In particular, in a multiple-chat configuration architecture where communication is divided through different independent

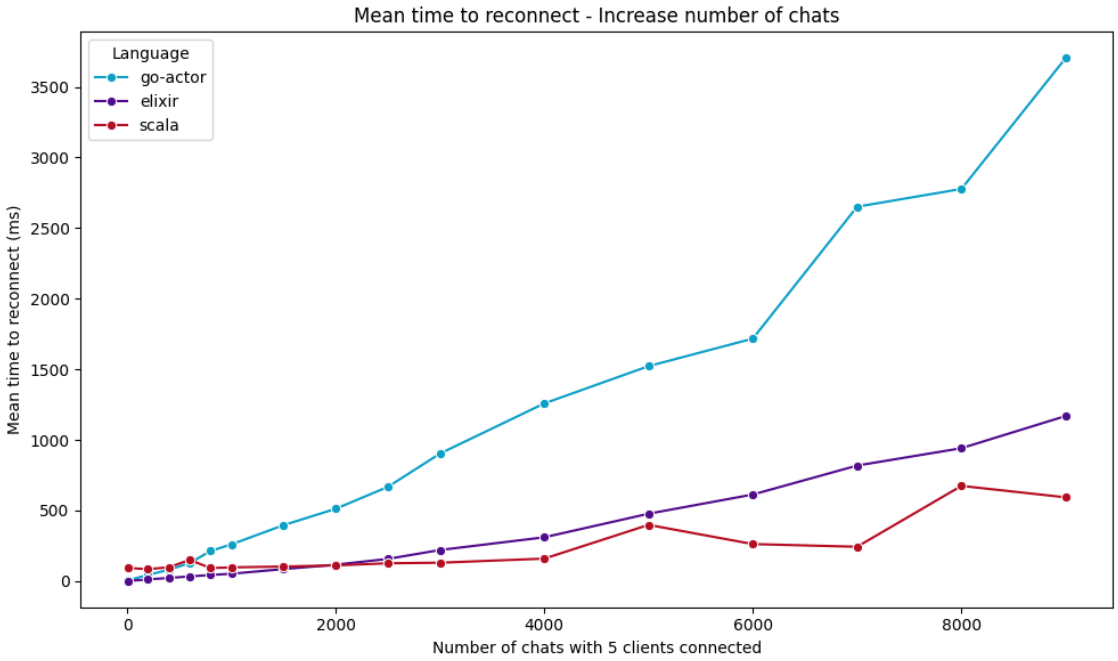

Figure 5.4: Mean time to reconnect after failure line graph over chats scalability

streams, it is plausible that the system suffers substantial costs due to context switching and memory consumption associated with multiple streams.

By contrast, when scaling clients on a single chat in Go, the system benefits slightly from the reuse of gRPC stream contexts, avoiding the overhead of managing thousands of parallel gRPC sessions on the same machine. This results in slightly better scalability, though not enough to compete with the reconnection times observed in Elixir and Scala. However, the reconnection time for Go in the single-chat case is ∼5% lower than the equivalent workload in the multi-chat scenario at peak concurrency, approximately at the 50,000 processes mark. This suggests that Proto.Actor's communication model may benefit from architectural patterns that consolidate sessions and reduce multiple stream connection endpoints.

Elixir and Scala exhibit more favorable and stable reconnection profiles. Scala with Akka displays particularly consistent performance, maintaining reconnection times close to ∼100 ms on the single-chat architecture, and not exceeding ∼700 ms at its highest mark on the multi-chat architecture, making it approximately ∼80% faster than Go at the same load. Elixir, while showing slightly more fluctuation at high scale, performs very well at lower concurrency levels, consistently outperforming Scala up to the 15,000 clients mark in Figure 5.3, and up to approximately 2,000 concurrent chats in Figure 5.4. Elixir is ∼70% faster than Go at the highest concurrency mark, indicating a comparatively smaller advantage against Go than Scala demonstrates.

