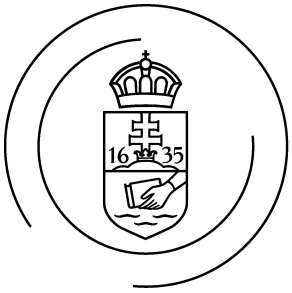


Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Languages and Compilers

Comparative Analysis of Energy Efficiency in Actor-Based Applications in Distributed Environments

Supervisor:
Dr. Melinda Tóth, Gharbi Youssef
Associate professor (PhD), PhD student

Author:
Botond Szirtes
Computer Science MSc

Budapest, 2025

Contents

Chapter 1

Introduction

As the energy footprint of digital systems grows, the need for efficient, sustainable software design becomes increasingly critical. While recent research has explored energy consumption across various programming languages and paradigms, much of this work has concentrated on sequential or isolated scenarios, lacking emphasis on concurrent, and distributed execution contexts. Actor-based programming models, exemplified by languages and frameworks such as Erlang [1] and the C++ Actor Framework (CAF) [2, 3], provide a natural fit for building scalable, fault-tolerant distributed applications. However, the energy implications of these models, especially in large-scale deployments, remain relatively underexplored.

This thesis investigates the energy efficiency of actor-based applications in both native and distributed environments, focusing on a comparative study between Erlang and CAF. By designing a controlled measurement framework, implementing a consistent set of benchmarks, and performing extensive measurements, the research aims to uncover how language design choices and deployment strategies influence energy consumption.

The results are organized around two major execution environments: native single-node deployments and distributed multi-node systems. Multiple benchmark patterns were evaluated, progressively increasing in communication and computational complexity, providing a holistic view of how these technologies behave under different workloads. Beyond runtime performance, the study places a strong emphasis on energy efficiency, recognizing it as a critical metric for modern, sustainable computing.

The rest of the thesis is structured as follows:

- Chapter 2 presents the background of green computing, actor models, and related work in energy-aware programming.
- Chapter 3 describes the methodology, measurement setup, and post-processing tools.
- Chapter 4 details the design and implementation of the benchmark applications.
- Chapter 5 analyzes the collected results, providing both quantitative and comparative insights.
- Chapter 6 discusses related work in energy efficiency across programming languages and distributed systems.
- Chapter 7 concludes the thesis with key findings, implications, and directions for future research.

Chapter 2

Background

This chapter provides an overview of the key concepts and technologies relevant to this thesis. It covers foundational ideas in green computing, summarizes previous work in energy-aware software analysis, and introduces the actor-based programming models and distributed deployment strategies used in the experiments.

2.1 Green computing and Programming Language Efficiency

Green computing refers to the practice of designing, using, and disposing of computing systems in ways that minimize their environmental impact. It involves reducing energy consumption, reducing greenhouse gas emissions, and promoting sustainable use of resources throughout the life cycle of computing hardware and software [4].

As the demand for computational power continues to increase—driven by data centers, cloud services, and always-on systems—the energy footprint of digital infrastructure has become a major concern. Green computing addresses these challenges by encouraging energy-efficient hardware designs, optimizing software to reduce power usage, and adopting environmentally friendly practices such as virtualization, recycling, and responsible e-waste management [4].

The field also emphasizes the role of software developers in minimizing energy usage through efficient algorithms, programming paradigms, and runtime behaviors. This makes energy-aware programming a growing area of interest, especially in large-

scale or always-active systems, where even small improvements can lead to significant savings [4].

The growing importance of sustainable software engineering has led to a significant interest in evaluating the energy efficiency of programming languages. Although software optimization has historically focused on execution speed and memory usage, energy consumption is now increasingly recognized as a key factor, especially in high-scale, battery-sensitive, or energy-constrained systems.

A large-scale study by Pereira et al. [5] conducted one of the most comprehensive comparisons of programming language efficiency to date. The research evaluated 27 popular programming languages using implementations of 10 standardized algorithmic problems from the *Computer Language Benchmarks Game* (CLBG). Each language was benchmarked in terms of energy consumption, execution time, and memory usage, producing both single-metric and multiobjective rankings.

One of the key insights from the study is that faster languages are not necessarily more energy efficient. Although there is often a strong correlation between execution time and energy consumption, the relationship is not linear or consistent across all languages. This is largely due to the varying power usage profiles, compiler optimizations, memory management strategies, and runtime overhead.

The study found that compiled languages generally outperform interpreted and virtual machine-based languages in terms of energy and time efficiency. Languages like C, Rust, and C++ ranked among the most energy-efficient and fastest across all benchmarks [5]. In contrast, languages with garbage collection or heavy runtime environments (e.g., Python, Perl, Erlang) often ranked near the bottom in both categories.

C++ vs Erlang

When focusing on the two languages relevant to this thesis—C++ (represented here via CAF, the C++ Actor Framework) and Erlang—the study provides valuable context. C++ was consistently ranked as one of the top three languages for both energy efficiency and speed. Its performance is attributed to being natively compiled and highly optimized, with minimal runtime overhead.

Erlang, by contrast, ranked significantly lower. As a virtual machine-based functional language designed for fault-tolerant distributed systems, Erlang exhib-

ited higher energy consumption and longer execution times across most benchmarks. These results suggest a trade-off between Erlang's reliability and concurrency benefits and its lower raw computational efficiency. However, it is important to note that Erlang's strength lies in distributed systems and soft real-time performance—scenarios not directly reflected in the CLBG's primarily algorithmic benchmarks.

This thesis builds upon these findings by comparing C++ (via CAF) and Erlang not only in isolated algorithmic scenarios but also in actor-based and distributed applications. By measuring real-world actor-based workloads, this research aims to contextualize the efficiency differences in environments more aligned with the languages' intended use cases.

2.2 Former Research at Eötvös Loránd University

Research conducted at Eötvös Loránd University (ELTE) has explored various aspects of energy-aware software engineering, particularly in the context of Erlang and functional programming. Several works have contributed to the development of tools, methodologies, and experimental insights related to energy efficiency in Erlang-based systems. These contributions provide valuable context and background for the current thesis.

Research on Green Computing in Erlang

Among the contributions to green computing within the Erlang ecosystem, several studies at Eötvös Loránd University (ELTE) have investigated how language constructs affect energy efficiency. One such study by Mészáros et al. explored the relationship between energy consumption and programming patterns in Erlang, examining how data structures, higher-order functions, and parallel constructs behave under energy profiling [6]. Using a measurement framework built on Intel's RAPL counters, the authors conducted experiments on computational tasks such as the N-Queens problem and sparse matrix multiplication to evaluate the energy implications of common functional idioms.

Building on this direction, another work by Nagy et al. presented an enhanced tooling infrastructure named GreenErl, developed to measure the energy usage of

Erlang programs in a more structured and extensible way [7]. The tool integrates a Python GUI for orchestrating experiments, Erlang modules for test execution, and a C backend for low-level energy measurements. The study went beyond analysis and introduced automated refactorings aimed at improving the energy behavior of Erlang code. These refactorings, incorporated into the RefactorErl framework, target common performance bottlenecks such as inefficient data structure use and higher-order function overhead.

Together, these works laid a strong foundation for continued investigations into the energy characteristics of Erlang, influencing subsequent tool development and research directions at ELTE.

Green Computing for Erlang

Later research conducted by Gharbi, Tóth, and Bozó at Eötvös Loránd University (ELTE) has focused extensively on the energy behavior of Erlang applications in the context of green computing [8]. Erlang, being widely used in high-availability server environments, presents significant potential for energy optimization. The research extended the aforementioned GreenErl framework, originally designed for Linux, which enabled precise measurement of the energy consumption of Erlang constructs and encouraged best practices for writing energy-efficient code.

This framework leveraged RAPL (Running Average Power Limit) counters to analyze the energy usage of various Erlang data structures and language elements, such as proplists, maps, dictionaries, and higher-order functions. The findings provided practical suggestions for refactoring Erlang code to reduce energy consumption, such as replacing higher-order functions with list comprehensions or optimizing how data structures are transmitted between processes.

Analyzing the Energy Usage of the Erlang BEAM

Building upon this work, in a later study, Gharbi, Tóth, and Bozó explored the effects of the Just-In-Time (JIT) compiler introduced in Erlang/OTP 24 and its influence on performance and energy usage [9]. The research introduced a cross-platform measurement framework capable of analyzing energy consumption on both Windows and Linux. To address the lack of RAPL support in Windows, the frame-

work integrated the Scaphandre tool, allowing accurate energy monitoring across platforms.

The study compared OTP versions 23, 26, and 27, measuring execution time, energy usage, and power draw for a series of benchmark applications and custom-written problems like the N-Queens problem. Results showed a significant reduction in energy consumption and runtime with the introduction of JIT in OTP 26. For instance, the n-body problem consumed 55% less energy in OTP 26 compared to OTP 23.

Measuring and Analyzing Erlang's Energy Usage

An additional investigation from Eötvös Loránd University (ELTE) presented by Gharbi, Tóth, and Bozó further extends the understanding of Erlang's energy characteristics [10]. This research re-evaluated the energy efficiency of Erlang by focusing on problem domains aligned with its original design principles, such as concurrency and networked server applications.

The study critiques earlier energy comparisons that positioned Erlang as an inefficient language, suggesting that the chosen benchmarks did not reflect Erlang's strengths. To address this, the researchers implemented server-client architectures and socket-based echo servers to measure energy usage and runtime performance more realistically. Additionally, the study ported its benchmarking framework to Windows, incorporating the Scaphandre energy measurement tool to support a cross-platform evaluation.

Results showed that when used in appropriate problem domains, Erlang's energy performance improved dramatically compared to earlier assumptions. For example, the energy consumption and execution time of an Erlang echo server were much closer to that of C and Java servers than previous studies indicated, especially when handling thousands of concurrent clients without artificial delays. These findings highlight the necessity of matching the benchmark tasks to the language's intended use cases when evaluating energy efficiency.

Scaphandre

To enable energy monitoring, ELTE researchers integrated Scaphandre—an open-source, cross-platform tool designed to monitor energy usage using RAPL

counters—into their GreenErl framework [11]. Scaphandre collects energy consumption data at the process level and supports high sampling precision, with output options such as JSON [12] or direct integration with monitoring dashboards [13]. Importantly, Scaphandre also supports container-aware energy monitoring [14]. It can attribute power consumption to specific containers in platforms like Docker and Kubernetes, enabling fine-grained energy analysis in containerized distributed environments.

It works by correlating total power consumption data (via RAPL) with per-process resource usage, allowing it to estimate the energy usage of each process [15]. In the extended GreenErl framework, Scaphandre was used to capture function-level energy metrics in Erlang applications. The measurement results were then parsed and visualized using Python scripts, enabling researchers to analyze the energy efficiency of individual functions and constructs.

Scaphandre proved especially effective for handling short-lived processes, which are common in Erlang applications, by offering nanosecond sampling precision and granular profiling. It provided consistent energy measurements across both Windows and Linux platforms, forming the basis of a reliable, cross-platform benchmarking framework for future research.

2.3 The Erlang Programming Language and the Actor Model

Originally developed at Ericsson in the 1980s, Erlang was designed to support distributed, fault-tolerant, soft real-time applications, especially in the telecommunications domain. Its core characteristics include functional programming, immutable data, and built-in support for concurrency and distribution [1].

Erlang is fundamentally based on the actor model of concurrency, where each actor is an isolated, lightweight process. These processes interact by sending and receiving messages asynchronously. Erlang's actor system is managed entirely by its virtual machine (VM), allowing for massive concurrency and fault isolation, without relying on OS threads [1].

Key Concepts within Erlang

- **Actors:** Each Erlang actor is a process with its own isolated memory and execution context. Processes are lightweight and created within the Erlang VM, making it feasible to run hundreds of thousands of them concurrently [1].
- **Messages:** Erlang processes communicate via asynchronous message passing. Any data structure in Erlang can be sent as a message, and messages are received from a process mailbox using pattern matching [1].
- **Behavior:** The behavior of an Erlang process is defined by recursive functions that wait for and pattern-match incoming messages. This allows processes to evolve dynamically by invoking new versions of themselves with different parameters [1].
- **Addresses:** Each process is uniquely identified by a process identifier (PID). Messages are sent to a process using its PID, enabling both local and remote communication [1].

Concurrency and Distribution in Erlang

Erlang provides robust support for both local concurrency and distributed computing.

Concurrency: Erlang provides a highly efficient concurrency model. Since processes do not share memory and communicate exclusively through message passing, issues like race conditions and deadlocks are naturally avoided. The VM handles process scheduling internally, resulting in microsecond-level process creation and message delivery times, even at scale [1].

Distribution: Distributed programming is deeply integrated into Erlang. Multiple Erlang nodes can form a cluster, with transparent message passing between processes on different nodes. The syntax and semantics for local and remote communication are identical, making it easy to scale applications across multiple machines. Distribution in Erlang is secured via cookie-based authentication and supports dynamic node discovery and clustering [1].

2.4 The C++ Actor Framework (CAF)

The C++ Actor Framework (CAF) is an open-source library that provides a comprehensive toolkit for building concurrent and distributed applications in modern C++. It implements the actor model, offering a high-level abstraction for managing communication, concurrency and distribution, aiming for scalability, robustness, and ease of development [16].

Key Concepts within CAF

- **Actors:** The fundamental building blocks of a CAF application. Each actor has a unique identity and state which can only be modified by sending messages to it. CAF supports both dynamically and statically typed actors. Dynamically typed actors, akin to those in Erlang, accept any message type and determine behavior at runtime, while statically typed actors define explicit messaging interfaces, enabling compile-time verification of communication protocols [17].

- **Messages:** Actors communicate by sending and receiving messages. In CAF, messages are immutable tuples composed of arbitrary, copyable C++ types. Messages can be delivered to both local and remote actors transparently, enabling seamless distribution. The framework serializes messages automatically when crossing process boundaries [18].

- **Behavior:** An actor's behavior defines how it responds to different types of messages. In CAF, this is expressed as a set of message handlers, often implemented using lambda expressions. These handlers map specific message patterns to corresponding actions [19].

- **Handles:** Each actor has a handle, that stores its unique logical address and type information. The handle of an actor serves as its reference and is used for message delivery. The distinction between handles and addresses, which is unique to CAF, when comparing to other actor systems, ensures type safety for messages [16].

Concurrency and Distribution in CAF

CAF provides strong support for both local concurrency and distributed computing.

Concurrency: Within a single process, CAF enables the creation and management of a large number of lightweight actors that can execute concurrently. The framework handles the underlying threading and synchronization mechanisms, allowing developers to focus on the actor-based logic of their application. The isolation of actor states inherently reduces the complexities associated with shared mutable state and the need for explicit locking [17].

Distribution: CAF facilitates the development of distributed applications by providing mechanisms for actors running on different nodes to communicate seamlessly via message passing. The framework handles the serialization of messages, as well as the underlying network communication. Features like actor proxies allow local actors to interact with remote actors as if they were local [20]. CAF supports various transport protocols for distributed communication.

2.5 Deploying CAF in a Distributed Environment

To deploy the distributed C++ Actor Framework (CAF) setup, this project utilizes Kubernetes as the orchestration platform, specifically through `KinD` (Kubernetes in Docker). While CAF itself provides actor distribution primitives, it does not enforce a deployment mechanism. Therefore, Kubernetes is used to manage actor-hosting containers, handle service discovery, and ensure scalability and reproducibility.

Kubernetes

Kubernetes is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. It provides a resilient infrastructure for running distributed systems by abstracting the complexities of underlying hardware and automating operational tasks [21].

Core Concepts

- **Pods**: The smallest deployable units in Kubernetes, a `Pod` encapsulates one or more containers that share storage, network, and runtime configuration. `Pods` are ephemeral and are typically managed by higher-level controllers such as `Deployments` or `StatefulSets` [22].
- **Nodes**: A `Node` is a worker machine in a Kubernetes cluster. Each `Node` runs a container runtime (e.g., `containerd`), the `kubelet` (agent), and the `kube-proxy`, which manages networking rules [23].
- **Control Plane**: The *control plane* is responsible for maintaining the desired state of the cluster. It includes core components such as the `kube-apiserver`, `controller-manager`, `scheduler`, and `etcd` [24].
- **Services**: A `Service` is an abstraction that defines a logical set of `Pods` and a policy for accessing them. `Services` enable load balancing and stable networking across transient or autoscaled `Pods` [25].

Declarative Configuration and Automation

Kubernetes follows a declarative configuration model. System state is described using configuration files written in YAML or JSON, which declare the desired status of resources. The `control plane` continuously works to match the actual state with the declared one, enabling automated deployment, scaling, and healing [21].

KinD

`KinD` (Kubernetes in Docker) is an open-source tool designed to run local Kubernetes clusters using Docker container "nodes". It was primarily developed for testing Kubernetes itself but is also suitable for local development and testing [26].

Architecture and Design

`KinD` creates Kubernetes clusters by launching Docker containers that act as cluster nodes. Each node is bootstrapped using `kubeadm`, ensuring a standard Kubernetes setup. The tool comprises:

- Go packages that implement cluster creation and image building.

- A command-line interface (`kind`) built on these packages.
- Docker images configured to run systemd and Kubernetes components.

This architecture allows for rapid provisioning of multi-node clusters, making it ideal for testing and development scenarios [26].

