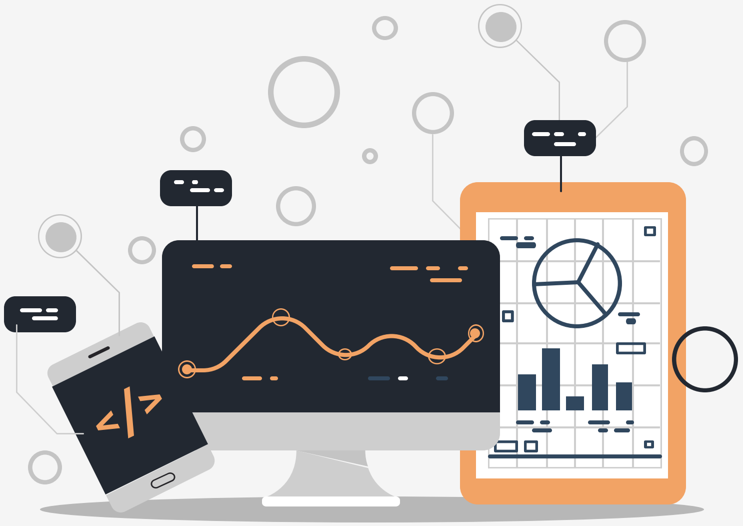
Comparative Analysis of Fault Tolerance in Elixir and Other Distributed and Concurrent Programming Languages



Dissertation Preparation

Nuno Ribeiro 1230201

Context and Problem

- **Rapidly evolving** landscape of **software development** makes **fault tolerance** and resilience to become a **critical attributes** for building robust and scalable systems.
- **Elixir** stands as an **important language** in building **fault-tolerant software**, and it is a popular and reference language on building this type of systems due to the **inheritance capabilities of Erlang and the BEAM.**
- **Lack of comprehensive,** up-to-date **research that directly compares** the fault tolerance and resilience aspects of Elixir with other programming languages motivate this dissertation.

Objectives

01

Comprehensively analyze the fault-tolerant mechanisms in Elixir.

02

Identify the most popular and relevant distributed and concurrent languages and their fault-tolerant mechanisms.

03

Compare Elixir's fault-tolerant capabilities with those languages about strengths, weaknesses, and trade-offs.

04

Conduct benchmarking experiments to empirically evaluate and compare the fault tolerance and resilience of Elixir and other languages.

05

Extract best practices and propose potential improvements in fault-tolerant system design.

Distributed and Concurrency Programming Languages

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

Research Questions

01 Research Question 1

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

02 Research Question 2

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

Elixir, Scala with Akka and Go

Elixir

- Built for fault tolerance and concurrency.
- “*Let it crash*” philosophy with process supervision.
- Mature resources (OTP) for fault recovery.
- Lightweight processes with preemptive scheduling.
- Hot-code swapping for live updates.

Scala with Akka

- Extends JVM with actor-based concurrency.
- Tools: Clustering, Persistence, Circuit Breakers.
- Requires configuration, runs atop JVM.
- Mutable constructs introduce risks.
- Presumably, higher overhead than BEAM on some cases.

Elixir, Scala with Akka and Go

Go

- Explicit error handling over "*let it crash*" philosophy.
- Concurrency via goroutines and channels.
- Libraries (Proto-Actor) merge CSP with actors.
- Cooperative scheduler that could lead to a CPU monopolization.
- Lacks native fault tolerance and distribution integration but it has mature libraries.

Comparison Points

- Preemptive Schedule vs Cooperative Schedule.
- How each languages handles distribution.
- Tools and support.
- Garbage collector strategies.
- Error handling philosophy.

Literature Review of Benchmarking Strategies

- **Benchmarking fault tolerance** in distributed environments **is challenging**.
- **No formal standard** exists for benchmarking fault tolerance, complicating comparisons.
- **Most benchmarks** evaluate both **fault tolerance** and **performance metrics** simultaneously.
- **Chaos engineering** tries to mimic **randomness real-world faults**, while **deterministic** error injection offers **greater reproducibility**.
- A **hybrid approach** combining **generic application** simulations with **deterministic error injection** offers a balanced perspective.
- **Effective strategies** should integrate **performance monitoring**, **resilience assessment**, and **static code analysis** for a holistic approach to fault tolerance.

Future Work

Purpose

- **Development** of a **chat application** to evaluate fault tolerance and resilience in distributed systems.
- Exploration of how **Elixir, Scala with Akka**, and **Go handle faults**.
- **Simulate distributed communication** within a fixed setup using **inter-VM communication**.
- Two implementations for Go: Proto-Actor library and a native model using goroutines and gRPC.

System Elements

- **Clients**: Send/receive messages, manage connections, simulate failures.
- **Discovery**: Registry for client locations.
- **Chats**: Manage group conversations.
- **External Injector**: Orchestrate faults and activities.
- **External Logger**: Log events and metrics for analysis.

Future Work

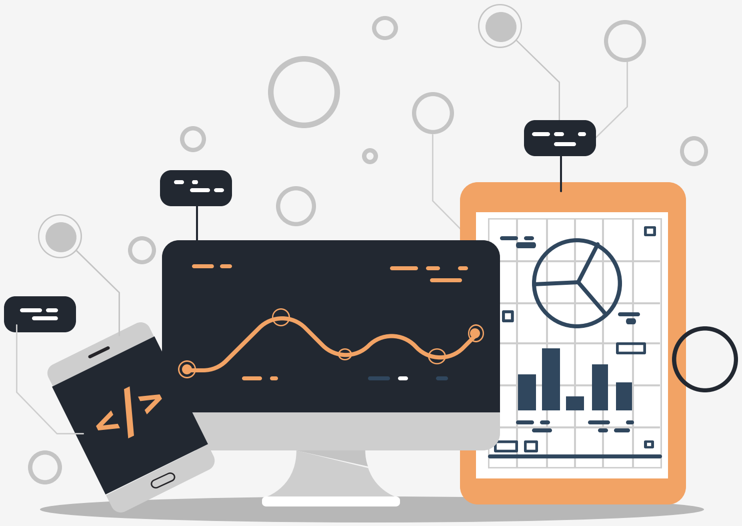
Configurations

- Adjustable **message characteristics** (size, type, algorithm).
- Simulate failures (crashes, cascading failures).
- Control **client behavior** (activity frequency, activity action).
- Test supervision strategies and connectivity parameters.

Test Scenarios

1. **Client Crash Recovery:** Measure recovery time and system state consistency.
2. **Chat Crash Recovery:** Evaluate replication and leader election (Raft).
3. **Message Delivery Durability:** Test message flow consistency under stress.
4. **Network Test:** Simulate network partitions and latency spikes.

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



Dissertation Preparation

Nuno Ribeiro 1230201