After approximately 15,000 clients and 2,000 concurrent chats, Scala begins to outperform Elixir, maintaining a more controlled reconnection time curve. Elixir, conversely, exhibits a slightly more accentuated increase in reconnection time, particularly in the multi-chat scenario. This performance difference may be attributed to architectural choices in the application design, which do not fully leverage the BEAM's lightweight process capabilities. Specifically, in this case, the use of a centralized discovery server introduces significant

stateful overhead during the registration and deregistration of chat processes, such as the management of multiple termination signals on the same actor instance. Additionally, the contention caused by a large number of clients concurrently querying the discovery server may limit the BEAM's concurrency model in comparison to Akka's approach. Although the BEAM VM is known for its efficient lightweight process management and concurrent message passing, its reliance on the `:global` module for the discovery server PID could introduce some delay. This contention in global lookup operations, coupled with the high responsibility placed on a single actor, can harm reconnection responsiveness, particularly under high fault pressure. A study by Trinder et al. [29] observed that the `:global` module negatively impacts Erlang's scalability in a multi-node scenario, where a global lock is necessary. In this test case scenario, a single-node architecture was employed, and it was verified that some also delay occurred, which can be partially related to the study analyzed.

Scala's Akka runtime, while potentially slower in actor instantiation as noted by Trinder et al. [29], benefits from architectural features such as the receptionist pattern, which decouples service discovery from actor addressing. Combined with the optimizations provided by the JVM, such as mature garbage collection strategies, for examine, Akka benefits from a runtime environment where threads are generally more capable of handling heavy workloads without needing to be lightweight. This contributes to its enhanced stability under high load, allowing it to manage high-responsibility actors more robustly. Consequently, Akka's reconnection times tend to remain more predictable and less affected by the overall scale of the system.

Overall, this test scenario demonstrates that while Go with Proto.Actor can scale under moderate loads, its reconnection performance degrades rapidly as the number of actors and gRPC streams increases, revealing limitations in handling high-concurrency fault recovery scenarios. Scala's Akka runtime proves to be the most robust at large scale, consistently maintaining low reconnection times and effectively managing complex actor state, particularly as seen in the discovery server. Elixir, on the other hand, achieves faster reconnection times when operating with lightweight, low-state actors, conditions that highlight its runtime model's strengths. This advantage becomes especially clear under low actor concurrency, where its design aligns closely with its strengths, often outperforming the other two implementations.

5.3 Detection Time Test Results

Figures 5.5 and 5.6 illustrate the mean fault detection time observed in the three actor-based environments. Detection time is defined as the interval between the moment a chat process crashes and the moment each connected client is notified of the failure.

The two experimental setups reuse the configuration employed for reconnection time measurements. In Figure 5.5, a single chat process is shared by an increasing number of clients, ranging from 5 up to 50,000. In Figure 5.6, the system is scaled horizontally by launching multiple independent chat processes, each handling 5 clients, and scaling from 1 to 9,000 chat instances.

In the single-chat scenario, Figure 5.5, Scala with Akka demonstrates robust performance, with detection times remaining consistently low across the entire scale. Starting at approximately $\sim$0 ms for small client groups, detection time increases linearly but modestly, staying below 50 ms even with 50,000 clients. This behavior suggests that Akka's failure detection