Configuration and Usage

`KinD` supports configuration through YAML files, enabling users to define cluster specifications such as the number of nodes, Kubernetes version, and networking settings. A minimal configuration file might look like [27]:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
- role: worker
```

Code 2.1: Example `KinD` config

To create a cluster with this configuration, the following command is used:

```
kind create cluster --config=config.yaml
```

This declarative approach fully aligns with Kubernetes conventions. Its ability to run entirely in Docker containers makes it platform-independent and easy to set up, providing a consistent environment across different development and testing stages [26].

In this thesis, `KinD` is chosen to simulate a multi-node Kubernetes cluster in a local environment and to make it possible to deploy and test distributed CAF applications in isolated containers that replicate real-world microservice conditions, without the need for a physical or cloud-based cluster. This strategy provides flexibility in managing the deployment, setting up different topologies, and conducting energy measurements in a controlled and reproducible distributed environment.

Chapter 3

Methodology and the Designed Framework

This chapter describes the experimental setup, tools, and procedures used to measure and analyze the energy efficiency of actor-based applications. It introduces the hardware and software environment, outlines the energy measurement framework, and details the specific steps taken to prepare native and distributed deployments. The methodology was designed to ensure reproducibility, fairness, and comparability across different programming languages and deployment architectures.

3.1 Hardware and Software Environment

All development, testing, and energy measurements for this thesis were conducted on a single, dedicated machine. The use of a consistent hardware and software configuration across all experiments ensures repeatability and minimizes external variability in performance and energy readings.

Hardware Specifications

The system used for all measurements was a `Dell XPS 15 9530` equipped with the following components:

- **Processor:** 13$^{\text{th}}$ Generation Intel® Core™ i9-13900H (24MB Cache, up to 5.4 GHz, 14 cores)

- **Graphics:** NVIDIA® GeForce RTX™ 4060 Laptop GPU with 8GB GDDR6 (40W)
- **Memory:** 32 GB DDR5 (2 x 16 GB, 4800 MHz, dual-channel)
- **Storage:** 1 TB M.2 PCIe NVMe Gen4 x4 SSD

Software Stack

All experiments were performed under the `Ubuntu 22.04 LTS` operating system. The following software tools and libraries were installed and used throughout the development and measurement phases:

- **Erlang/OTP:** Version 25
- **C++ Actor Framework (CAF):** Version 1.0.2
- **Python:** Version 3.10.12
- **Scaphandre:** Version 1.0.0
- **Docker:** Version 28.0.4
- **Kubernetes Client:** Version 1.29.2
- **Kubernetes Server:** Version 1.28.0
- **KinD (Kubernetes IN Docker):** Version 0.22.0

All measurements were performed locally on this system.

3.2 Energy Measurement Framework

To ensure reproducible and consistent energy measurements across different deployment modes, a custom Python script named `measure_linux.py` [1] was developed. The current implementation builds upon a measurement script [2] developed by Gharbi, Tóth, and Bozó in a prior research project at Eötvös Loránd University [9].

[1] `https://github.com/bszirtes/THESIS/blob/main/scripts/measure_linux.py`

[2] `https://github.com/joegharbi/SQAMIA_GREEN_JIT/blob/Linux/RosettaQueens/run_linux.py`

The script is designed to automate the execution, monitoring, and logging of actor-based programs deployed natively or in distributed setups, whether written in Erlang or C++ (CAF). It supports native and containerized environments and integrates directly with Scaphandre, the power telemetry agent, which is capable of process- and container-level energy accounting.

Supported Deployment Modes

The script supports four distinct configurations, reflecting the architectural scope of this research:

1. **Native Erlang:** Local execution of an Erlang module using the BEAM virtual machine (`beam.smp`).
2. **Native CAF:** Local execution of a compiled C++ program using the CAF framework.
3. **Distributed Erlang:** Multiple Erlang nodes deployed on a single machine, launched using `-sname` and a shared cookie for inter-node communication.
4. **Distributed CAF:** Containerized CAF actor systems orchestrated by `Kubernetes`, with energy profiling targeting specific container names via Scaphandre's container integration.

Execution and Measurement Workflow

The script performs the following steps in each run:

1. **Folder Setup:** Creates a dedicated folder for storing measurement logs if it does not already exist.
2. **Program Compilation:** For native deployments, the script compiles either `.erl` or `.cpp` source files at every initial run of a parameter from the set of provided parameters.
3. **Scaphandre Initialization:** Launches Scaphandre JSON Exporter with appropriate flags, targeting process names (e.g., `beam.smp`) or container names using `--container-regex`.

4. **Program Execution:** Runs the specified program and tracks its runtime using Python's `timeit` module.

5. **Data Collection:** After execution, the script reads the generated JSON file, filters it for relevant consumers (excluding Scaphandre itself and other unrelated shells) and summarizes the valid consumption entries.

6. **Energy Calculation:** Using the total consumption, the number of valid samples and the runtime, it calculates the average and the final energy usage.

7. **Output Logging:** Appends a new row to the CSV file of results with all the acquired data from the measurement.

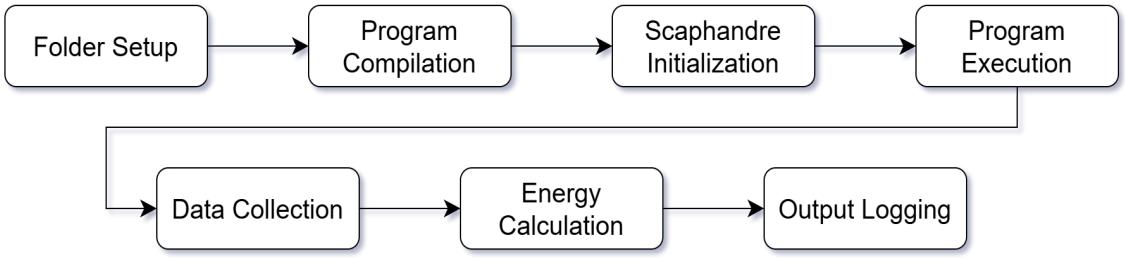


Figure 3.1: Steps of the Measurement Workflow

Energy Consumption Calculation

The script uses the following formula to calculate the energy consumption in microjoules (µJ):

$$E = \bar{P} \times t$$

where:

- $\bar{P}$ is the average power in microwatts (µW) calculated from the JSON samples,
- t is the execution time of the program in seconds.

The average power is calculated as:

$$\bar{P} = \frac{\sum_{i=1}^{N} P_i}{N}$$

where P_i are individual sample values and N is the number of valid nonzero samples.

Final energy is computed only if at least one valid sample exists; otherwise, a fallback value of 0 is recorded. All of the captured and calculated data, including the raw JSON measurement files, is saved per repetition, enabling traceability and statistical post-analysis.

Container-Aware Monitoring

For distributed CAF deployments under Kubernetes, the script uses the JSON Exporter's [12] `-containers` and `-container-regex` options in Scaphandre to filter energy data associated with the relevant CAF node containers deployed via `KinD`.

Output and Result Logging

All measurement outputs are logged into:

- A JSON file for each run, containing raw Scaphandre output.
- A CSV file summarizing the experiment metadata and results.

Each row in the CSV contains:

`[Module, Function, Parameter, Runtime (s), Samples, Average (µW), Consumption (µJ), Sampling interval (ns), File name]`

This dual-format approach enables reproducibility, traceability and statistical analysis in later phases of the thesis.

Script Configuration Parameters

The developed script allows for extensive control over the measurement process through a comprehensive set of configurable command-line arguments. This design enables users to tailor the measurement environment and behavior to specific needs, whether it involves native or containerized applications, repeated runs, or fine-grained power sampling. Notably, the script supports the provision of multiple parameter combinations for the measured program, facilitating automated batch experiments with different inputs.

The available arguments are listed below:

- `module` (positional) — Specifies the name of the program module to be measured. For Erlang, this refers to the module without the `.erl` extension; for C++, it refers to the base name of the source file (e.g., `example` for `example.cpp`).
- `function` (positional) — The entry function to be called within the module. While this is meaningful for Erlang, for C++ programs it may be a placeholder to distinguish configurations or measurement variants.
- `parameters` (positional, variadic) — One or more parameter combination to be passed to the program. Each measurement is repeated for every provided combination, enabling batch-style automation.
- `-c`, `--cmd` — A custom command string to override the internal execution logic. If specified, this command will be used directly for launching the measured application.
- `-e`, `--exe` — Specifies the name of the executable process to monitor (e.g., `beam.smp` for native, `erlang` for distributed Erlang or the binary name for C++). This name is used for identifying the correct process in the energy trace.
- `-r`, `--rep` — The number of repetitions to perform for each parameter set. This allows for averaging out fluctuations in power readings and execution time.
- `-f`, `--file` — Optional prefix for the result CSV file. If omitted, a default filename is constructed from the executable and module name.
- `--folder` — Destination folder for storing the JSON output from Scaphandre. Defaults to `measurements`.
- `--container` — A regular expression to match containers (e.g., by container names) if container-based execution is used. When this is set, the script enables container mode in the JSON Exporter of Scaphandre.
- `--nano` — Specifies the sample interval in nanoseconds for Scaphandre. If set to `0`, a default of 1 second is used.

- `-v`, `--verbose` — Enables detailed logging during script execution for debugging or monitoring purposes.

This parameterized architecture, together with its batch processing capability, makes the script suitable for extensive measurement campaigns without manual intervention.

Example Usage

The script can be invoked from the command line using:

```
sudo python3 measure_linux.py prime_calculation main \
  "4000000" \
  "8000000" \
  "12000000" \
  --rep 15 \
  --folder erl_measurements \
  --file prime_calculation_erl_results_raw \
  --nano 1000000 \
  --exe beam.smp

sudo python3 measure_linux.py prime_calculation "" \
  "4000000" \
  "8000000" \
  "12000000" \
  --rep 15 \
  --folder caf_measurements \
  --file prime_calculation_caf_results_raw \
  --nano 1000000 \
  --exe prime_calculation
```

A more detailed example can be found in Appendix A.

Preparing for Distributed Deployments

While native applications (both Erlang and CAF) can be measured with minimal to none preparation, distributed deployments require additional setup to ensure correct execution and reproducibility.

In the case of distributed Erlang deployments, multiple nodes must be launched manually prior to measurement. These nodes need to be initialized with the same `-setcookie` and assigned unique names using the `-sname` flag. Before executing the measurement command, the participating nodes must be connected using the `net_adm:ping(Node)` function. Furthermore, it is essential to verify that each node contains the latest compiled version of the Erlang module under test, to prevent inconsistencies during distributed function execution.

For distributed CAF deployments, the Kubernetes environment is managed via `KinD`. Before measurements, a `KinD` configuration file must be created that defines the desired number of worker nodes. An example configuration is provided in Subsection 2.6.2 of this thesis. The configuration is then used to instantiate a local Kubernetes cluster.

These preparatory steps ensure that distributed systems are initialized in a consistent state before measurements begin, enabling meaningful comparison with their native counterparts.

3.3 Post-Processing

To ensure the reliability of measurement data and minimize the influence of anomalous results, a custom post-processing script[3] was developed in Python using the `pandas` library. This script automates several cleanup and aggregation steps on the raw CSV files exported from the measurement environment.

First, the script removes entries with zero or negative values in critical columns such as `Runtime (s)` and `Consumption (`μ`J)`, as these typically indicate failed or invalid measurements. Next, it applies an Interquartile Range (IQR) filter to eliminate statistical outliers in the `Consumption (`μ`J)` column. This step improves robustness by excluding extreme values that could skew average energy results.

The script then groups the results by `Parameter`, and from each group selects the n most representative trials, those with energy values closest to the group's median. This ensures that the most consistent measurements are retained. By default, $n = 10$, but this can be adjusted via command-line arguments.

[3] `https://github.com/bszirtes/THESIS/blob/main/scripts/process_raw_measurements.py`

After filtering, the data is grouped by `Module`, `Function`, and `Parameter`, and all numerical columns are averaged to produce a condensed summary. The final dataset is saved in CSV format for use in subsequent analysis and visualization.

This post-processing step significantly improves data quality and consistency, helping to isolate meaningful trends in runtime and energy behavior across configurations.

Chapter 4

Designing the Algorithms to Measure

In order to conduct meaningful measurements across multiple runtime environments and communication patterns, a consistent set of benchmark cases was designed and implemented. These cases simulate simple but computationally and structurally distinct workloads that are representative of distributed systems behavior. The focus lies on comparing the cost of message passing, actor spawning, load balancing, and lightweight computation under varying levels of distribution.

This chapter presents the design of these benchmark cases, detailing their behavior and implementation across both Erlang and CAF, in native and distributed settings. Special attention is given to structural consistency across implementations, enabling fair comparison, while also documenting the platform-specific adaptations required to support deployment, coordination, and measurement.

4.1 Overview of Native Measurement Scenarios

To explore how language-level design and actor model implementations affect energy consumption, four application patterns were designed and implemented in both Erlang and the C++ Actor Framework (CAF). These scenarios incrementally increase in computational and communication complexity and are executed in a non-distributed (native) setting to establish a controlled baseline for measurement.

Each of the implemented programs adheres to the actor model and utilizes asynchronous message passing. The implementations share common structural princi-

ples across languages to enable meaningful energy comparison. Their purpose is not only to stress different aspects of system resources but also to reflect realistic microservice-like coordination and compute patterns.
The four scenarios are as follows:

- **Prime Number Calculation:** A purely computational task that calculates the number of prime numbers within a given range. This application does not involve inter-process communication, making it an ideal baseline for analyzing raw compute efficiency across languages. This test case will be referred to as `prime_calculation`.
- **Ping-Pong:** A message-heavy workload in which several client actors send repeated `ping` messages to a single server actor, which responds with corresponding `pong` messages. This program introduces communication overhead and actor scheduling dynamics, but retains minimal computational load. This test case will be referred to as `ping_pong`.
- **Ping-Pong with Load Balancer:** A more sophisticated message-passing structure, where client messages are routed through a load balancer actor that forwards them to multiple server actors in a round-robin manner. This design introduces dynamic message routing and parallelism on the server side, thereby simulating a lightweight coordination layer. This test case will be referred to as `ping_pong_lb`.
- **Ping-Pong with Load Balancer and Prime Calculation:** A combination of the previous patterns, this benchmark routes messages from multiple clients through a load balancer to a pool of server actors. Each server, upon receiving a request, calculates the number of primes up to a given input and returns the result. This scenario tests the system's ability to coordinate message routing and manage compute-intensive tasks concurrently, making it the most demanding configuration among the four. This test case will be referred to as `ping_pong_lb_prime`.

Each scenario accepts runtime parameters such as the number of clients, servers, messages, and computation range, allowing the experiments to be scaled and repeated consistently. All implementations support verbose mode for debugging and transparency during execution.

The following sections detail the final and most complex application, which combines the logic of the preceding ones, with a focus on the architecture, communication flow, and design choices that influence both performance and energy characteristics.

Ping-Pong with Load Balancer and Prime Calculation in Erlang

The most complex native Erlang scenario[1] designed for this thesis integrates both actor-based communication and compute-intensive tasks. It builds on the previous ping-pong with load balancing example by adding a layer of numeric computation, where server actors are tasked with counting the number of prime numbers up to a given limit.

The purpose of this program is to evaluate the energy impact of interleaving message routing with active CPU usage across multiple actors.

Architecture and Actor Roles

The system consists of five main components:

- **Main process:** Coordinates the startup and shutdown of the system, and monitors termination messages from clients, servers, and the load balancer.
- **Client actors:** Each client sends a series of requests to the system. Each request contains a numerical range for which the number of prime numbers should be computed. The client awaits the result of each computation before sending the next request.
- **Load balancer:** Acts as an intermediary that forwards messages from clients to server actors using round-robin scheduling. It ensures even distribution of computational load.
- **Server actors:** Each server receives a number from the client (via the load balancer) and calculates the number of prime numbers up to that limit. The result is sent back to the originating client.

[1] `https://github.com/bszirtes/THESIS/blob/main/src/native/ping_pong_lb_prime_erl/ping_pong_lb_prime.erl`

- **Prime-checking functions:** Utility functions `is_prime/1` and `find_primes/1` implement the actual logic for determining whether a number is prime and for counting all primes up to a given integer.

System Flow and Message Structure

The high-level control flow proceeds as follows:

1. The main process spawns the configured number of servers and launches a single load balancer process, passing it the list of server PIDs.
2. Clients are then spawned. Each sends a fixed number of requests to the load balancer, where each message consists of a tuple: `{ClientPID, Range}`.
3. The load balancer selects a server using round-robin indexing and forwards the message unchanged.
4. Upon receiving the range, the server calculates the number of prime numbers up to that value using `find_primes/1`, and replies to the client with the result.
5. After sending all messages, each client notifies the main process of its completion. When all clients finish, the main process terminates the servers by sending `stop` messages.
6. Once all servers have confirmed termination, the main process sends a `stop` message to the load balancer, concluding the execution.

Prime Calculation Logic

The computation is performed on the server side using a simple brute-force algorithm. The `is_prime/1` function checks if an integer n is divisible by any number from 2 to $\sqrt{n}$, and returns a boolean. The `find_primes/1` function then folds over the range from 2 to the input upper bound, counting all numbers for which `is_prime/1` returns `true`.