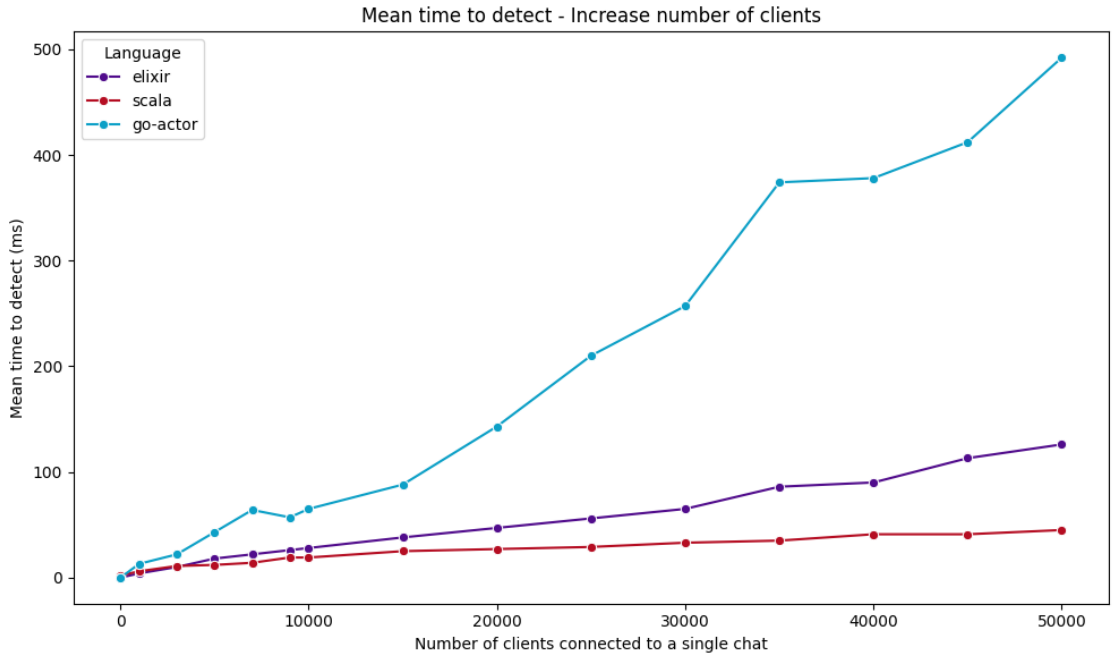

Figure 5.5: Mean time to detect failures line graph over clients scalability

mechanisms are stable under an architecture that centralizes the chat process connected with multiple clients.

Elixir, on the other hand, performs competitively at smaller scales but begins to exhibit higher detection times at larger client counts. Detection starts at around ∼0 ms, remaining relatively flat until approximately ∼10,000 clients. Beyond this point, the time increases progressively, reaching approximately ∼130 ms at 50,000 clients. This trend may reflect growing overhead in the BEAM VM's process monitoring mechanisms under high concurrency, although can keep a considerable low delay.

Go with Proto.Actor demonstrates reasonable performance at smaller scales, exhibiting near-zero latency for client numbers below 1,000. However, as the system scales, particularly approaching 50,000 clients, its detection time degrades sharply, reaching nearly ∼500 ms. This represents a substantial difference, making it approximately ∼75% slower than Elixir and ∼90% slower than Scala at peak load. This observed increase in detection time appears consistent with the inherent costs associated with managing a large volume of concurrent gRPC streams. As Proto.Actor does not provide a native protocol for inter-actor communication, all communication, including monitoring signals, relies on gRPC, consequently obligating the serialization and deserialization process of Protocol Buffers.

Figure 5.6, which evaluates horizontal scalability by increasing the number of chat actors while maintaining five clients per actor, reveals similar behavior on Go and even more performant on Elixir and Scala. Scala with Akka consistently maintains low detection latency, rounding to ∼0 ms across the entire scale. This indicates that the runtime handles supervision and failure notification more efficiently when the number of monitors is divided. Similarly, Elixir also performs more reliably in this distributed topology. Detection latency begins near ∼0 ms, remaining on a range of ∼10 ms to ∼40 ms throughout the test. Both results suggest that monitoring signals perform more effectively when distributed among multiple distinct actors compared with relying on a single chat instance.

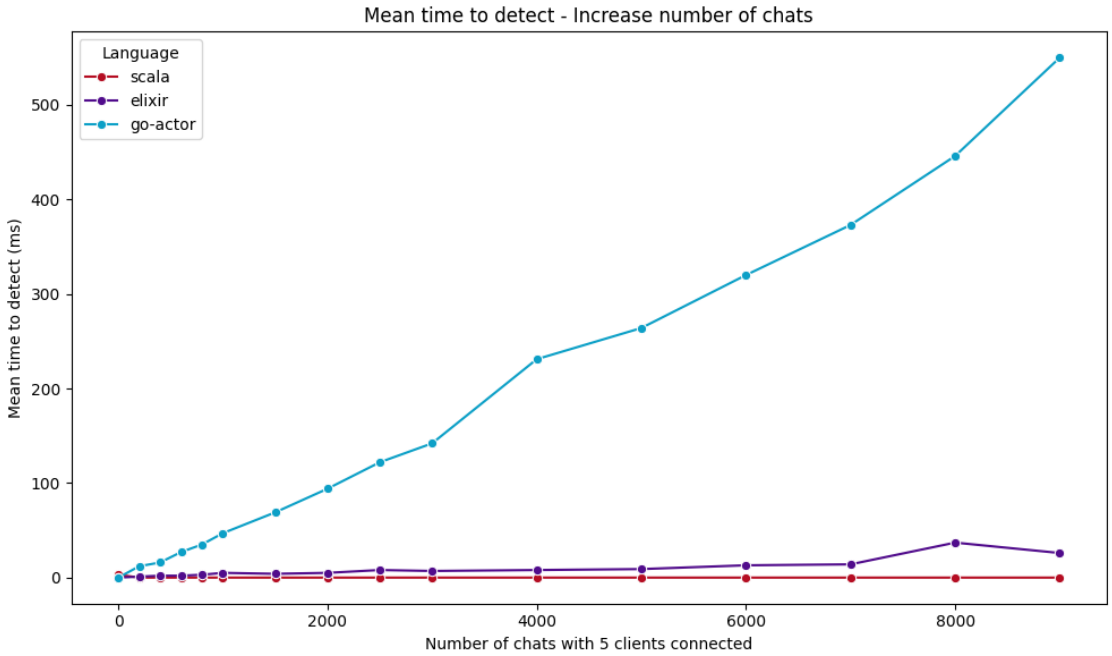

Figure 5.6: Mean time to detect failures line graph over chats scalability

Go with Proto.Actor exhibits a consistent detection time pattern across both architectural choices, likely due to the persistent and constant overhead of remote communication, regardless of whether clients are connected to a centralized or multiple chat instances. Nevertheless, a slight increase in overhead is observed when a multi-chat architecture is introduced, resulting in slightly higher values compared with the same concurrency ratio in the single-chat scenario.

Overall, Elixir and Scala demonstrate strong performance when the system is partitioned by chat actors, likely benefiting from more distributed supervision and monitoring structures. In contrast, Go exhibits consistent behavior across both configurations. These findings suggest that Scala and Elixir are better suited for distributed multi-chat architectures, as well as single-chat setups, whereas Go is more appropriate for simpler and very low concurrency scenarios, though its detection time consistently remains higher than the other two implementations.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

First, it was concluded that distributed systems are an essential part of modern global technologies. However, faults are an inherent aspect that must be taken into consideration, making fault tolerance strategies crucial in software design. Some mechanisms of fault tolerance were studied, including retries, the circuit breaker pattern, replication and redundancy with consensus algorithms, and paradigms like the Actor Model. Of all strategies, the Actor Model was detailed in depth in conjunction with Erlang principles, because it establishes a set of rules and terms to guarantee isolation in terms of concurrency, and, combined with its supervision tree, it enforces fault tolerance aspects.

As a result, and aligning with the motivation for this study, the scope of this document it was focused on Elixir. It was observed that Elixir, leveraging the BEAM VM, stands at the forefront in supporting fault tolerance capabilities. Inherently, Elixir, through the BEAM, is classified as state-of-the-art in concurrency models, having natively adopted the Actor Model. This model, with its "let it crash" philosophy, introduces fault-tolerant capabilities into the system. Furthermore, Elixir, within the OTP environment, provides a powerful set of tools to produce fault-tolerant systems.