```
1 find_primes(Max) ->
2     lists:foldl(fun(N, Acc) ->
3                     case is_prime(N) of
4                         true -> Acc + 1;
```

```
5                        false -> Acc
6                    end
7                end, 0, lists:seq(2, Max)).
```

Code 4.1: Prime counting logic for Erlang

This function is invoked for every incoming message handled by a server, contributing a non-negligible computational load to each round-trip interaction.

Load Balancing Strategy

The load balancer maintains a rotating index over the list of available server PIDs. For each client message, it selects the next server in the sequence:

```
Server = lists:nth((Index rem length(Servers)) + 1, Servers),
Server ! {From, Msg},
```

This simple yet effective round-robin algorithm ensures equitable distribution of computation across the servers, which is critical for consistent energy profiling across parallel actors.

Termination Handling

The main process tracks completion using the pattern `{client, Pid, done}` from clients, and similar messages from servers and the load balancer. Once all expected messages are received, it orchestrates system-wide termination in the following order:

- Clients finish and notify the main process.
- Main sends `stop` to all servers.
- Servers notify the main process when done.
- Main sends `stop` to the load balancer.
- Load balancer notifies the main process when done.

This structured shutdown process prevents zombie processes and ensures clean energy measurement boundaries.

Performance Considerations

This benchmark maximizes both communication and CPU usage, providing an upper-bound scenario for actor-based energy profiling. The use of brute-force prime checking intentionally amplifies CPU activity to test system behavior under stress. The inclusion of a load balancer also tests message forwarding efficiency and actor scheduling overhead.

All actors operate concurrently under Erlang's lightweight process model, allowing scalable concurrency with minimal runtime management overhead.

Ping-Pong with Load Balancer and Prime Calculation in CAF

The C++ Actor Framework (CAF) implementation[2] mirrors the architecture and behavior of its Erlang counterpart, extending the previous ping-pong with load balancer example to include compute-heavy operations. This scenario aims to evaluate both CAF's message-passing performance and its raw compute efficiency in a native environment.

Actor System Design

The CAF implementation consists of the following actor components:

- **Main actor:** Responsible for orchestration and termination. It launches and links all other actors and monitors system shutdown via completion messages.
- **Client actors:** Each client sends a predefined number of messages to the load balancer. Every message contains a numerical range for which the number of prime numbers should be calculated. Clients await a response before sending the next message.
- **Load balancer:** Distributes incoming client messages to a pool of server actors using a round-robin scheduling algorithm.
- **Server actors:** Upon receiving a numerical range, each server computes the number of primes within that range and replies directly to the requesting client.

[2] `https://github.com/bszirtes/THESIS/blob/main/src/native/ping_pong_lb_prime_caf/ping_pong_lb_prime.cpp`

All actors in this implementation are created using CAF's `event_based_actor` or `stateful_actor` actor types. [17]

Computation Logic

The prime-checking routine uses a naive brute-force method similar to the Erlang implementation. It checks whether each number from 2 to the upper bound is divisible by any smaller number up to its square root. The implementation is written in pure C++ for consistency with the rest of the system:

```
int find_primes(int range) {
  int prime_count = 0;
  for (int n = 2; n <= range; ++n) {
    if (is_prime(n)) {
      ++prime_count;
    }
  }
  return prime_count;
}
```

Code 4.2: Prime counting logic for CAF

This function is called by each server actor in response to a message.

Client Behavior

Clients are spawned by the main actor and interact with the load balancer through message passing. Each client sends the same numeric range multiple times and waits for the result before sending the next message. When all requests are fulfilled, the client sends a shutdown message to the main actor and terminates. CAF's `aout()` stream is used for optional verbose logging throughout the process.

Load Balancing Mechanism

The load balancer is implemented as a `stateful_actor` maintaining an internal state, that contains an index to track which server should handle the next message. It receives messages of the form `(client, int)` and forwards them to a server actor in round-robin order:

```
self->send(servers[self->state.server_index], client, n);
self->state.server_index =
    (self->state.server_index + 1) % servers.size();
```

Termination is handled by a message of type `atom("stop")`, which triggers notification to the main actor and actor shutdown.

Main Actor Coordination

The main actor manages the entire system's lifecycle and holds an internal state for:

- A reference to the load balancer
- A list of server actors
- Counters for completed clients and servers

Once all clients signal completion, the main actor sends a `stop` message to each server. When the servers report back, the main actor sends a final `stop` message to the load balancer and concludes the system execution:

```
for (auto &server : self->state.servers) {
  self->send(server, atom("stop"));
}
...
self->send(self->state.load_balancer, atom("stop"));
```

This design ensures that all components shut down in a deterministic and observable way, enabling precise measurement boundaries.

Resource Utilization and Behavior

This application combines two dominant energy factors:

1. **Message overhead:** Clients communicate through a centralized balancer, increasing indirection and actor switching.
2. **Computational stress:** Server actors repeatedly execute a CPU-bound loop to compute prime numbers, which places significant pressure on CPU cycles and actor scheduling.

CAF's architecture ensures that all actors execute concurrently.

4.2 Distributed Implementations

This section introduces the distributed variants of the previously discussed actor-based systems, showcasing how the same logical patterns were extended across multiple nodes or containers. Both Erlang and the C++ Actor Framework (CAF) versions of the `ping_pong`, `ping_pong_lb`, and `ping_pong_lb_prime` cases were redesigned for deployment in a distributed environment. The main focus of this section is not the core message-passing logic, which remains largely unchanged, but rather the architectural and infrastructural transformations required to facilitate communication and orchestration across node boundaries.

Architectural Enhancements for Distribution

The primary challenge in transitioning to a distributed setting is enabling actor processes to interact across networked nodes, while preserving actor semantics such as location transparency and asynchronous messaging. This required several key modifications:

- **Naming and Lookup:** Erlang relies on the `global` module for naming and locating actors across nodes. For example, the server (in the `ping_pong` case) or the load balancer (in the `ping_pong_lb` and `ping_pong_lb_prime` cases) is registered with a globally known name using:

  ```
  global:register_name(ping_pong_server, spawn(...)).
  ```

  CAF, by contrast, uses explicit networking primitives: actors are published on TCP ports using `publish()`, and remote actors are resolved using `remote_actor()`.

- **Modular Separation:** Each actor type (client, server, load balancer) is moved into its own source file or compilation unit. This decouples component responsibilities and enables independent deployment, testing, and scaling.

- **Deployment Lifecycle:** Both implementations introduce orchestration logic to automate deployment, monitoring, and termination across nodes.

Deployment and Measurement Strategies

Given the goal of conducting reproducible and realistic performance experiments, particular attention was paid to how and when actors are spawned and terminated.

Erlang cases use custom deployment modules written in Erlang to:

1. Parse configuration arguments (e.g., number of nodes, clients, messages).
2. Spawn load balancers and servers on specified nodes.
3. Distribute clients in a round-robin fashion.
4. Synchronize and coordinate termination using inter-process messages.

CAF cases leverage Kubernetes to orchestrate deployments using scripts written in Bash. For each distributed case:

- Docker images are built for each component (client, server, load balancer).
- The system is deployed using dynamically generated YAML manifests via a Bash script.
- Load balancer and servers are either freshly deployed or reused using flags such as `--no-load-balancer`.
- Clients are spawned as Kubernetes Jobs, allowing parallel, isolated execution.

The following Kubernetes job snippet shows how client jobs[3] are defined in the distributed CAF system:

```
1 apiVersion: batch/v1
2 kind: Job
3 metadata:
4   name: caf-client
5   namespace: ping-pong
6 spec:
7   completions: 2
8   parallelism: 2
```

[3] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_caf/client/client-job.yaml

```
  template:
    metadata:
      labels:
        app: caf-client
    spec:
      restartPolicy: Never
      containers:
        - name: caf-client
          image: localhost:5001/ping-pong-client:v4
          args:
            - "--server-address=caf-server"
            - "--messages=20"
            - "--delay=1000"
            - "--verbose"
```

Code 4.3: CAF Client Kubernetes Job

Here, the values `completions: 2` and `parallelism: 2` specify that two client job pods are created and executed concurrently.

Measurement-Oriented Deployment

To support reliable benchmarking and avoid introducing artificial delays from orchestration, both Erlang and CAF systems provide flags for measurement mode. These include:

- `no-server` (Erlang) or `--no-server` (CAF): Skips the deployment and shutdown of the server component.
- `no-load-balancer` (Erlang) or `--no-load-balancer` (CAF): Used in the `ping_pong_lb` and `ping_pong_lb_prime` cases to avoid interfering with predeployed load balancers and servers.

These modes ensure that measurement windows are isolated and do not include startup or teardown times of the components, except the lightweight clients, providing more accurate and consistent metrics.

Summary of Differences

Aspect	Erlang	CAF
Actor Lookup	`global:register/send`	`publish + remote_actor`
Deployment Language	Erlang module	Bash + Kubernetes
Network Abstraction	Node names via Erlang cookies	Ports and service discovery
Dynamic Scaling	`spawn` to node	Kubernetes `replicas`
Measurement Mode Flags	`no-server`, `no-load-balancer`	`--no-server`, `--no-load-balancer`

Table 4.1: Comparison of distributed implementation aspects between Erlang and CAF

The next sections will dive into the implementation details of the most advanced distributed variant, `ping_pong_lb_prime`, for both Erlang and CAF.

Distributed Erlang: `ping_pong_lb_prime`

This section presents the major components and behaviors of the distributed `ping_pong_lb_prime` system, with a focus on its distributed execution, orchestration, and termination logic.

System Behavior and Flow

At runtime, the system executes as follows:

1. The load balancer is globally registered and started on a designated node, if it is not already running.
2. Worker servers connect to the load balancer and register themselves.
3. Clients send prime calculation tasks to the load balancer, which routes them to servers in round-robin fashion.
4. Each server calculates the number of primes up to a given number and returns the result to the originating client.
5. The system terminates once all clients have completed.

Load Balancer

The load balancer[4] manages the server pool and handles the forwarding of the computation requests. It is globally registered via:

```
global:register_name(ping_pong_load_balancer, spawn(fun() ->
    load_balancer([], 0, Main, Verbose)
end)).
```

In addition to the native implementation, the load balancer supports a `servers` query message, responding with a list of registered servers. This feature is used by the deployment script to facilitate the scaling of the servers.

Server Behavior

Each server[5] registers itself with the load balancer on startup:

```
global:send(ping_pong_load_balancer, {self(), register})
```

Then it waits for messages in the form `{ClientPid, PrimeRange}`, performs the prime-counting logic locally and sends the result back.

Client Behavior

The client[6] actor sends a given number of computation requests with the same range value to the load balancer using `global:send`. It awaits a plain integer response for each request. After completing all messages, it notifies the main deployment process to enable controlled shutdown.

Deployment Manager

The deployment logic is handled by a dedicated module[7], which provides the full lifecycle management. Key functions include:

[4] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_erl/ping_pong_lb_prime_load_balancer.erl

[5] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_erl/ping_pong_lb_prime_server.erl

[6] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_erl/ping_pong_lb_prime_client.erl

[7] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_erl/deploy_ping_pong_lb_prime.erl

- `start_load_balancer/4`: Conditionally starts the load balancer depending on the `no-load-balancer` flag.
- `distribute_servers/4`: Dynamically adjusts the number of active servers by querying the load balancer and spawning additional workers if needed.
- `distribute_clients/5`: Distributes client processes across nodes in a round-robin pattern.
- `stop_servers/2` and `stop_load_balancer/2`: Gracefully terminates the managed actors and synchronizes the final shutdown.

Measurement Mode and Flexibility

For accurate performance measurement, the system supports a flag `no-load-balancer` to skip the deployment and teardown of the load balancer and servers. This approach enables pre-deployment of components and allows server scaling between executions, helping to stabilize the system and eliminate orchestration overhead from benchmarking runs.

Scalability and Robustness Features

Several design choices enhance the scalability and robustness of the system:

- Load balancer is stateless with respect to task execution and only handles routing.
- Servers can be started dynamically and register themselves without synchronization requirements.
- Clients and servers do not share any state and operate independently.
- The main deployment process tracks all actors and enforces deterministic shutdown, ensuring no actor is left running post-experiment.

Summary

The distributed Erlang `ping_pong_lb_prime` system demonstrates how actor-based computations can be extended to a real-world setting with a central load balancer and scalable workers. Through modularization, global naming, and precise

orchestration, the system achieves high flexibility, good scalability, and support for detailed experimentation workflows.

The next section presents the CAF version of the same system, with emphasis on Kubernetes orchestration and binary deployment strategies.

Distributed CAF: `ping_pong_lb_prime`

The `ping_pong_lb_prime` case in CAF mirrors its Erlang counterpart in terms of logical flow but diverges significantly in implementation and deployment. Rather than relying on a shared runtime or Erlang's inter-node message passing, this implementation compiles actors into standalone binaries, which are containerized and deployed to Kubernetes using custom Bash deployment scripts. This section explores the architectural structure, communication mechanisms, and container orchestration flow.

System Architecture Overview

The system consists of the following containerized components:

- **CAF Load Balancer:** Routes client requests to registered servers using a round-robin strategy.
- **CAF Server(s):** Handle prime-counting computations and return results to the clients.
- **CAF Client(s):** Generate requests and await prime calculation results.

Each actor type is compiled into a standalone executable and launched in containers governed by Kubernetes deployments and jobs. Communication occurs over TCP via CAF's `middleman` module.

Load Balancer

The load balancer[8] actor maintains a list of registered server actors and distributes messages in a round-robin fashion:

[8] `https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_caf/load_balancer/ping_pong_lb_prime_load_balancer.cpp`

```
behavior load_balancer_fun(stateful_actor<lb_state>* self) {
  return {
    [=](const actor& actor, atom_value msg) {
      if (msg == atom("register")) {
        self->state.servers.push_back(actor);
        aout(self) << "Registered server <" << actor->address() << "
            ↪ >\n";
      }
    },
    [=](const actor& actor, int range) {
      auto& s = self->state.servers[self->state.server_index];
      self->send(s, actor, range);
      self->state.server_index = (self->state.server_index + 1) %
          ↪ self->state.servers.size();
    }
  };
}
```

Code 4.4: CAF Load Balancer Behavior

The actor is published over the network using CAF's `publish()`:

```
auto res = sys.middleman().publish(balancer, cfg.port);
```

This enables external actors to remotely connect and interact with the load balancer.

Server Behavior

Each server[9] connects to the load balancer using `remote_actor()` and registers itself:

```
auto load_balancer = sys.middleman().remote_actor(cfg.
    ↪ load_balancer_address, cfg.load_balancer_port);
self->send(*load_balancer, self, atom("register"));
```

[9] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_caf/server/ping_pong_lb_prime_server.cpp

After the connection is successfully established, the server is ready to receive computation requests forwarded by the load balancer. The actor behavior includes termination logic, although it is not triggered in this setup, rather handled externally by Kubernetes cleanup.

Client Behavior

Clients[10] connect to the load balancer using TCP, send computation requests, and receive plain integer responses. Once the configured number of messages is handled, the client terminates.

```
self->send(load_balancer, self, prime_range);
...
[=](int found_primes) mutable {
  ++responses_received;
  if (responses_received == num_messages) self->quit();
  else self->send(load_balancer, self, prime_range);
}
```

Code 4.5: CAF Client Behavior

Deployment via Kubernetes and Bash scripts

Unlike Erlang's runtime-controlled deployment, the CAF version leverages Bash scripts to dynamically generate YAML config files and invoke Kubernetes operations.

Dynamic YAML Generation: The script[11] constructs deployment/job definitions for the components:

- `temp-server-deployment.yaml` for servers.
- `temp-client-job.yaml` for clients.

[10] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_caf/client/ping_pong_lb_prime_client.cpp

[11] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_caf/deploy_ping_pong_lb_prime.sh

```
cat <<EOF >"$SCRIPT_DIR"/temp-client-job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: caf-client
spec:
  completions: $NUM_CLIENTS
  parallelism: $NUM_CLIENTS
  template:
    spec:
      containers:
        - name: caf-client
          image: $CLIENT_IMAGE
          args:
            - "--server-address=caf-load-balancer"
            - "--server-port=4242"
            - "--messages=$NUM_MESSAGES"
            - "--range=$PRIME_RANGE"
EOF
```

Code 4.6: Client YAML Generation Snippet

The load balancer configuration remains unchanged and is deployed using its corresponding manifest file[12].

Orchestration Flow:

1. Load balancer and its service are deployed (unless `--no-load-balancer` is passed).
2. Servers are launched and connect to the load balancer.
3. Clients are submitted as a Kubernetes job with `completions = parallelism = N`.

[12] https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_caf/load_balancer/load-balancer-deployment.yaml

4. Script polls for job completion using `kubectl wait`.

5. All of the deployed components are deleted.

Containerization

Each component is built into a statically linked binary and placed into a minimal container using a Dockerfile. An example for the load balancer[13]:

```
FROM ubuntu:22.04

RUN apt-get update && apt-get install -y \
    g++ cmake libcaf-dev && rm -rf /var/lib/apt/lists/*

WORKDIR /app
COPY ping_pong_lb_prime_load_balancer.out /app/
    ↪ ping_pong_lb_prime_load_balancer

EXPOSE 4242
ENTRYPOINT ["/app/ping_pong_lb_prime_load_balancer"]
```

Code 4.7: Load Balancer Dockerfile

This pattern is followed for servers and clients as well, using different binaries and arguments.

Local Registry and Image Management

One of the implementation decisions in this project was to build Docker images manually and store them in a local Docker registry, accessible at `localhost:5001`. This allowed for:

- Full control over the image building and versioning process.
- Fast iteration without requiring upload to a remote registry.
- Local caching and reuse of large base images (e.g., CAF-enabled Ubuntu).

[13] `https://github.com/bszirtes/THESIS/blob/main/src/distributed/ping_pong_lb_prime_caf/load_balancer/Dockerfile`

As a result, the image references in the deployment scripts consistently use the prefix `localhost:5001`, e.g.,