The success of this ecosystem has inspired other languages to replicate the concepts of location transparency, actor isolation, and the fault tolerance aspects of the Actor Model. Among the most successful of these replications are Scala with the Akka framework and Go with Proto.Actor. Akka was found to integrate the maturity of the JVM environment, including the possibility to use one of the state-of-the-art garbage collectors like the Z Garbage Collector. Additionally, it was found to come with numerous built-in features, such as Akka Clustering and Akka circuit breaker, among other functionalities.

Go, a popular programming language for cloud and distributed systems, leverages the Actor Model by providing a framework called Proto.Actor. This framework blends the CSP model with the Actor Model, merging both paradigms into one. This combination unites the fast execution of goroutines with the abstraction of the Actor Model, creating an alternative advantageous for exploration within this document. Similar to Akka, this framework builds an abstraction on top of the native language that leverages the HTTP/2 protocol with gRPC to achieve location transparency and remote communication. Moreover, it introduces the concept of an "Actor Standard Protocol" to enable language-agnostic feature implementations.

Furthermore, in order to conduct an empirical benchmark, a preliminary analysis of existing studies by various authors was performed. These studies typically employed micro-benchmarking strategies, often drawing from methodologies like the Computer Language Benchmarks Game and Savina, focusing on small, specific benchmarks to test concurrency and parallelism. Consequently, a direct comparative analysis between these languages, particularly concerning fault tolerance, has been lacking. Existing research has either focused solely on performance without incorporating fault tolerance, or has excluded some of the languages addressed in this study.

To address this gap, this work proposed an extensible and configurable architectural test framework, mimicking a chat application, featuring distributed and fault-tolerant capabilities. The proposed architecture allows for testing various fault types, different supervisor-to-supervisee ratios, and the ability to employ different message types to test CPU or RAM stress. It also incorporates a discovery server responsible for mapping clients to servers in a distributed manner. Moreover, the design enforces a separation of metric calculation from the core test logic, supporting both performance and fault tolerance evaluations. This framework emphasizes configurability as a crucial aspect, along with asynchronous communication via a message queue to offload statistical computation.

However, due to resource and time limitations, the concrete architecture implemented was ultimately focused on a single node with a fault injection trigger. Consequently, this study presented a comparative evaluation of three actor-based runtime systems: Elixir, Scala with Akka, and Go with Proto.Actor. The benchmarks were designed to assess their behavior in fault-prone and highly concurrent distributed environments. The evaluation focused on three main dimensions: throughput under failure, reconnection latency, and fault detection time, all under different fault conditions and scaling configurations.

This comparative analysis of Elixir, Scala with Akka, and Go with Proto.Actor under fault-prone reveals a differentiated results panorama, where each runtime exhibits distinct strengths and weaknesses depending on the metric and system scale. Elixir demonstrated the most robust throughput across varying fault injection intervals, especially under moderate to low fault conditions, maintaining both high message delivery rates, peaking at ∼30,000 messages per second under no-fault, and with low variance. However, this comes at the cost of reduced scalability in reconnection and detection latency at larger scales, likely due to the lookup mechanism in the `:global` process and the centralized discovery server's state overhead. For example, at the higher concurrency peak, approximately ∼50,000 concurrent processes, Elixir's reconnection time increased from near ∼0 ms at low concurrency to ∼400 ms on a single-chat architecture and ∼1200 ms on a multi-chat architecture. This plausibly suggests that Elixir exhibits better recovery time when less work is required from the discovery service, rendering it a more lightweight actor, which aligns with the BEAM philosophy. In terms of detection time, Elixir emerged as a fast detection language. With approximately ∼50,000 processes, it demonstrated an average detection time of ∼130 ms on a single-chat architecture. However, this value remained within a range of ∼10 ms to ∼40 ms on a multi-chat architecture. Nevertheless, concerning lightweight processes, Elixir consistently performed well at low to medium scale across all tests, maintaining this performance benchmark.