```
SERVER_IMAGE="localhost:5001/ping-pong-lb-server:v2"
CLIENT_IMAGE="localhost:5001/ping-pong-client:v4"
```

These images are pulled by Kubernetes during pod scheduling.

Measurement Mode

The system provides a `--no-load-balancer` flag to skip deployment and shutdown of the load balancer and servers. This allows for real-world benchmark scenarios where server instances are kept alive across runs.

Challenges and Observations

- Unlike Erlang, CAF does not have built-in notion of global name registration across nodes, requiring explicit addressing.
- Both message types and actor identities require manual serialization and interpretation over TCP.
- System-wide coordination (e.g., when all actors have completed) must be externalized or handled by Kubernetes-level observation.

Summary

The `ping_pong_lb_prime` case in CAF demonstrates how actor-based systems can be deployed as containerized microservices, leveraging Kubernetes for elasticity, orchestration, and life cycle control. Though requiring more boilerplate and infrastructure setup compared to Erlang, this approach better aligns with industry-standard deployment pipelines and enables integration with broader cloud-native stacks.

Chapter 5

Analyzing the Results

This chapter presents a comprehensive evaluation of the benchmarks introduced in the previous sections. The results are organized by execution environment, starting with native benchmarks and followed by distributed deployments. Each experiment was repeated across a range of parameter configurations to reveal trends in runtime and energy efficiency. Whenever possible, performance and energy metrics are compared across both actor frameworks—Erlang and CAF—as well as across deployment models to highlight the trade-offs introduced by distribution and parallelism. The complete individual results are accessible in the thesis repository [1], with the raw JSON measurements contained within a compressed archive [2].

5.1 Native Results

The following section presents measurements gathered from native executions of all benchmark variants. These results serve as a performance and efficiency baseline, unaffected by distributed coordination or network-induced overhead. They reflect the raw computational and concurrency capabilities of each language under isolated conditions.

Prime Calculation

This benchmark measures the computational and energy characteristics of calculating prime numbers up to a given limit. Three different input sizes were chosen:

[1] https://github.com/bszirtes/THESIS/tree/main/results

[2] https://ikelte-my.sharepoint.com/:u:/g/personal/luvfez_inf_elte_hu/EVRbopG7CDpEgd3hs0Kg5GgByn7dr-udnSdbxIcU1yCGqw?e=mSFrgR

4 million, 8 million, and 12 million. The results shown are averages of multiple measurement runs.

Erlang

Table 5.1 shows the runtime, average power draw, and total energy consumption of the Erlang implementation. As expected, with increasing input size, the execution time and energy consumption rise significantly, while the average power consumption shows a decreasing trend, possibly due to more consistent CPU utilization over longer durations.

Table 5.1: Erlang `prime_calculation` results

Input Size	**Runtime (s)**	**Avg Power (W)**	**Energy (J)**
4,000,000	19.42	0.54	10.53
8,000,000	53.42	0.50	26.89
12,000,000	165.28	0.45	74.33

CAF

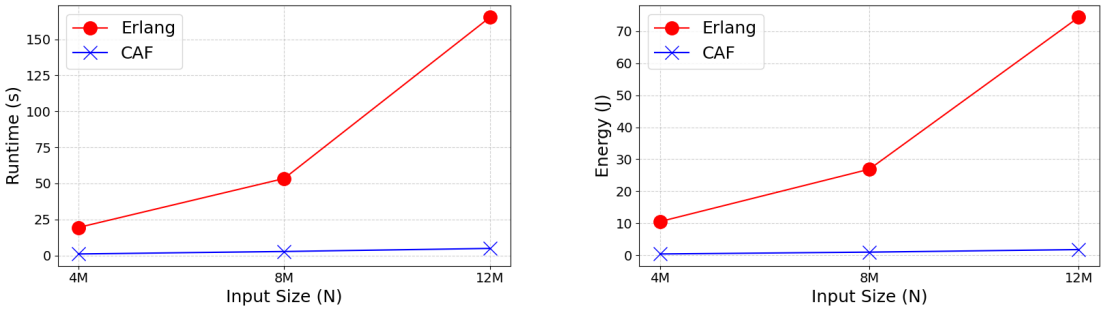
CAF shows a much more efficient performance in the same task, both in terms of execution time and energy consumption. Table 5.2 presents the results for the same input sizes. Even at the highest input size, the runtime remains below 5 seconds, highlighting the strong performance of CAF in this compute-bound scenario.

Table 5.2: CAF `prime_calculation` results

Input Size	**Runtime (s)**	**Avg Power (W)**	**Energy (J)**
4,000,000	1.07	0.39	0.42
8,000,000	2.79	0.36	0.99
12,000,000	4.91	0.37	1.79

Comparison

The performance delta between the Erlang and CAF implementations is substantial in both time and energy. CAF is approximately 16-33 times faster and significantly more energy-efficient for all inputs tested (Figure 5.1).



(a) Runtime Comparison (b) Energy Consumption Comparison

Figure 5.1: Prime Calculation Comparison

Ping-Pong

The ping-pong case simulates a message-passing system where multiple clients send ping messages to a central server and wait for pong responses. The objective is to measure the runtime and energy efficiency of this concurrent message-passing pattern in both Erlang and CAF across varying workloads.

Erlang

The Erlang results in Table 5.3 demonstrate a linear increase in runtime and energy consumption as the number of clients and messages increases. Notably, the average power draw also increases steadily, indicating intensive CPU usage due to concurrent actor scheduling.

Table 5.3: Erlang `ping_pong` results

Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
50	50000	3.18	2.60	8.23
50	100000	5.62	2.40	13.45
50	200000	10.15	2.35	23.80
50	500000	24.47	2.33	57.10
100	50000	6.77	2.49	16.84
100	100000	12.08	2.54	30.64
100	200000	22.63	2.55	57.72
100	500000	56.81	2.58	146.45
200	50000	13.11	2.62	34.35
200	100000	24.77	2.67	66.12
200	200000	51.16	2.68	137.30
200	500000	134.17	2.68	359.66
400	50000	30.03	2.82	84.53
400	100000	62.26	2.82	175.39
400	200000	134.13	2.94	395.32
400	500000	372.46	3.01	1123.19

CAF

The CAF results for the `ping_pong` benchmark, summarized in Table 5.4, demonstrate a considerably different performance profile compared to Erlang. While CAF's average power consumption is significantly lower, this does not directly translate into better runtime efficiency.

In fact, across all configurations, the runtime values are consistently higher than those observed in Erlang, often by a large margin. For example, with 400 clients sending 500,000 messages each, the CAF runtime reaches over 479 seconds, compared to Erlang's 372 seconds in the same setup.

However, the trade-off lies in the dramatically lower power consumption. Despite longer execution times, the total energy consumption is still lower due to CAF's ability to maintain relatively low average power usage.

Table 5.4: CAF `ping_pong` results

Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
50	50000	7.48	0.66	4.91
50	100000	15.45	0.66	10.19
50	200000	30.92	0.67	20.58
50	500000	76.10	0.67	51.05
100	50000	14.50	0.69	10.06
100	100000	29.53	0.69	20.44
100	200000	58.28	0.69	40.45
100	500000	147.97	0.70	104.04
200	50000	27.12	0.73	19.89
200	100000	54.81	0.73	40.01
200	200000	109.61	0.74	81.16
200	500000	272.16	0.74	202.54
400	50000	48.15	0.79	38.02
400	100000	95.49	0.80	76.22
400	200000	190.08	0.80	152.37
400	500000	479.97	0.80	383.88

Comparison

The results of the ping-pong benchmark demonstrate an interesting contrast between Erlang and CAF. While Erlang achieves lower runtimes, this comes at the cost of significantly higher power draw — averaging between 2.3 and 3.0 Watts across configurations. CAF, on the other hand, maintains much lower power usage (around 0.65–0.80 W), which leads to better energy efficiency despite higher runtimes.

This trade-off highlights the distinct philosophies of the two runtimes: Erlang aggressively utilizes CPU resources to minimize execution time, whereas CAF is more

conservative in resource usage, leading to reduced energy expenditure. Figure 5.2 and Figure 5.3 provide a visual representation of this trade-off.

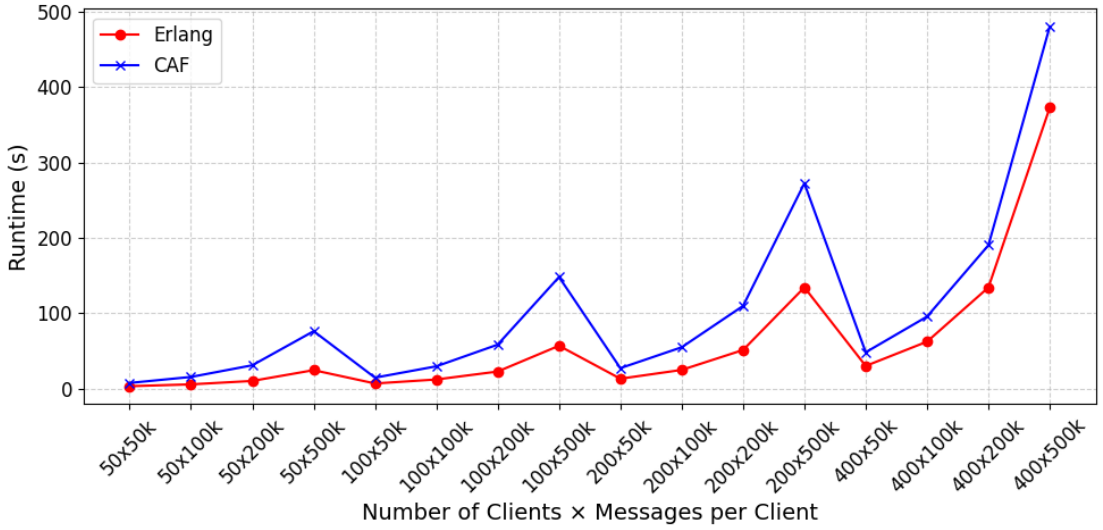


Figure 5.2: Ping-Pong Runtime Comparison

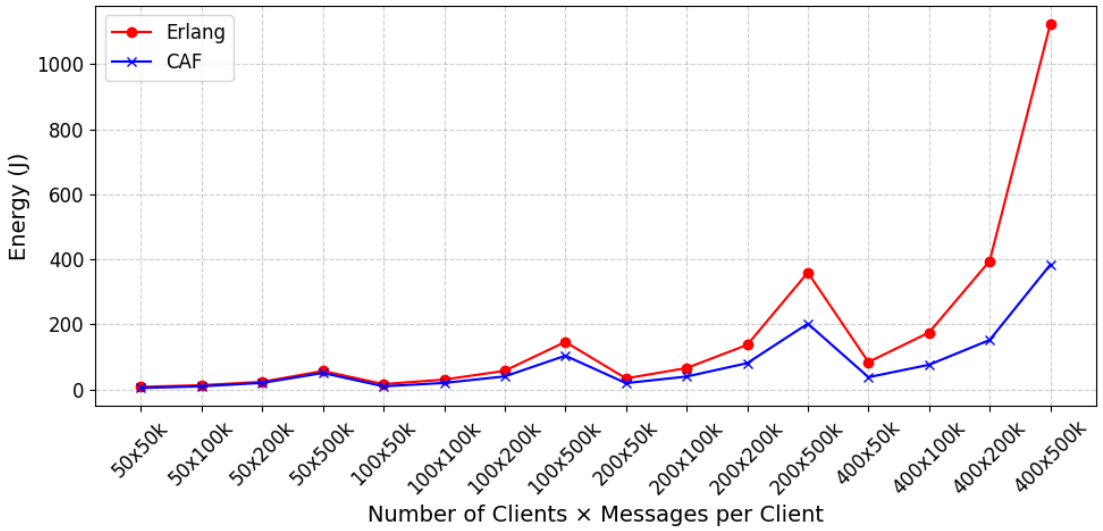


Figure 5.3: Ping-Pong Energy Consumption Comparison

Ping-Pong with Load Balancer Results

In this benchmark, a load balancer is introduced between clients and servers to distribute the workload. The goal is to evaluate how the addition of this component affects both the runtime performance and energy consumption across different concurrency levels.

Erlang

Table 5.5 summarizes the energy, runtime, and power characteristics for the native Erlang implementation of the `ping_pong_lb` benchmark. Each row represents an experiment conducted with a specific number of clients and servers, with each client sending a fixed number of messages.

Table 5.5: Erlang `ping_pong_lb` results

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	7.03	8.67	60.87
2	200	100000	14.24	9.74	138.80
2	200	200000	28.69	9.93	285.40
2	200	500000	68.40	9.60	656.70
2	400	50000	15.27	11.61	177.69
2	400	100000	35.63	12.07	427.28
2	400	200000	71.87	11.96	854.82
2	400	500000	178.68	12.23	2181.38
4	200	50000	9.65	6.39	61.69
4	200	100000	19.04	6.57	125.07
4	200	200000	38.11	7.00	266.63
4	200	500000	95.04	7.19	683.30
4	400	50000	20.56	8.07	165.90
4	400	100000	40.85	8.13	331.99
4	400	200000	81.09	8.41	682.21
4	400	500000	204.69	8.93	1826.96
8	200	50000	12.02	5.93	71.20
8	200	100000	23.73	5.84	138.43
8	200	200000	47.30	5.91	279.67
8	200	500000	118.63	6.52	773.14
8	400	50000	24.57	7.05	173.12
8	400	100000	48.81	6.72	328.22
8	400	200000	97.58	6.99	682.43
8	400	500000	243.26	8.34	2028.00

Erlang's runtime energy consumption grows near-linearly with the message and client counts, reflecting efficient scheduling and message-passing overhead management. Interestingly, increasing the number of servers reduces the average power consumption (notably in the 8-server case), which leads to lower energy cost per message compared to smaller configurations. This indicates improved concurrency handling at scale, despite the higher absolute energy numbers.

CAF

Table 5.6 shows the corresponding results for the CAF implementation. Here we observe longer runtimes compared to Erlang, but drastically reduced power usage across the board.

Table 5.6: CAF `ping_pong_lb` results

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	39.48	1.03	40.45
2	200	100000	79.30	1.03	81.96
2	200	200000	158.38	1.04	165.45
2	200	500000	397.06	1.05	416.19
2	400	50000	69.67	1.15	80.14
2	400	100000	138.38	1.16	159.96
2	400	200000	273.39	1.16	317.53
2	400	500000	687.32	0.88	603.47
4	200	50000	39.57	1.04	41.34
4	200	100000	79.39	1.07	85.13
4	200	200000	160.91	1.07	171.76
4	200	500000	395.96	1.07	425.37
4	400	50000	68.47	0.88	60.02
4	400	100000	136.23	0.88	120.19
4	400	200000	270.44	0.89	240.58
4	400	500000	683.37	0.89	611.00
8	200	50000	37.86	1.10	41.46
8	200	100000	76.69	1.09	83.40
8	200	200000	154.27	1.09	168.70
8	200	500000	385.60	1.10	423.19
8	400	50000	67.91	0.89	60.24
8	400	100000	133.88	0.89	118.54
8	400	200000	266.17	0.89	236.26
8	400	500000	675.53	0.89	599.52

CAF demonstrates extremely stable and low power consumption across all configurations, rarely exceeding 1.2 W. However, this efficiency comes at the cost of performance. In every configuration, runtimes are significantly longer than their Erlang counterparts, often two to four times longer. The energy values show a relatively linear increase with message volume and client count, which suggests that CAF's task distribution and load balancing are more uniform but also less aggressive in exploiting concurrency.

Comparison

The `ping_pong_lb` experiment results highlight a clear trade-off between Erlang's high-throughput concurrency model and CAF's energy-conscious design.

- Erlang outperforms CAF in terms of runtime across nearly all configurations.
- However, CAF consumes significantly less energy due to its low average power usage compared to Erlang.
- As the number of servers increases, Erlang's power consumption grows more aggressively, while CAF's remains mostly flat.

Figure 5.4 and Figure 5.5 show the runtime and energy comparisons for both runtimes, grouped by language and server count (`S`).

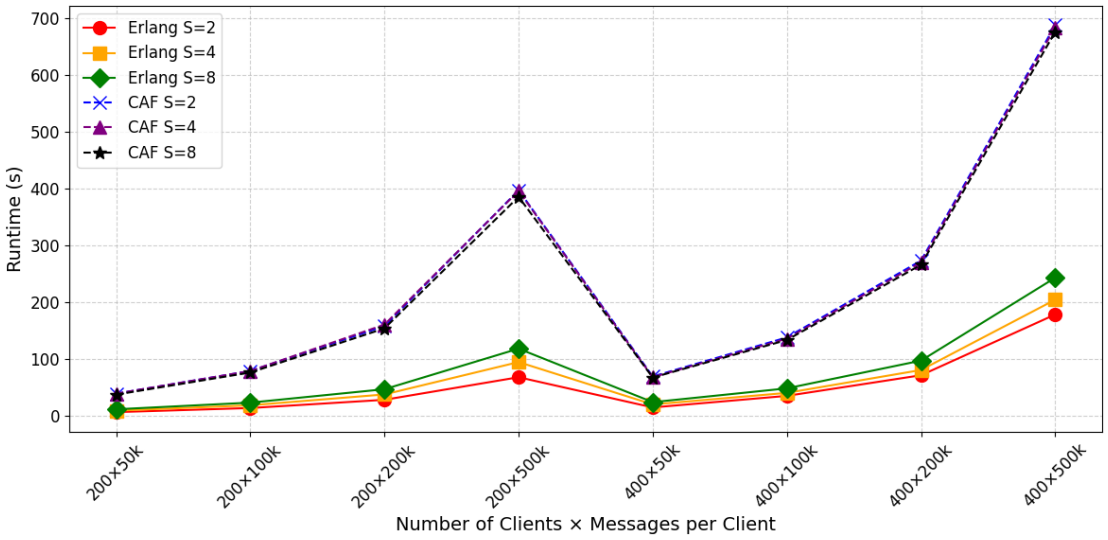


Figure 5.4: Ping-Pong with Load Balancer Runtime Comparison

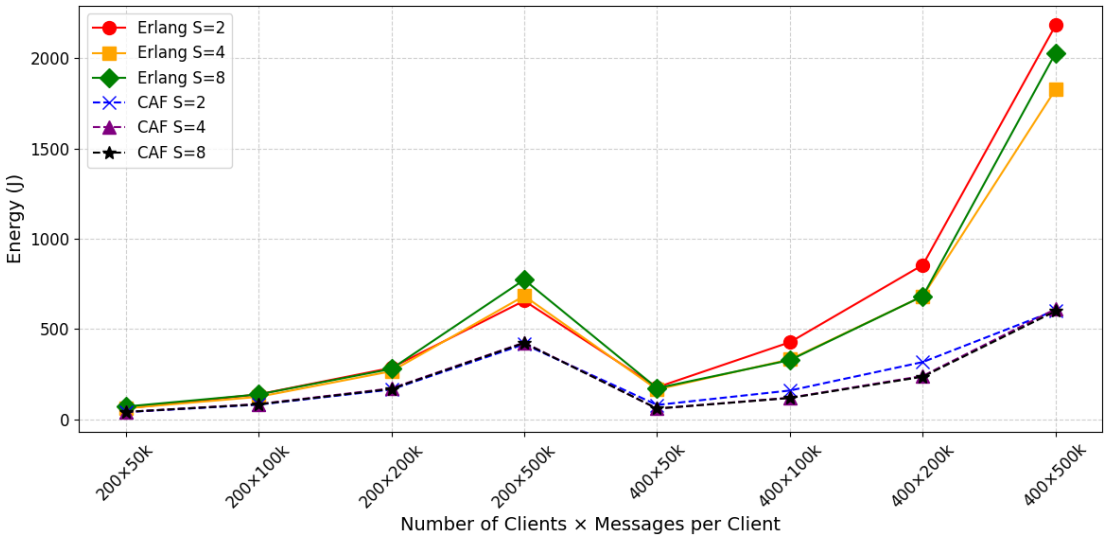


Figure 5.5: Ping-Pong with Load Balancer Energy Consumption Comparison

Ping-Pong with Load Balancer and Prime Calculation Results

This benchmark case combines the characteristics of load balancing with computational workload by having each server compute the number of prime numbers up to a specified upper bound. The final parameter in each measurement (either 50 or

100) defines this upper bound and directly affects CPU utilization on the server side. The goal is to examine how Erlang and CAF handle increasing compute intensity in a concurrent messaging environment.

Erlang

The performance data for the Erlang implementation is shown in Table 5.7 and Table 5.8. As expected, both runtime and energy consumption increase with larger client counts and higher message loads. Increasing the number of servers generally improves performance, particularly when moving from 2 to 4 servers. However, gains beyond 4 servers are limited: although runtimes decrease slightly, the energy consumption rises substantially due to significantly higher power usage.

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	59.22	2.02	119.38
2	200	100000	119.81	1.92	229.16
2	200	200000	233.46	2.02	470.43
2	400	50000	116.96	2.10	245.30
2	400	100000	231.42	2.13	491.62
2	400	200000	465.39	2.12	985.34
4	200	50000	38.05	3.47	131.87
4	200	100000	75.52	3.56	268.58
4	200	200000	152.27	3.47	528.25
4	400	50000	79.95	3.31	264.19
4	400	100000	158.98	3.35	533.18
4	400	200000	318.11	3.36	1070.09
8	200	50000	37.05	6.33	234.36
8	200	100000	73.57	6.49	477.58
8	200	200000	146.46	6.58	963.23
8	400	50000	73.80	7.32	540.13
8	400	100000	146.99	7.25	1066.13
8	400	200000	295.86	7.42	2196.55

Table 5.7: Erlang `ping_pong_lb_prime` results (Prime Range = 50)

In terms of energy efficiency, there is a clear penalty associated with a higher prime range (100), as seen in the elevated energy consumption values across nearly every configuration. While runtimes roughly double when moving from Prime Range 50 to 100, energy usage more than doubles in many cases, highlighting the increased CPU-boundedness of the workload. For larger configurations like 400 clients and 200,000 messages, runtime and energy demands escalate sharply, further underscoring the computation-heavy nature of the benchmark.

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	115.08	1.56	179.53
2	200	100000	226.81	1.59	359.94
2	200	200000	448.68	1.61	720.17
2	400	50000	249.28	1.50	373.31
2	400	100000	509.99	1.49	761.37
2	400	200000	1014.97	1.51	1531.74
4	200	50000	74.25	2.97	220.52
4	200	100000	147.93	3.01	445.78
4	200	200000	295.30	2.96	874.57
4	400	50000	155.04	2.81	435.03
4	400	100000	309.34	2.85	882.57
4	400	200000	617.38	2.85	1761.81
8	200	50000	73.48	6.01	441.31
8	200	100000	146.20	6.11	892.99
8	200	200000	296.36	6.08	1800.65
8	400	50000	147.77	6.73	994.49
8	400	100000	292.20	6.76	1976.25
8	400	200000	596.69	6.82	4069.14

Table 5.8: Erlang `ping_pong_lb_prime` results (Prime Range = 100)

CAF

The results for the CAF implementation are shown in Tables 5.9 and 5.10.

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	43.77	0.81	35.32
2	200	100000	87.17	0.83	72.12
2	200	200000	175.68	0.83	144.93
2	400	50000	78.46	0.88	69.26
2	400	100000	158.03	0.88	139.44
2	400	200000	311.21	0.89	275.95
4	200	50000	42.60	0.87	36.87
4	200	100000	86.26	0.87	75.15
4	200	200000	169.90	0.87	147.62
4	400	50000	71.82	0.98	70.08
4	400	100000	147.59	0.97	143.71
4	400	200000	287.68	0.98	283.27
8	200	50000	41.05	0.89	36.53
8	200	100000	84.92	0.88	74.89
8	200	200000	165.54	0.89	147.49
8	400	50000	69.28	1.01	69.99
8	400	100000	138.35	1.02	140.92
8	400	200000	280.25	1.01	282.58

Table 5.9: CAF `ping_pong_lb_prime` results (Prime Range = 50)

CAF consistently delivers much lower energy consumption than Erlang across all configurations — often by a factor of 3–5× — even when runtimes are comparable or slightly longer. This suggests a more power-efficient internal execution model, likely due to more conservative CPU usage and efficient task management.

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	50.64	0.83	41.91
2	200	100000	102.13	0.83	84.52
2	200	200000	207.81	0.82	170.82
2	400	50000	89.38	0.90	80.63
2	400	100000	184.21	0.90	164.86
2	400	200000	357.32	0.91	324.22
4	200	50000	47.82	0.87	41.49
4	200	100000	96.38	0.88	84.32
4	200	200000	191.40	0.88	169.08
4	400	50000	81.27	0.99	80.53
4	400	100000	164.60	1.11	183.24
4	400	200000	330.02	1.12	369.87
8	200	50000	45.93	0.92	42.04
8	200	100000	92.62	0.91	84.38
8	200	200000	184.11	0.92	169.28
8	400	50000	77.86	1.20	93.01
8	400	100000	155.02	1.23	190.81
8	400	200000	311.42	1.22	381.40

Table 5.10: CAF `ping_pong_lb_prime` results (Prime Range = 100)

Interestingly, runtime scalability with increased server counts is less aggressive compared to Erlang. Adding more servers improves performance, but only modestly. However, the average power draw remains consistently low, generally staying under 1.25 W even in the largest test configurations. Notably, at high client and message loads (e.g., 400 clients, 200,000 messages), CAF maintains stable and efficient behavior, avoiding the sharp increases in energy consumption seen in Erlang.

Comparison

The performance differences between CAF and Erlang in the `ping_pong_lb_prime` native case are more nuanced compared to previous workloads.

In the **Prime Range = 50** configuration, in terms of runtime, Erlang scales better and even proves to be faster in some cases as the number of servers increases. This can be attributed to Erlang's lightweight process model, which favors shorter, communication-heavy computations. However, CAF compensates with significantly lower energy consumption due to its more conservative scheduling and reduced CPU overhead.

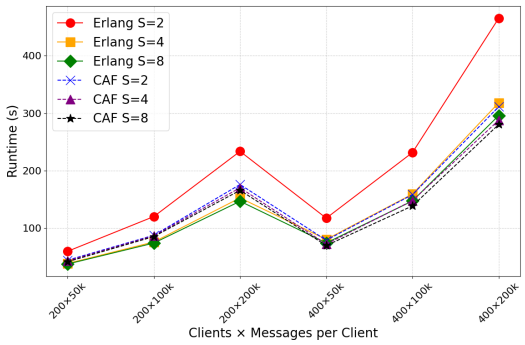
For **Prime Range = 100**, where the computation demands are higher, Erlang maintains a runtime scaling advantage as the number of servers increases, but suffers from overall longer execution times in all cases. Additionally, CAF again outper-

forms Erlang in energy efficiency, maintaining low average power even under heavy computational load, making CAF the asbolute winner of this benchmark.

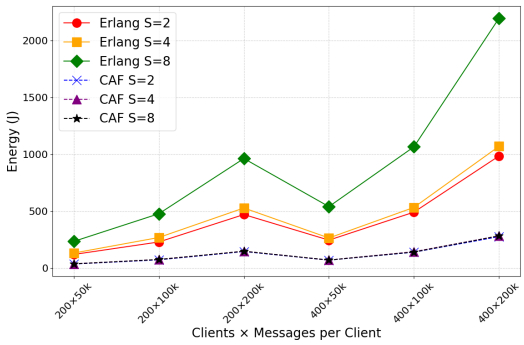
These findings emphasize an important trade-off:

- **Erlang** is favorable in communication-heavy systems, where the computational load is light and the energy constraints are secondary.
- **CAF** suits systems with high computational loads and is also suitable to minimize energy consumption.

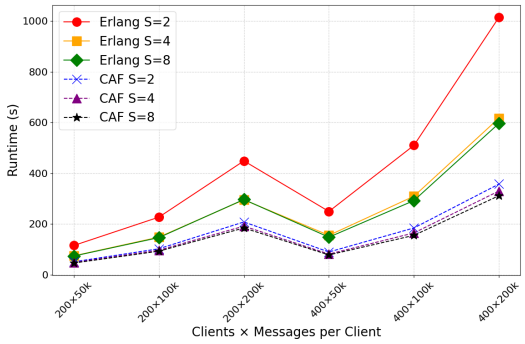
The comparison plots in Figures 5.6a–5.6d illustrate these dynamics clearly, making the impact of server count and workload size visible for both languages.



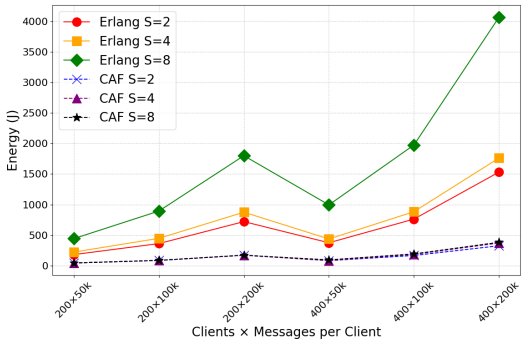
(a) Runtime at Prime Range = 50



(b) Consumption at Prime Range = 50



(c) Runtime at Prime Range = 100



(d) Consumption at Prime Range = 100

Figure 5.6: Ping-Pong with Load Balancer and Prime Calculation Comparison

5.2 Distributed Results

This section explores how each benchmark behaves in a distributed environment, where tasks are spread across multiple nodes. This setting introduces additional overhead from networking and process coordination but enables horizontal scalability. Comparisons to the native results reveal the trade-offs involved in distribution.

Distributed Ping-Pong

The following part explores the distributed results for the `ping_pong` benchmark, measured across two nodes. This setup serves as the minimal distributed deployment scenario in this study, enabling an investigation into baseline inter-node communication.

Erlang

Table 5.11 presents the results of the distributed Erlang `ping_pong` benchmark. The experiments were deployed over two Erlang nodes using distributed message passing.

Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
50	50000	8.88	2.32	20.57
50	100000	17.82	2.34	41.70
50	200000	34.91	2.42	84.27
50	500000	84.33	2.47	208.13
100	50000	15.62	2.95	46.14
100	100000	30.64	3.01	92.26
100	200000	60.34	3.01	181.74
100	500000	151.59	3.06	463.39
200	50000	30.75	3.52	108.49
200	100000	60.00	3.40	203.84
200	200000	119.04	3.47	412.97
200	500000	299.94	3.57	1070.03
400	50000	60.05	3.76	225.81
400	100000	119.37	3.84	458.03
400	200000	238.40	3.86	920.63
400	500000	591.42	4.02	2377.55

Table 5.11: Distributed Erlang `ping_pong` results

The measurements indicate a clear increase in both runtime and energy consumption as the number of clients and messages increases. Notably, even with only 50 clients, the distributed version exhibits significant overhead in runtime compared to the native implementation. This is expected given the overhead of inter-node communication, serialization, and distributed process management.

As client count and messages scale, the power consumption also grows, peaking around 4W with 400 clients and 500,000 messages. While this power draw is reasonable, it contributes to much higher energy usage due to the increased duration.

Comparison with Native

To understand the impact of distribution, we compare these results with the native measurements in Figures 5.7 and 5.8.

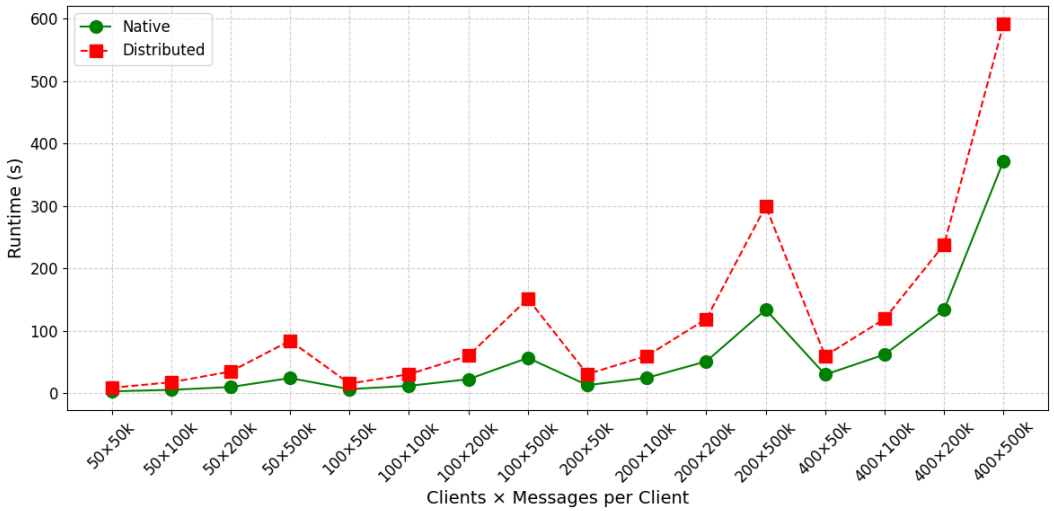


Figure 5.7: Runtime Comparison — Native vs Distributed Erlang `ping_pong`

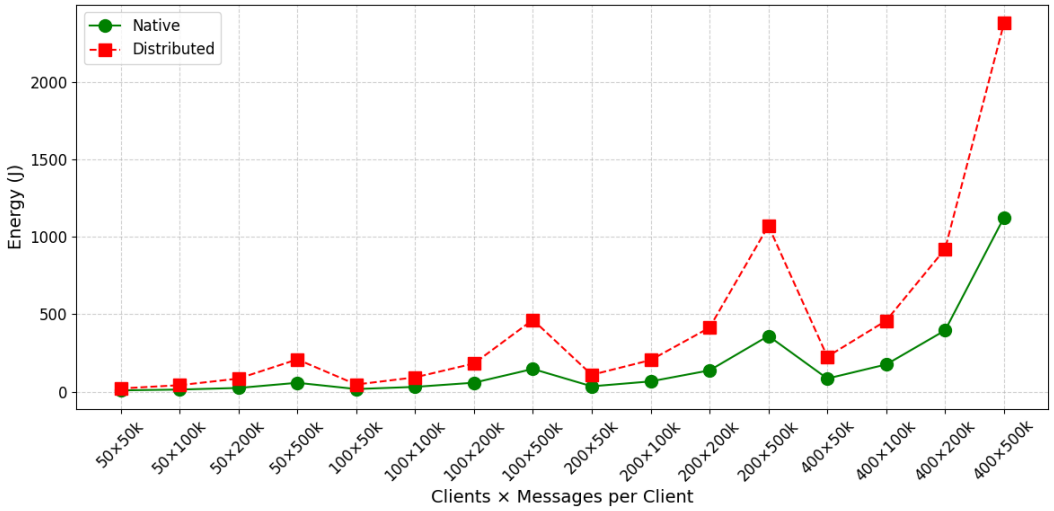


Figure 5.8: Energy Consumption Comparison — Native vs Distributed Erlang `ping_pong`

The runtime plot clearly shows up to a 2–3× overhead in the distributed case for each parameter set. For instance, with 100 clients and 100k messages, native execution takes about 12 seconds, whereas the distributed case takes over 30 seconds. Similarly, energy usage rises drastically — a consequence of both longer runtimes and increased CPU/network activity due to inter-node communication.

This illustrates the classical trade-off in distributed systems: distribution allows horizontal scaling and fault tolerance, but often introduces latency and resource overhead. Despite this, Erlang's inherent support for distribution maintains relatively acceptable scaling behavior even with this complexity.

CAF

Table 5.12 summarizes the measurements for the distributed CAF `ping_pong` scenario. The measurements were conducted across two Kubernetes nodes.

Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
50	50000	574.76	33.20	19079.89
50	100000	1142.48	35.84	40946.55
100	50000	1167.57	47.12	55020.17

Table 5.12: Distributed CAF `ping_pong` results

As evident from the results, the distributed CAF implementation exhibits extremely high runtimes and energy consumption — even with relatively low message volumes and client counts. For example, a 50-client 100k-message configuration resulted in over 1100 seconds of execution time and more than 40 kJ of energy consumption.

This drastic degradation is primarily attributed to the non-optimized nature of distributed CAF deployments over Kubernetes pods. Since CAF does not have native distribution primitives like Erlang, all inter-pod communication incurs additional serialization, connection management, and round-trip latencies.

Due to the infeasibly long execution times and high resource usage, further distributed measurements for CAF have been intentionally omitted. The same reduced scope will be used for subsequent distributed CAF cases.

Comparison with Native

The native CAF implementation performs orders of magnitude better. For example, the same 50-client 100k-message configuration in the native case runs in approximately 15 seconds (vs. 1142 seconds in distributed) and consumes only about 10.2 J of energy (vs. over 40,000 J in distributed).

This highlights the critical overhead imposed by distributed deployment when not leveraging a runtime designed for distributed messaging, as is the case with CAF. The native runtime benefits from shared memory and efficient local message passing, both of which are lost in the distributed setting.

Comparison

When comparing the distributed implementations of Erlang and CAF for the `ping_pong` benchmark, the results are clear: Erlang drastically outperforms CAF

in both runtime and energy consumption. Erlang's runtime model and native support for distributed messaging allow it to scale across nodes while still maintaining reasonable efficiency.

CAF, on the other hand, incurs extreme latency and resource usage due to a lack of distribution-aware mechanisms. The Kubernetes deployment model further aggravates these issues due to pod-to-pod networking, which introduces substantial delays and overhead.

It is worth noting that due to these severe inefficiencies, all future distributed CAF cases will be measured using a similarly reduced configuration scope. This ensures that the experiments remain manageable while still capturing the relative behavior of CAF in distributed scenarios.

Distributed Ping-Pong with Load Balancer

This part details the distributed results for the `ping_pong_lb` benchmark, executed across eight nodes. The distributed environment enables horizontal scaling by introducing a load balancer and increasing the number of concurrently running servers.

Erlang

Table 5.13 presents the final distributed results for the Erlang `ping_pong_lb` case. This experiment deployed a central load balancer and multiple worker nodes, spanning eight Erlang nodes over localhost-distributed hosts to simulate a cluster environment. Each configuration varies in server count, client count, and messages sent.

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	46.07	15.45	711.85
2	200	100000	90.40	15.55	1405.88
2	200	200000	179.91	15.72	2828.34
2	200	500000	444.91	15.79	7026.61
2	400	50000	89.91	17.09	1536.66
2	400	100000	178.25	17.18	3061.81
2	400	200000	353.30	17.27	6100.38
2	400	500000	877.81	17.45	15319.15
4	200	50000	39.33	17.66	694.47
4	200	100000	78.14	17.98	1404.84
4	200	200000	153.67	18.14	2786.71
4	200	500000	385.12	18.12	6977.54
4	400	50000	78.50	19.98	1568.30
4	400	100000	157.73	20.01	3155.41
4	400	200000	312.32	20.13	6288.46
4	400	500000	782.14	20.30	15874.75
8	200	50000	39.71	18.10	718.92
8	200	100000	79.29	18.36	1455.96
8	200	200000	157.00	18.34	2879.43
8	200	500000	392.23	18.51	7259.98
8	400	50000	82.40	20.41	1681.97
8	400	100000	165.51	20.57	3405.27
8	400	200000	331.03	20.81	6887.41
8	400	500000	819.04	20.84	17068.64

Table 5.13: Distributed Erlang `ping_pong_lb` results

Compared to the previous distributed `ping_pong` scenario, we observe a significant increase in power usage, runtime, and energy consumption due to the added complexity of the load balancer and multiple servers. For example, the 2-server, 200-client, 100k-message configuration now requires over 90 seconds and consumes 1405 J of energy, compared to 60 seconds and 204 J in `ping_pong` with the same client and message count.

As the server count increases, we do not see immediate runtime reductions. This can be attributed to the overhead of routing and context switching introduced by the load balancer. Especially in higher message count configurations, this overhead becomes significant.

Comparison with Native

To better understand the distributed overhead, Figure 5.9 and Figure 5.10 compare distributed and native runtimes and energy usage. The plots clearly show that the distributed deployment is significantly more resource-intensive.

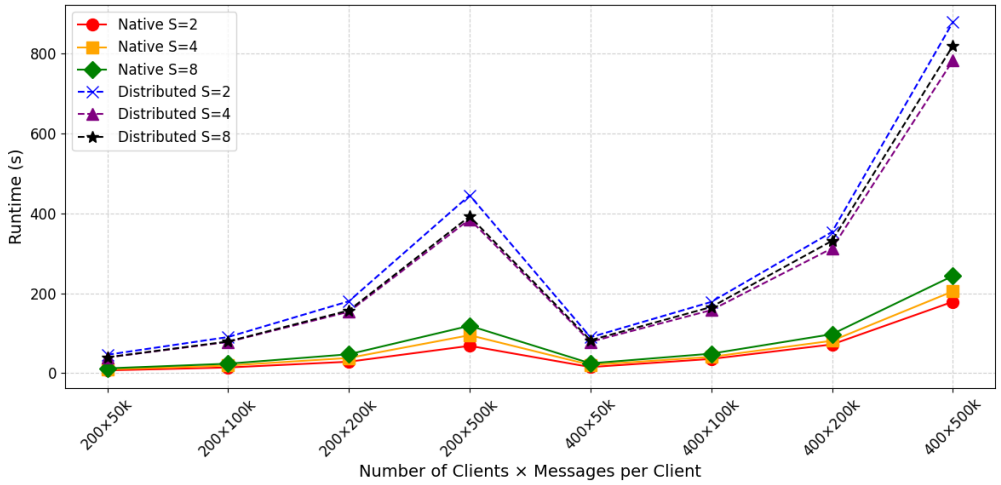


Figure 5.9: Runtime Comparison — Native vs Distributed Erlang `ping_pong_lb`

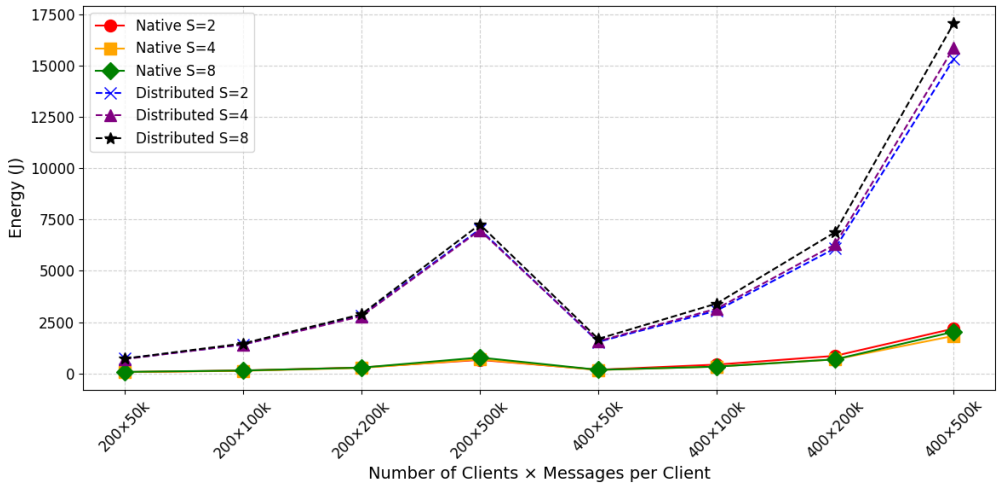


Figure 5.10: Energy Consumption Comparison — Native vs Distributed Erlang `ping_pong_lb`

In nearly all configurations, distributed runtimes are 4–6× higher than their native counterparts, and energy consumption grows proportionally. For instance, using 8 servers and 400 clients with 500k messages results in energy usage surpassing 17,000 J, while the native measurement caps around 2,000 J for the same setup.

This further validates that, while Erlang handles distribution better than CAF, deploying at scale over networked environments introduces considerable system-level overhead.

CAF

Table 5.14 presents the results of running the `ping_pong_lb` benchmark in a distributed environment using CAF across multiple containerized nodes. For this particular setup, due to the extremely long runtimes experienced in earlier configurations, only a reduced test configuration was executed with 50 clients, 50,000 messages per client, and varying server counts (2, 4, 8).

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	50	50000	659.83	43.92	28978.38
4	50	50000	750.03	44.86	33643.92
8	50	50000	982.04	46.09	45257.01

Table 5.14: Distributed CAF `ping_pong_lb` results

Compared to the native execution, distributed CAF suffers from a significant increase in both runtime and energy consumption. When comparing the minimal setups in both cases, the runtime has grown from just under 40 seconds in native to nearly 660 seconds in the distributed setup, with the energy consumption rising from under 40 J to almost 29,000 J, even though the native setup involved four times more clients. One potential explanation for this dramatic escalation involves communication overhead between containers, network delays, and orchestration latencies inherent in Kubernetes-managed deployments.

Comparison

Table 5.14 shows that the performance penalty of running CAF in a distributed environment is severe. In contrast, the distributed Erlang implementation — while still slower and more power-hungry than native — remained reasonably efficient and consistent.

This substantial difference may be attributed to several factors. Erlang's native support for distributed messaging and remote process spawning integrates seamlessly with multi-node environments. CAF, on the other hand, while supporting distributed actors, requires more explicit management of communication channels and synchronization, particularly in containerized settings.

Due to these challenges, the remainder of the distributed experiments for CAF will also be executed under minimal configurations only, in order to maintain feasibility of measurement and avoid unreasonably long benchmark durations.

Ping-Pong with Load Balancer and Prime Calculation Results

The most complex benchmark in this study, `ping_pong_lb_prime`, is executed in a distributed configuration using eight nodes. Each experiment explores how compute-heavy workloads (prime number calculations) behave when distributed across multiple servers and clients. By varying the number of messages, clients, and the range of prime computation, this benchmark highlights the interplay between parallelism and computational intensity in a message-passing distributed system.

Erlang

Table 5.15 and Table 5.16 details the results of the benchmark, `ping_pong_lb_prime`, in a distributed Erlang environment. Each configuration was executed on eight nodes using varying numbers of servers (2, 4, 8), client counts (200, 400), message counts (50k, 100k, 200k), and prime range limits (50 and 100).

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	54.03	15.85	856.57
2	200	100000	106.95	15.93	1703.33
2	200	200000	212.12	16.04	3402.65
2	400	50000	106.01	17.93	1901.14
2	400	100000	211.18	18.18	3839.58
2	400	200000	417.95	18.28	7641.21
4	200	50000	48.59	18.97	921.78
4	200	100000	95.65	19.06	1822.99
4	200	200000	190.73	19.30	3681.75
4	400	50000	94.56	22.33	2111.27
4	400	100000	192.42	22.28	4286.97
4	400	200000	385.53	22.49	8671.64
8	200	50000	50.54	22.11	1117.46
8	200	100000	101.10	22.40	2264.39
8	200	200000	202.67	22.05	4468.61
8	400	50000	98.88	26.35	2605.60
8	400	100000	196.77	26.14	5143.92
8	400	200000	396.51	25.82	10239.20

Table 5.15: Distributed Erlang `ping_pong_lb_prime` results (Prime Range = 50)

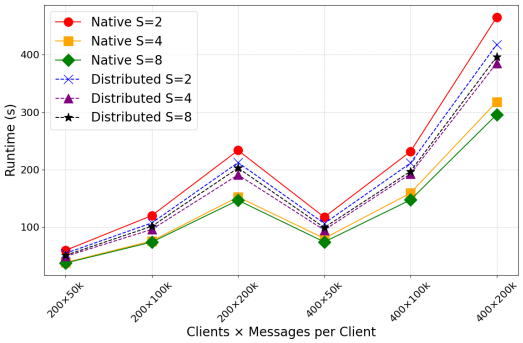
Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	200	50000	94.62	7.90	747.84
2	200	100000	187.69	8.00	1501.82
2	200	200000	373.79	8.10	3027.54
2	400	50000	183.36	12.44	2281.70
2	400	100000	365.54	12.22	4466.99
2	400	200000	729.68	12.15	8865.18
4	200	50000	60.02	18.41	1104.93
4	200	100000	118.81	18.56	2204.59
4	200	200000	237.78	18.71	4448.43
4	400	50000	114.68	22.72	2606.02
4	400	100000	230.43	21.93	5054.24
4	400	200000	460.28	21.82	10045.26
8	200	50000	59.37	23.88	1417.66
8	200	100000	118.54	23.59	2796.03
8	200	200000	236.64	23.29	5512.26
8	400	50000	110.17	27.72	3053.58
8	400	100000	220.23	27.35	6023.48
8	400	200000	449.14	27.09	12165.65

Table 5.16: Distributed Erlang `ping_pong_lb_prime` results (Prime Range = 100)

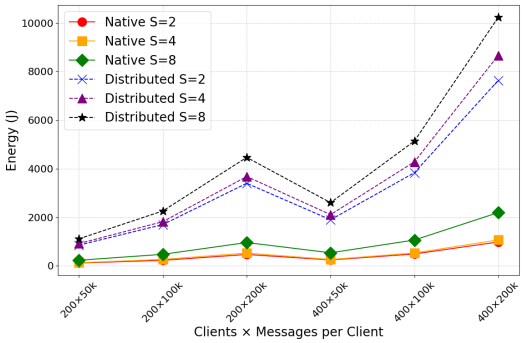
The runtime and energy consumption values for distributed `ping_pong_lb_prime` are significantly higher than in native environments. Even with 2 servers, modest client/message loads lead to execution times near or above 200 seconds, and energy usage in the kilojoule range. This trend scales steeply with the number of clients and messages. The prime range also has a noticeable impact — the `range=100` setting results in longer runtimes but lower average power usage than `range=50`, likely due to more efficient computation per message at higher ranges.

Comparison with Native Results

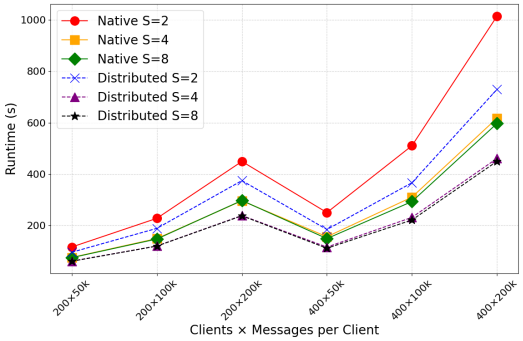
Compared to the corresponding native results, the energy overhead of distributed execution becomes stark. In the most extreme case (S=8, C=400, M=200k, R=100), energy usage jumped from 4.07 kJ to over 12.16 kJ.



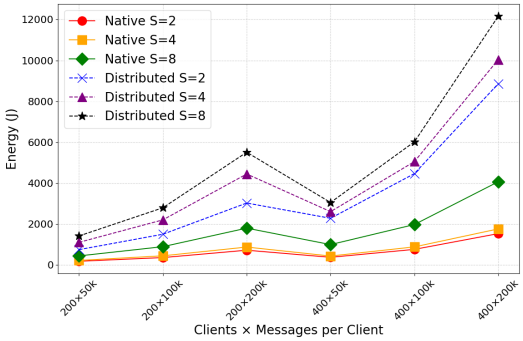
(a) Runtime at Prime Range = 50



(b) Consumption at Prime Range = 50



(c) Runtime at Prime Range = 100



(d) Consumption at Prime Range = 100

Figure 5.11: Runtime and Energy Consumption Comparison — Native vs Distributed Erlang `ping_pong_lb_prime`

Figures 5.11b and 5.11d demonstrate a substantial increase in energy consumption when transitioning from native to distributed execution in Erlang. This elevated consumption is largely attributed to the involvement of multiple nodes communicating over a network interface, even though both environments use the same logical topology. The distributed setting inherently introduces networking stack overhead and inter-node synchronization costs, both of which contribute to increased energy usage.

Interestingly, the runtime comparison reveals a more nuanced story. Figures 5.11a and 5.11c show that, while in the **Prime Range = 50** configurations the distributed version performs worse in terms of execution time, with the exception of the 2-server scenario, in the **Prime Range = 100** configurations it clearly outpaces the native version. This is likely due to better utilization of available CPU resources across nodes, especially when local scheduler contention becomes a limiting factor in the native setup. However, this advantage comes at the cost of the previously mentioned substantial energy consumption overhead.