Scala with Akka emerged as the most stable and scalable platform for reconnection and fault detection latency at high process counts. It consistently maintained near-zero detection times and less than ∼700 ms reconnection delays, even with up to 50,000 clients and 9,000 chat actors. This resilience might be attributable to Akka's receptionist pattern and mature JVM optimizations. This also suggests that Scala with Akka exhibits consistent performance

regardless of the underlying client-server architecture, implying that Akka is optimized for a consistent handling of both low and high numbers of lightweight processes and stateful dimensions. However, this stability incurs a throughput penalty: Scala with Akka produced lower and more inconsistent message rates under frequent transient failures, for instance, at a 300 ms interval. This is likely due to overhead from the JVM, internal message routing, and serialization within the Artery protocol, factors which may have contributed to the observed high variation across all throughput testing. However, further investigation is needed to confirm these causal relationships.

Go with Proto.Actor displayed competitive throughput at low fault frequencies and performed adequately under moderate load, but its architecture exhibited clear limitations at scale. In both reconnection and detection latency, performance degraded rapidly as the number of clients or chat actors increased: reconnection rose from ∼150 ms at 5,000 clients to over ∼3,500 ms at 50,000 clients, and detection from nearly ∼0 ms to nearly ∼500 ms in both cases. This highlights the overhead of managing large volumes of gRPC bidirectional streams and context switches. As a result, Go with Proto.Actor is less suitable for highly dynamic or large-scale fault-tolerant systems, though it remains acceptable in smaller or more controlled deployments where gRPC's ecosystem advantages, such as tooling and language support, are desirable. Additionally, it is concluded that Go with Proto.Actor presents better results on a single-chat architecture, which could be related to the facility of managing a single gRPC stream compared with the overhead of managing multiple separated streams.

It was concluded that, for the throughput test with 256 chats and 5 clients each under varying fault rates:

- Elixir and Go with Proto.Actor outperform Scala with Akka by ∼30% at a 300 ms fault interval and by ∼10% under no-fault conditions.
- Go with Proto.Actor can match Elixir's throughput above 3,000 ms fault intervals, though with greater variance during intermediate rates.
- Scala with Akka maintains consistent average messages per second but exhibits higher variability compared to the more stable profiles of Elixir and Go with Proto.Actor.

On the reconnection time test with a fixed 5-second fault injection, comparing a single-chat (up to 50,000 clients) and a multi-chat setup (up to 9,000 chats × 5 clients):

- Go with Proto.Actor achieved ∼5% faster reconnection time in the single-chat architecture versus the multi-chat configuration at peak load, demonstrating the benefit of stream reuse.
- Elixir outperforms Scala with Akka at lower process counts, up to 15,000 clients or 2,000 chats, indicating that centralized actors with state, like the discovery server, can become a bottleneck for Elixir as scale increases.
- Both Scala with Akka and Elixir can outperform Go with Proto.Actor at high peak rates by being approximately ∼80% and ∼70% faster on the reconnection process, respectively.

On the detection time test with a fixed 5-second fault injection, comparing a single-chat (up to 50,000 clients) and a multi-chat setup (up to 9,000 chats × 5 clients):

- Elixir and Scala with Akka maintain low, consistent detection latency when scaling chats up to 9,000 actors, with means on a range of ∼10 ms to ∼40 ms and near 0 ms

respectively. On the other side, in a single-chat scenario, detection rises modestly to ∼130 ms for Elixir and ∼50 ms for Scala at 50,000 clients.

- Go with Proto.Actor becomes approximately ∼75% slower than Elixir and approximately ∼90% slower than Scala at high peak load in the multi-chat setup concerning detection time.
- Scala with Akka and Elixir demonstrate superior capabilities in detection time within a multi-chat architecture compared with the single-chat architecture.