CAF

Table 5.17 and Table 5.18 shows the results of distributed CAF executions for `ping_pong_lb_prime`, limited to a subset of the full workload due to the prohibitive runtime and energy overhead observed during testing. Configurations were executed on eight nodes, each with varying numbers of servers (2, 4, 8), a fixed client count of 50, 50k messages per client, and two prime ranges: 50 and 100.

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	50	50000	654.10	44.08	28833.88
4	50	50000	742.55	44.64	33149.88
8	50	50000	979.47	46.32	45373.05

Table 5.17: Distributed CAF `ping_pong_lb_prime` results (Prime Range = 50)

Servers	Clients	Messages	Runtime (s)	Avg Power (W)	Energy (J)
2	50	50000	653.50	43.93	28705.05
4	50	50000	742.45	44.79	33251.83
8	50	50000	981.55	46.38	45519.60

Table 5.18: Distributed CAF `ping_pong_lb_prime` results (Prime Range = 100)

Given the extremely high runtime and energy consumption encountered during initial distributed CAF experiments, especially when compared to both the native CAF and distributed Erlang results, the scope of measurement was intentionally limited to a single configuration: `clients=50` and `messages=50000`, across three server counts and two prime ranges. Even under these modest parameters, runtimes remained above 10 minutes and energy consumption reached tens of thousands of joules. These results stand in sharp contrast to the corresponding native CAF measurements, where runtimes remained below 360 seconds and energy consumption stayed under 400 J in all cases. Notably, expanding the range for Prime Calculation from 50 to 100 yielded no discernible impact on runtime or performance. This observation further supports the conclusion that the observed overhead stems primarily from communicational bottlenecks rather than computational limitations.

Comparison

Similarly to the two previous scenarios, the distributed `ping_pong_lb_prime` case showcases a stark contrast in performance and scalability between Erlang and CAF. Distributed CAF execution incurred runtimes up to an order of magnitude

higher than those of Erlang, even when the number of clients and messages was kept intentionally low.

While distributed Erlang did experience performance degradation compared to its native variant—largely due to inter-node communication and distributed scheduling—its performance remained within a reasonable factor of the original and scaled predictably with the workload. In contrast, CAF's Kubernetes based distributed mode exhibited unbounded scaling costs and failed to deliver meaningful speedups from additional nodes. Runtime and energy usage ballooned to levels that rendered further experimentation impractical.

These findings reinforce that while CAF's actor model performs adequately in native settings, its runtime infrastructure is not currently suited for large-scale distributed execution under high-concurrency message-passing workloads in a Kubernetes environment. Erlang's mature, battle-tested distribution and process management mechanisms clearly give it a significant edge in these scenarios.

Chapter 6

Related Works

The topic of energy-efficient software design has garnered significant attention across diverse areas of computing, from programming language behavior to distributed system architectures and emerging IoT technologies. In this chapter, a selection of prior research is reviewed to establish the broader context for this thesis. The discussion covers studies on language-level energy efficiency, energy measurement and optimization in distributed environments, as well as considerations at the system and hardware levels. These insights provide a foundational background and help position the present investigation into the energy behavior of actor-based distributed frameworks.

6.1 Energy Efficiency in Programming Languages

Recent research has increasingly focused on the relationship between programming language characteristics and energy efficiency.

Analyzing the Energy Consumption of Programming Languages

Georgiou, Kechagia, and Spinellis conducted an empirical study to investigate the energy consumption characteristics of a wide variety of programming languages, including compiled, semi-compiled, and interpreted ones [28]. Using a set of tasks extracted from the Rosetta Code repository, they measured both runtime performance and energy consumption across multiple languages on a Linux environment, utilizing external hardware tools for direct energy measurement.

Their findings show that compiled languages such as Go and C tend to offer higher energy efficiency compared to semi-compiled (e.g., Java, C#) and interpreted languages (e.g., Python, PHP, Ruby). Interestingly, optimization flags (e.g., `-O3` for GCC) often significantly reduced energy consumption, although in some cases they had the opposite effect, demonstrating that compiler optimizations need to be applied carefully.

Furthermore, they observed that while runtime performance often correlates with energy efficiency, there are important exceptions, particularly with languages like Rust, which demonstrated relatively high energy consumption despite competitive runtime performance. This work highlights the importance of empirical energy measurements when selecting a programming language for energy-sensitive applications. These findings support the methodological approach of this thesis, which similarly emphasizes real-world measurements rather than relying solely on assumptions about language efficiency.

Energy Behavior of Functional Languages

In a separate line of research, Lima et al. explored the energy efficiency of programs written in Haskell, a purely functional and lazily evaluated programming language [29]. Their study conducted more than 20,000 benchmark executions, focusing on two critical dimensions: strictness (eager vs. lazy evaluation) and concurrency primitives.

The results reveal that small changes in concurrency structures or evaluation strategies can yield significant differences in energy consumption. For example, switching between different concurrency primitives like `MVar` and `TMVar` resulted in energy savings up to 60%, depending on the benchmark. However, the relationship between energy consumption and runtime performance was found to be non-linear: in some cases, the fastest implementation also had the worst energy efficiency.

They also extended standard Haskell benchmarking and profiling tools (such as Criterion and GHC's profiler) to collect energy consumption data using the RAPL interface. This extension enabled a more granular analysis of how energy usage is distributed across different parts of a Haskell program. These observations are relevant for this thesis, as Erlang — while also rooted in functional programming — employs a strict evaluation strategy and an actor-based concurrency model, offering

a distinct perspective on energy behavior.

Together, these studies emphasize that programming language design choices can have substantial impacts on energy consumption, and that energy profiling tools are crucial for developers aiming to build sustainable and efficient software.

6.2 Energy Efficiency in Distributed Systems and Cost Modeling

The growing importance of energy efficiency and cost-aware computing in distributed systems has led to extensive research across different layers of system architecture. In this section, two relevant studies are reviewed: one focusing on energy measurement techniques and optimization strategies in distributed computing systems, and the other on cost modeling in virtualized cloud environments based on actual energy consumption.

Energy Efficiency in Large-Scale Distributed Systems

The work by Trobec et al. addresses the growing energy consumption challenges in large-scale distributed systems such as clusters, grids, and clouds [30]. The paper highlights that energy inefficiency arises at multiple layers of system architecture, including the hardware, middleware, networking, and application levels. The authors emphasize that accurate, reliable, and continuous power consumption measurement is a fundamental prerequisite for introducing energy efficiency in distributed environments. They classify existing energy measurement techniques into hardware-based, software-based, and hybrid approaches, and present a real-world case study measuring power consumption in a multi-core computing system using Intel's Performance Counter Monitor (PCM) library. Notably, their experimental results demonstrate that minimizing application runtime—by fully utilizing all available CPU cores at maximum frequency—can be more energy-efficient than slower execution with fewer active cores. These findings underline the complex interplay between performance and energy consumption and provide a valuable context for evaluating actor-based applications' behavior under different system loads and configurations.

Energy-Based Cost Models in Cloud Environments

Aldossary and Djemame propose an energy-based cost model tailored for virtualized cloud environments, where energy consumption has become a significant operational cost [31]. Their model accounts for both idle and active power consumption at the virtual machine (VM) level, attributing the physical machine's energy usage proportionally to each VM based on its resource utilization. They rely mainly on CPU utilization as the primary indicator of energy consumption, based on previous findings that CPU load correlates strongly with overall system power. Their experiments, conducted on a cloud testbed, show that the model effectively differentiates the energy and cost contributions of heterogeneous VMs. The model also supports more accurate billing and resource management strategies by considering real energy use instead of relying solely on predefined tariffs. The principles outlined in their model directly reinforce the importance of precise energy measurements employed in this thesis when evaluating distributed execution scenarios.

6.3 Energy Efficiency in IoT Systems and Hardware Implementations

Recent research has expanded the discussion of energy efficiency to both large-scale IoT ecosystems and the direct comparison of hardware versus software algorithm implementations. This section explores two studies that offer insights into the challenges and opportunities for improving energy consumption across different technological domains.

Energy-Efficient Solutions for IoT-Based Applications

Singh and Chana present a comprehensive analysis of energy consumption issues in IoT systems and propose multiple strategies for achieving greater energy efficiency [32]. They highlight that while IoT devices offer significant potential for saving energy through features like remote monitoring and smart automation, the vast number of devices and their supporting infrastructure can paradoxically contribute to high overall energy consumption. The paper categorizes the IoT workflow into four stages—sensing, networking, analysis, and storage—and identifies energy-demanding factors at each stage, such as continuous sensing, data transmission, and

cloud storage. To address these challenges, they propose solutions including energy harvesting, periodic data transfer, data filtration, local computation through edge and fog computing, and more efficient scheduling and routing protocols. This system-level view on optimizing communication and computation patterns strengthens the rationale for evaluating actor-based languages and frameworks like Erlang and CAF in energy-sensitive distributed settings.

Energy Efficiency of Hardware Versus Software Algorithm Implementations

Kirkeby et al. investigate the comparative energy efficiency of hardware and software implementations of two fundamental algorithms: Heapsort and Dijkstra's shortest path algorithm [33]. Their experimental setup includes C-based software running on a Raspberry Pi and Chisel-based hardware implementations on an FPGA. They emphasize the importance of fair measurement techniques, employing external measurement tools to capture whole-system energy consumption rather than relying solely on CPU-level indicators. Their findings reveal that for algorithms with inherent local parallelism, such as Heapsort, hardware implementations can offer significant energy savings compared to software. However, for algorithms like Dijkstra's, which involve more sequential dependencies, the hardware did not provide clear energy or performance advantages. These results suggest that the nature of the algorithm—specifically its parallelization potential—plays a critical role in determining whether hardware acceleration is energy efficient. This insight directly informs the exploration of actor-based models in distributed systems, where fine-grained concurrency can either amplify or mitigate energy trade-offs depending on the computational pattern.

Chapter 7

Conclusions

This thesis presented a detailed comparative analysis of energy efficiency in actor-based applications implemented with Erlang and the C++ Actor Framework (CAF), across both native and distributed environments. The study began with an overview of green computing principles, actor-based concurrency, and prior work in energy-aware software systems (Chapter 2). It then described the experimental methodology, including the hardware setup, benchmarking procedures, and post-processing tools used to filter and aggregate power data (Chapter 3). Chapter 4 outlined the design and implementation of benchmark applications that reflect realistic communication and computation patterns in actor systems. Using this foundation, Chapter 5 conducted a detailed quantitative analysis of runtime, power draw, and energy consumption under varying workloads and configurations. The discussion was further contextualized through a survey of related literature (Chapter 6), highlighting how this study extends existing knowledge into distributed and concurrent execution contexts. Through this consistent benchmarking approach and custom-built measurement infrastructure, several important insights were uncovered regarding the energy behavior and performance trade-offs of actor-based frameworks.

In native settings, CAF demonstrated a clear advantage in energy efficiency across all benchmarks. Although its runtimes were generally longer in pure communication-heavy scenarios, its significantly lower average power consumption more than compensated for this, resulting in lower total energy usage compared to Erlang. Erlang, on the other hand, achieved faster runtimes in native communication benchmarks, highlighting the efficiency of its lightweight process model in messaging-dominated workloads.

When computational complexity was introduced, CAF began to outperform Erlang in both runtime and energy consumption. This suggests that CAF's design offers better CPU-bound performance characteristics, making it more suitable for applications combining computation and messaging.

In distributed deployments, the situation reversed. Erlang's built-in distribution mechanisms allowed it to maintain relatively efficient performance and acceptable energy profiles, even across multiple nodes. CAF, however, suffered significant degradation when containerized and orchestrated over Kubernetes, leading to much longer runtimes and drastically higher energy consumption. The overheads introduced by containerization, networking, and orchestration overwhelmed CAF's inherent advantages seen in native execution.

Overall, the findings reveal an important trade-off:

- In native environments, **CAF** offers superior energy efficiency, and becomes even more favorable as workloads shift toward computational intensity.
- In distributed environments, **Erlang** remains the more practical choice, offering better scalability, faster execution, and lower energy consumption overall.

Future Work: This work opens up several new paths for investigation, including:

- Investigating optimizations for CAF in containerized distributed deployments.
- Extending measurements to heterogeneous environments, including real-world cloud clusters.
- Exploring actor frameworks with native containerization support to minimize orchestration overhead.
- Studying energy behavior under dynamic scaling, load balancing, and failure recovery scenarios.

This work highlights that energy-efficient design in actor-based systems depends critically not just on programming model and language choices, but also on deployment architecture, workload characteristics, and the surrounding execution environment.

A paper summarizing the main achievements of this thesis is submitted for publication at the Workshop on Performance and Energy Efficiency in Concurrent and Distributed Systems, 2025.

Appendix A

Measurement Workflow Example

A.1 Erlang Example

```
$ sudo python3 ../../../scripts/measure_linux.py ping_pong main \
  "50 50000" \
  "50 100000" \
  --rep 2 \
  --file ping_pong_erl_results \
  --folder measurements \
  --nano 1000000
[INFO] Starting measurement for parameter '50 50000' (Repetition 1/2)
[INFO] Compiling Erlang module: ping_pong.erl
[INFO] Running Erlang command: erl -noshell -run ping_pong main 50
    ↪ 50000 -s init stop
[INFO] Starting program execution and timing...
[INFO] Execution time: 3.0323 seconds
[INFO] Reading Scaphandre output from /home/bszirtes/University/
    ↪ THESIS/THESIS_GIT/src/native/ping_pong_erl/measurements/
    ↪ ping_pong__main__2025-04-26-15-22-56.json
[INFO] Sum of consumption samples: 109407543.30000001 µW
[INFO] Number of samples: 41
[INFO] Average energy per sample: 2668476.6658536587 µW
[INFO] Final energy consumption for this run: 8091610.476857657 µJ
[INFO] Results saved to ping_pong_erl_results.csv
```

```
19 ------------------------------
20 [INFO] Starting measurement for parameter '50 50000' (Repetition 2/2)
21 [INFO] Running Erlang command: erl -noshell -run ping_pong main 50
      ↪ 50000 -s init stop
22 [INFO] Starting program execution and timing...
23 [INFO] Execution time: 3.3243 seconds
24 [INFO] Reading Scaphandre output from /home/bszirtes/University/
      ↪ THESIS/THESIS_GIT/src/native/ping_pong_erl/measurements/
      ↪ ping_pong__main__2025-04-26-15-23-09.json
25 [INFO] Sum of consumption samples: 110871005.10000001 µW
26 [INFO] Number of samples: 44
27 [INFO] Average energy per sample: 2519795.570454546 µW
28 [INFO] Final energy consumption for this run: 8376489.612563986 µJ
29 [INFO] Results saved to ping_pong_erl_results.csv
30 ------------------------------
31 [INFO] Starting measurement for parameter '50 100000' (Repetition
      ↪ 1/2)
32 [INFO] Compiling Erlang module: ping_pong.erl
33 [INFO] Running Erlang command: erl -noshell -run ping_pong main 50
      ↪ 100000 -s init stop
34 [INFO] Starting program execution and timing...
35 [INFO] Execution time: 6.0674 seconds
36 [INFO] Reading Scaphandre output from /home/bszirtes/University/
      ↪ THESIS/THESIS_GIT/src/native/ping_pong_erl/measurements/
      ↪ ping_pong__main__2025-04-26-15-23-23.json
37 [INFO] Sum of consumption samples: 215029260.4 µW
38 [INFO] Number of samples: 83
39 [INFO] Average energy per sample: 2590713.9807228916 µW
40 [INFO] Final energy consumption for this run: 15718892.856301067 µJ
41 [INFO] Results saved to ping_pong_erl_results.csv
42 ------------------------------
43 [INFO] Starting measurement for parameter '50 100000' (Repetition
      ↪ 2/2)
```