From a testing perspective, Elixir and Scala demonstrated consistently superior results across the various scenarios, particularly in terms of fault detection and recovery performance. These outcomes reflect the high degree of optimization inherent in their runtime environments, namely the BEAM and Akka, where all components operate within the same language and ecosystem. However, this coupling can present limitations, for example, in a microservices architecture, where heterogeneity and language diversity are often required. In contrast, although Go with Proto.Actor did not achieve the same level of performance, but it introduces a notably innovative architectural proposition through its agnostic Actor Model. By promoting language-agnostic communication via the Actor Standard Protocol, Proto.Actor enables actor-based systems to be built across diverse services and technologies. This is a flexibility that Elixir and Scala cannot easily provide. This makes Proto.Actor a valid option in polyglot microservice environments, despite its performance trade-offs.

In summary, Elixir offers excellent message throughput and low-variance performance for medium-sized, fault-tolerant applications. Additionally, it is the predominant language in architectures that favor low-responsibility, lightweight actors. Scala with Akka, in turn, prioritizes predictable recovery and minimal fault-handling latency, making it ideal for large-scale distributed systems where consistency and stability are required on architectures that imply actors with some responsibility and state, though with a throughput trade-off compared to Go and Elixir. Go with Proto.Actor performs well in smaller scenarios requiring few gRPC streams on a single node, but necessitates architectural refinement to address its sensitivity to scale and fault conditions. Also, Go brings advantages to a microservice architecture due to the language-agnostic protocol. Ultimately, system architects must balance throughput needs, recovery guarantees, and operational scale when selecting an actor runtime, as no single implementation excels across all dimensions.

6.2 Future Work

While the benchmarking framework presented in this study provides useful insight into actor-based runtime performance under fault conditions, several directions remain open for future exploration and enhancement, with the goal of expanding the literature encompassing fault tolerance benchmarking.

Multi-Node Testing This represents a critical direction for future work, as Elixir with the BEAM VM is inherently designed for distributed computing. However, the benchmarks in this study were limited to a single-node setup. Extending the tests to support multi-node deployments would allow for evaluating system behavior under more realistic conditions, including network partitions, inter-node latency, and distributed fault recovery. This would provide insights into how synchronization strategies in each runtime, such as global state consistency, supervision, and actor lookup, impact the speed and stability of recovery mechanisms.

For instance, Elixir's use of `:global` for process registration can degrade scalability due to coordination, and similar stress scenarios could test whether comparable bottlenecks exist in Scala with Akka, particularly when utilizing Akka Cluster features. Go with Proto.Actor, which relies on gRPC streams, may also behave differently under network partitions.

Stateful Actor Behavior Under Fault The existing tests focus on actors that have state, such as chat actors whose state increases with the number of clients. However, these actors do not currently retain their state upon restart. Introducing scenarios where actors must recover their state after failure could provide a more comprehensive and realistic evaluation of fault tolerance. This could involve testing snapshotting or event sourcing mechanisms and exploring auxiliary tools, such as Mnesia, for example, in comparison with other distributed storage mechanisms, to assess each runtime's ability to persist and restore state under fault conditions.

Memory and CPU Profiling Under Stress A future direction could involve profiling CPU and memory usage during fault injection and recovery phases. This could help analyze how efficiently each runtime handles resource contention under stress. It could expose potential bottlenecks in garbage collection, scheduling, or message handling.

Message Semantics and Delivery Guarantees The framework could be extended to evaluate delivery guarantees such as at-least-once and exactly-once semantics. Testing message ordering and duplication under failure scenarios and network stress.

Comparison with Additional Actor Frameworks Incorporating additional runtimes, such as Rust (Actix[1]) and Python (Thespian[2]) actor libraries, for example, could expand the comparative landscape and offer broader insight into language-level versus framework-level performance characteristics.

Comparison with General Fault-Tolerant Strategies As observed, strategies such as retries, circuit breakers, and checkpoints are often implemented relying on third-party libraries. A valuable direction for future work could be to evaluate how these mechanisms perform across different runtime implementations, enabling a comparative analysis of their effectiveness and overhead in each context. This would also facilitate identifying areas for improvement and potential optimizations within existing community libraries.