```
[INFO] Running Erlang command: erl -noshell -run ping_pong main 50
   ↪ 100000 -s init stop
[INFO] Starting program execution and timing...
[INFO] Execution time: 5.8191 seconds
[INFO] Reading Scaphandre output from /home/bszirtes/University/
   ↪ THESIS/THESIS_GIT/src/native/ping_pong_erl/measurements/
   ↪ ping_pong__main__2025-04-26-15-23-39.json
[INFO] Sum of consumption samples: 183437172.09999996 µW
[INFO] Number of samples: 80
[INFO] Average energy per sample: 2292964.6512499996 µW
[INFO] Final energy consumption for this run: 13342927.231423836 µJ
[INFO] Results saved to ping_pong_erl_results.csv
------------------------------
```

Code A.1: Command Line Output

```
Module,Function,Parameter,Runtime (s),Samples,Average (µW),
   ↪ Consumption (µJ),Sample frequency (ns),File
ping_pong,main,50 50000,3.0322957589996804,41,2668476.6658536587,
   ↪ 8091610.476857657,1000000,ping_pong__main__2025-04-26-15-22-56
ping_pong,main,50 50000,3.324273489000916,44,2519795.570454546,
   ↪ 8376489.612563986,1000000,ping_pong__main__2025-04-26-15-23-09
ping_pong,main,50 100000,6.067398012000922,83,2590713.9807228916,
   ↪ 15718892.856301067,1000000,ping_pong__main__2025
   ↪ -04-26-15-23-23
ping_pong,main,50 100000,5.819072362999577,80,2292964.6512499996,
   ↪ 13342927.231423836,1000000,ping_pong__main__2025
   ↪ -04-26-15-23-39
```

Code A.2: Result File

```
{
    "consumers" : [
       {
          "cmdline" : "/usr/lib/erlang/erts-13.2/bin/beam.smp---root
             ↪ /usr/lib/erlang-bindir/usr/lib/erlang/erts-13.2/bin-
```

```
            ↪ prognameerl---home/root---noshell-
            ↪ runping_pongmain50100000-sinitstop",
         "consumption" : 2309952.2,
         "container" : null,
         "exe" : "/usr/lib/erlang/erts-13.2/bin/beam.smp",
         "pid" : 33206,
         "resources_usage" : null,
         "timestamp" : 1745673814.52941
      },
      {
         "cmdline" : "scaphandrejson--step0--step-nano1000000--
            ↪ process-regex=/beam.smp-fmeasurements/
            ↪ ping_pong__main__2025-04-26-15-23-23.json",
         "consumption" : 461990.44,
         "container" : null,
         "exe" : "/usr/bin/scaphandre",
         "pid" : 33174,
         "resources_usage" : null,
         "timestamp" : 1745673814.52945
      },
      {
         "cmdline" : "/bin/sh-cscaphandre json --step 0 --step-nano
            ↪  1000000 --process-regex=\\/beam.smp -f measurements
            ↪ /ping_pong__main__2025-04-26-15-23-23.json",
         "consumption" : 0,
         "container" : null,
         "exe" : "/usr/bin/dash",
         "pid" : 33173,
         "resources_usage" : null,
         "timestamp" : 1745673814.52941
      }
   ],
   "host" : {
      "components" : {
```

```
            "disks" : [
               {
                  "disk_available_bytes" : "13965635584",
                  "disk_file_system" : "overlay",
                  "disk_is_removable" : false,
                  "disk_mount_point" : "/var/lib/docker/overlay2/
                     ↪ fd15c6f6786815895d2e3b57a4d4e48e0951c601e79
                     ↪ bd9ae28ad44679a0ede94/merged",
                  "disk_name" : "overlay",
                  "disk_total_bytes" : "48891670528",
                  "disk_type" : "Unknown"
               }
            ]
         },
         "consumption" : 31085356,
         "timestamp" : 1745673814.52964
      },
      "sockets" : [
         {
            "consumption" : 22064284,
            "domains" : [
               {
                  "consumption" : 0,
                  "name" : "uncore",
                  "timestamp" : 1745673814.46654
               },
               {
                  "consumption" : 18543512,
                  "name" : "core",
                  "timestamp" : 1745673814.46653
               }
            ],
            "id" : 0,
            "timestamp" : 1745673814.46586
```

```
        }
      ]
   },
```

Code A.3: Raw Measurement Sample

A.2 CAF Example

```
$ sudo python3 ../../../scripts/measure_linux.py ping_pong "" \
  "--clients=50 --messages=50000" \
  "--clients=50 --messages=100000" \
  --rep 2 \
  --file ping_pong_caf_results \
  --folder measurements \
  --exe ping_pong \
  --nano 1000000
[INFO] Starting measurement for parameter '--clients=50 --messages
    ↪ =50000' (Repetition 1/2)
[INFO] Compiling C++ program: ping_pong.cpp
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bszirtes/University/THESIS
    ↪ /THESIS_GIT/src/native/ping_pong_caf
Consolidate compiler generated dependencies of target ping_pong.out
[100%] Built target ping_pong.out
[INFO] Running C++ command: ./ping_pong.out --clients=50 --messages
    ↪ =50000
[INFO] Starting program execution and timing...
[INFO] Execution time: 5.5909 seconds
[INFO] Reading Scaphandre output from /home/bszirtes/University/
    ↪ THESIS/THESIS_GIT/src/native/ping_pong_caf/measurements/
    ↪ ping_pong____2025-04-26-15-28-57.json
[INFO] Sum of consumption samples: 66810329.379999965 µW
[INFO] Number of samples: 97
```

```
[INFO] Average energy per sample: 688766.2822680409 µW
[INFO] Final energy consumption for this run: 3850798.2572317496 µJ
[INFO] Results saved to ping_pong_caf_results.csv
------------------------------
[INFO] Starting measurement for parameter '--clients=50 --messages
   ↪ =50000' (Repetition 2/2)
[INFO] Running C++ command: ./ping_pong.out --clients=50 --messages
   ↪ =50000
[INFO] Starting program execution and timing...
[INFO] Execution time: 5.8392 seconds
[INFO] Reading Scaphandre output from /home/bszirtes/University/
   ↪ THESIS/THESIS_GIT/src/native/ping_pong_caf/measurements/
   ↪ ping_pong____2025-04-26-15-29-12.json
[INFO] Sum of consumption samples: 67171213.28999996 µW
[INFO] Number of samples: 104
[INFO] Average energy per sample: 645877.0508653843 µW
[INFO] Final energy consumption for this run: 3771428.638077707 µJ
[INFO] Results saved to ping_pong_caf_results.csv
------------------------------
[INFO] Starting measurement for parameter '--clients=50 --messages
   ↪ =100000' (Repetition 1/2)
[INFO] Compiling C++ program: ping_pong.cpp
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bszirtes/University/THESIS
   ↪ /THESIS_GIT/src/native/ping_pong_caf
Consolidate compiler generated dependencies of target ping_pong.out
[100%] Built target ping_pong.out
[INFO] Running C++ command: ./ping_pong.out --clients=50 --messages
   ↪ =100000
[INFO] Starting program execution and timing...
[INFO] Execution time: 16.3086 seconds
[INFO] Reading Scaphandre output from /home/bszirtes/University/
   ↪ THESIS/THESIS_GIT/src/native/ping_pong_caf/measurements/
```

```
   ↪ ping_pong____2025-04-26-15-29-28.json
[INFO] Sum of consumption samples: 178624804.85999998 µW
[INFO] Number of samples: 252
[INFO] Average energy per sample: 708828.5907142856 µW
[INFO] Final energy consumption for this run: 11560033.334365074 µJ
[INFO] Results saved to ping_pong_caf_results.csv
------------------------------
[INFO] Starting measurement for parameter '--clients=50 --messages
   ↪ =100000' (Repetition 2/2)
[INFO] Running C++ command: ./ping_pong.out --clients=50 --messages
   ↪ =100000
[INFO] Starting program execution and timing...
[INFO] Execution time: 16.2135 seconds
[INFO] Reading Scaphandre output from /home/bszirtes/University/
   ↪ THESIS/THESIS_GIT/src/native/ping_pong_caf/measurements/
   ↪ ping_pong____2025-04-26-15-29-55.json
[INFO] Sum of consumption samples: 180665407.05000004 µW
[INFO] Number of samples: 256
[INFO] Average energy per sample: 705724.2462890627 µW
[INFO] Final energy consumption for this run: 11442227.826194588 µJ
[INFO] Results saved to ping_pong_caf_results.csv
------------------------------
```

Code A.4: Command Line Output

```
Module,Function,Parameter,Runtime (s),Samples,Average (µW),
   ↪ Consumption (µJ),Sample frequency (ns),File
ping_pong,,--clients=50 --messages
   ↪ =50000,5.590863485000227,97,688766.2822680409,
   ↪ 3850798.2572317496,1000000,ping_pong____2025-04-26-15-28-57
ping_pong,,--clients=50 --messages
   ↪ =50000,5.839236171999801,104,645877.0508653843,
   ↪ 3771428.638077707,1000000,ping_pong____2025-04-26-15-29-12
ping_pong,,--clients=50 --messages
   ↪ =100000,16.308644270000514,252,708828.5907142856
```

```
    ↪ ,11560033.334365074,1000000,ping_pong____2025-04-26-15-29-28
ping_pong,,--clients=50 --messages
    ↪ =100000,16.21345431499867,256,705724.2462890627,
    ↪ 11442227.826194588,1000000,ping_pong____2025-04-26-15-29-55
```

Code A.5: Result File

```
{
     "consumers" : [
        {
           "cmdline" : "./ping_pong.out--clients=50--messages
              ↪ =100000",
           "consumption" : 588996.4,
           "container" : null,
           "exe" : "/home/bszirtes/University/THESIS/THESIS_GIT/src/
              ↪ native/ping_pong_caf/ping_pong.out",
           "pid" : 35292,
           "resources_usage" : null,
           "timestamp" : 1745674190.01574
        },
        {
           "cmdline" : "scaphandrejson--step0--step-nano1000000--
              ↪ process-regex=/ping_pong-fmeasurements/
              ↪ ping_pong____2025-04-26-15-29-28.json",
           "consumption" : 392664.25,
           "container" : null,
           "exe" : "/usr/bin/scaphandre",
           "pid" : 35260,
           "resources_usage" : null,
           "timestamp" : 1745674190.01568
        },
        {
           "cmdline" : "/bin/sh-cscaphandre json --step 0 --step-nano
              ↪  1000000 --process-regex=\\/ping_pong -f
              ↪ measurements/ping_pong____2025-04-26-15-29-28.json",
```

```
            "consumption" : 0,
            "container" : null,
            "exe" : "/usr/bin/dash",
            "pid" : 35259,
            "resources_usage" : null,
            "timestamp" : 1745674190.0157
         },
         {
            "cmdline" : "/bin/sh-c./ping_pong.out --clients=50 --
               ↪ messages=100000",
            "consumption" : 0,
            "container" : null,
            "exe" : "/usr/bin/dash",
            "pid" : 35291,
            "resources_usage" : null,
            "timestamp" : 1745674190.01566
         }
      ],
      "host" : {
         "components" : {
            "disks" : [
               {
                  "disk_available_bytes" : "13965627392",
                  "disk_file_system" : "overlay",
                  "disk_is_removable" : false,
                  "disk_mount_point" : "/var/lib/docker/overlay2/
                     ↪ fd15c6f6786815895d2e3b57a4d4e48e0951c601e79
                     ↪ bd9ae28ad44679a0ede94/merged",
                  "disk_name" : "overlay",
                  "disk_total_bytes" : "48891670528",
                  "disk_type" : "Unknown"
               }
            ]
         },
```

```
54          "consumption" : 29057156,
55          "timestamp" : 1745674190.01578
56       },
57       "sockets" : [
58          {
59             "consumption" : 15620672,
60             "domains" : [
61                {
62                   "consumption" : 0,
63                   "name" : "uncore",
64                   "timestamp" : 1745674189.96566
65                },
66                {
67                   "consumption" : 11398277,
68                   "name" : "core",
69                   "timestamp" : 1745674189.96565
70                }
71             ],
72             "id" : 0,
73             "timestamp" : 1745674189.96517
74          }
75       ]
76    },
```

Code A.6: Raw Measurement Sample

Bibliography

[1] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009. ISBN: 9780596518189.

[2] Dominik Charousset et al. "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments". In: *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* New York, NY, USA: ACM, 2013, pp. 87–96.

[3] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. "Revisiting Actor Programming in C++". In: *Computer Languages, Systems & Structures* 45 (2016), pp. 105–131.

[4] IBM Cloud Education Team. *What is green computing?* Accessed April 12, 2025. 2022. URL: https://www.ibm.com/think/topics/green-computing.

[5] Rui Pereira et al. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[6] Áron Attila Mészáros et al. "Towards Green Computing in Erlang". In: *Studia Universitatis Babeș-Bolyai, Informatica* 63.1 (2018). Accessed April 13, 2025, pp. 64–79. DOI: 10.24193/subbi.2018.1.05. URL: https://doi.org/10.24193/subbi.2018.1.05.

[7] Gergely Nagy et al. "Tools supporting green computing in Erlang". In: *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang*. Erlang 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 30–35. ISBN: 9781450368100. DOI: 10.1145/3331542.3342570. URL: https://doi.org/10.1145/3331542.3342570.

[8] Youssef Gharbi, Melinda Tóth, and István Bozó. *Green Computing for Erlang*. SQAMIA 2024: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 9–11, 2024, Novi Sad, Serbia. 2024. URL: https://github.com/joegharbi/greenErl.

[9] Youssef Gharbi, Melinda Tóth, and István Bozó. *Analysing the Energy Usage of the Erlang BEAM*. 3rd workshop on Resource AWareness of Systems and Society (RAW 2024). 2024. URL: https://github.com/joegharbi/SQAMIA_GREEN_JIT.

[10] Youssef Gharbi, Melinda Tóth, and István Bozó. "Measuring and Analysing Erlang's Energy Usage". In: *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*. 2024, pp. 1993–1998. DOI: 10.1109/MIPRO60963.2024.10569977.

[11] Hubblo. *Scaphandre Documentation*. Accessed April 14, 2025. 2024. URL: https://hubblo-org.github.io/scaphandre-documentation/.

[12] Hubblo. *Scaphandre Documentation - JSON exporter*. Accessed April 14, 2025. URL: https://hubblo-org.github.io/scaphandre-documentation/references/exporter-json.html.

[13] Hubblo. *Scaphandre Documentation - Prometheus exporter*. Accessed April 14, 2025. URL: https://hubblo-org.github.io/scaphandre-documentation/references/exporter-prometheus.html.

[14] Hubblo. *Scaphandre Documentation - Metrics exposed by Scaphandre*. Accessed April 14, 2025. URL: https://hubblo-org.github.io/scaphandre-documentation/references/metrics.html.

[15] Hubblo. *Scaphandre Documentation - Explanations about host level power and energy metrics*. Accessed April 14, 2025. URL: https://hubblo-org.github.io/scaphandre-documentation/explanations/host_metrics.html.

[16] The C++ Actor Framework (CAF) Authors. *CAF Documentation - Concepts*. Accessed April 12, 2025. URL: https://actor-framework.readthedocs.io/en/stable/intro/Concepts.html.

[17] The C++ Actor Framework (CAF) Authors. *CAF Documentation - Actors*. Accessed April 12, 2025. URL: https://actor-framework.readthedocs.io/en/stable/core/Actors.html.

[18] The C++ Actor Framework (CAF) Authors. *CAF Documentation - Message Passing*. Accessed April 12, 2025. URL: https://actor-framework.readthedocs.io/en/stable/core/MessagePassing.html.

[19] The C++ Actor Framework (CAF) Authors. *CAF Documentation - Message Handlers*. Accessed April 12, 2025. URL: https://actor-framework.readthedocs.io/en/stable/core/MessageHandlers.html.

[20] The C++ Actor Framework (CAF) Authors. *CAF Documentation - Middleman*. Accessed April 12, 2025. URL: https://actor-framework.readthedocs.io/en/latest/io/NetworkTransparency.html.

[21] The Kubernetes Authors. *Kubernetes Documentation - Overview*. Accessed April 13, 2025. 2024. URL: https://kubernetes.io/docs/concepts/overview/.

[22] The Kubernetes Authors. *Kubernetes Documentation - Pods*. Accessed April 13, 2025. 2025. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[23] The Kubernetes Authors. *Kubernetes Documentation - Nodes*. Accessed April 13, 2025. 2024. URL: https://kubernetes.io/docs/concepts/architecture/nodes/.

[24] The Kubernetes Authors. *Kubernetes Documentation - Kubernetes Components*. Accessed April 13, 2025. 2024. URL: https://kubernetes.io/docs/concepts/overview/components/.

[25] The Kubernetes Authors. *Kubernetes Documentation - Service*. Accessed April 13, 2025. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[26] The Kubernetes Authors. *KinD Documentation - Overview*. Accessed April 13, 2025. 2025. URL: https://kind.sigs.k8s.io/.

[27] The Kubernetes Authors. *KinD Documentation - Configuration*. Accessed April 13, 2025. 2025. URL: https://kind.sigs.k8s.io/docs/user/configuration/.

[28] Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. "Analyzing Programming Languages' Energy Consumption: An Empirical Study". In: *Proceedings of the 21st Pan-Hellenic Conference on Informatics*. PCI '17. Larissa, Greece: Association for Computing Machinery, 2017. ISBN: 9781450353557. DOI: 10.1145/3139367.3139418. URL: https://doi.org/10.1145/3139367.3139418.

[29] Luís Gabriel Lima et al. "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 517–528. DOI: 10.1109/SANER.2016.85.

[30] R. Trobec et al. "Energy efficiency in large-scale distributed computing systems". In: *2013 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2013, pp. 253–257.

[31] Mohammad Aldossary and Karim Djemame. "Energy-based Cost Model of Virtual Machines in a Cloud Environment". In: *2018 Fifth International Symposium on Innovation in Information and Communication Technology (ISIICT)* (2018), pp. 1–8.

[32] Utkarsha Singh and Inderveer Chana. "Energy Efficient Solutions for IoT Based Applications". In: *2019 2nd International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*. Vol. 1. 2019, pp. 1287–1292. DOI: 10.1109/ICICICT46008.2019.8993351.

[33] Maja Kirkeby et al. "Energy-efficiency of software and hardware algorithms". In: *Computer Science and Information Systems* (Jan. 2025), pp. 28–28. DOI: 10.2298/CSIS240914028K.

List of Figures

List of Tables

List of Codes