Leader Election and Discovery Overhead Another relevant direction for future work could involve evaluating the cost and impact of consensus algorithms, such as Raft, particularly during leader election scenarios. This is especially applicable to systems that require strong coordination and consistency guarantees. A potential experiment could measure the time required for a new leader to be elected and for clients to rediscover the leader following a crash across different runtimes.

Scheduler-Aware and Sticky Operations A direction for future work could be to examine how runtime schedulers influence fault-tolerant behavior, particularly under conditions where actors exhibit sticky execution patterns.

[1] Actix: `https://github.com/actix/actix/` (accessed 29 June 2025)
[2] Thespian: `https://github.com/kquick/Thespian/` (accessed 29 June 2025)

References

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

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

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

[4] Go. *Official documentation of Go programming language*. Accessed at 29.06.2025. url: https://go.dev/doc//.

[5] Ivan Valkov, Natalia Chechina, and Phil Trinder. "Comparing languages for engineering server software: Erlang, go, and scala with akka". In: *Proceedings of the ACM Symposium on Applied Computing*. Association for Computing Machinery, Apr. 2018, pp. 218–225. isbn: 9781450351911. doi: 10.1145/3167132.3167144.

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

[7] IEEE. *IEEE Code of Ethics*. Accessed at 29.06.2025. url: https://www.ieee.org/about/corporate/governance/p7-8.html.

[8] ACM. *ACM Code of Ethics and Professional Conduct*. Accessed at 29.06.2025. url: https://www.acm.org/code-of-ethics/.

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

[10] 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.

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

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

[13] 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.

[14] Atlassian. *Reliability vs availability: Understanding the differences*. Accessed at 29.06.2025. url: https://www.atlassian.com/incident-management/kpis/reliability-vs-availability.

[15] 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.

[16] 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.

[17] AWS - Amazon. *Challenges with distributed systems*. Accessed at 29.06.2025. url: https://aws.amazon.com/builders-library/challenges-with-distributed-systems/.

[18] IBM. *What is the CAP theorem?* Accessed at 29.06.2025. Aug. 2024. url: https://www.ibm.com/topics/cap-theorem.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] *Apache Flink Documentation*. Accessed at 29.06.2025. url: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/fault-tolerance/checkpointing/.

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[39] *Elixir Language Documentation*. Accessed at 29.06.2025. url: https://hexdocs.pm/elixir/introduction.html.

[40] Attila Sragli. *Optimising for Concurrency: Comparing and contrasting the BEAM and JVM virtual machines*. Accessed at 29.06.2025. Nov. 2024. url: https://www.erlang-solutions.com/blog/optimising-for-concurrency-comparing-and-contrasting-the-beam-and-jvm-virtual-machines/.

[41] *Akka Official Documentation*. Accessed at 29.06.2025. 2024. url: https://doc.akka.io/libraries/akka-core/current/typed/.

[42] *Proto.Actor Documentation*. Accessed at 29.06.2025. 2024. url: https://proto.actor/docs/.

[43] *Elixir School - Official*. Accessed at 29.06.2025. 2024. url: https://elixirschool.com/.

[44] 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.

[45] 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.

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

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

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

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

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

[50] Brian Kennedy. *Go in Action*. Manning, 2016. isbn: 1617291781.

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

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

[53] Alexander. Shuiskov. *Microservices With GO : Building Scalable and Reliable Go Microservices*. Packt Publishing Limited, 2022. isbn: 9781804617007.

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

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

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

[57] *Go-kit Documentation*. Accessed at 29.06.2025. 2024. url: https://gokit.io/.

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

[59] *Discord blog - Why discord is switching from go to rust*. Accessed at 29.06.2025. 2024. url: https://discord.com/blog/why-discord-is-switching-from-go-to-rust/.

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

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

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

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

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