画像処理向け耐障害性並列処理パイプラインの実装と評価

Implementation and Evaluation of Fault-tolerant Parallel Processing Pipeline for Image Processing

2024年03月

北九州市立大学大学院 国際環境工学研究科

ヨウ　サイケイ

（指導教員：山崎　進）

概 要

医療や災害検出などの画像処理を行う場合，処理性能と耐障害性が求められる．本研究では，処理性能と耐障害性の両立する OSS である Broadway と，画像処理 OSS である Evision を利用した画像処理パイプラインの構築を検討した．Supervisor がプロセスを監視して再起動することで，このような耐障害性を実現する．しかし，Evision に含まれるネイティブコードの実行が異常終了した場合，たとえ Supervisor の監視下でも，処理系全体が終了してしまい，耐障害性が損なわれるという技術的課題があることを我々は発見した．そこで本研究では，Broadway と，Supervisor 監督下の Linux プロセスを生成してネイティブコードを実行し，分散コンピューティング機構である Node を用いて通信を行うことでネイティブコード実行の耐障害性を実現する OSS である SpawnCoElixir を組みわせることを提案する．提案手法が設計意図通りネイティブコード障害時の耐障害性を備えていることを実証するために，ネイティブコードに Fault-Injection を埋め込むことで異常終了をシミュレートした．また障害から回復するまでに必要な処理時間も計測した．Fault-Injection を用いた異常終了シミュレーションの評価結果では，設計通りに障害復旧できることを確認した．また，障害復旧に必要な処理時間の平均は 284.79ns と極めて短かった．将来課題としては，ネイティブコードで途中までデータ処理を実行できた場合に，ネイティブコード中での障害発生時にデータ消失する問題を解決することが挙げられる．

Abstract

Processing performance and fault tolerance are required when performing image processing for medical or disaster detection purposes. In this research, we considered building an image-processing pipeline using Broadway, an OSS that achieves both processing performance and fault tolerance, and Evision, an image-processing OSS. A supervisor monitors and restarts the observed process to achieve such fault tolerance. However, we found a technical issue: to lose fault tolerance by terminating the entire processing system even though Evision is under the Supervisor if the execution of the native code included in Evision terminates abnormally. Therefore, this research proposes a combination of Broadway and SpawnCoElixir, an OSS, to achieve fault tolerance of native code execution by spawning a Linux process under the Supervisor to execute the native code and communicating using Node, a distributed computing mechanism. In order to demonstrate that the proposed method has fault tolerance against native code faults as intended, we simulated abnormal termination by embedding a fault injection in the native code. We also measured the processing time required to recover from the fault. The evaluation results of an abnormal termination simulation using the fault injection confirmed that the native code could recover from faults as designed. The average processing time required for failure recovery was extremely short at 284.79ns. Our future works include solving the problem of data loss in the event of a fault in native code, if data processing can be executed partway through the native code.

目 次

第1章　はじめに

1.1　背景

近年，画像処理技術は現代では非常に普及しており，私たちの日常生活のさまざまな局面で広く活用されている．たとえば，医療画像，衛星画像分析，物体認識，スペクトル分析などである．また，画像処理の精度を向上させるために，機械学習を使う画像処理の大規模化・複雑化が顕著に進んでいる [1].

このような画像処理では，処理性能と耐障害性が求められるため，高速化を実現する 1 つの手段としてマルチコア等を使った並列処理が重要となる．ただし，画像処理のシステムにおいて，マルチコアプロセッサシステム上でコードを実行しても，自動的にコードが効率化されるわけではない．また，並行・並列プログラミングにおいて, デッドロックや性能低下の問題がある [2].

画像処理のソフトウェア全体で最適な並列化を実現するため，山崎らが Elixir に着目し，画像処理のデファクトスタンダードである Python と比べて Elixir の潜在的優位性を示した [3].

画像処理に Elixir をより効率的に使うために，Nx と Evision が開発された．Nx は高速的に行列変換に対処できる．Evision は画像処理に対処できる．

しかし，Nx と Evision を使って，巨大な画像を処理すると，メモリ不足などによる異常終了になることがある．特に Evision と Nx のアクセラレータである EXLA が NIF 関数を使用しており，NIF 関数が異常終了すると，Erlang VM 全体が異常終了になる可能性がある．

このように多様かつ複雑なアプリケーションは画像処理システムの異常終了により，極端な場合，人命や巨額の金銭が危険にさらされる可能性がある．

したがって，このような画像処理システムの構築において耐障害性はかなり重要な部分である．

画像処理において耐障害性の高いパイプラインを構築するために，山崎進は 2022 年の ElixirConf で巨大な画像に対する堅牢な分散並列処理について講演し，HtPipe を提案した [5]．HtPipe は NIF 関数が異常終了することで，Erlang VM 全体が異常終了するのは回避できるが，並列処理で関数の作成と呼び出しには Broadway を使用する方が簡潔で効率的である．また，Broadway ではデータの生成部分と処理部分が分離されており，一時的に一部の機能が失われたとしても，データの流れは整然と処理される．

したがって，そのようなパイプラインには，まだ改善すべき部分が存在していると考えられる．このような背景から，我々は，処理性能と耐障害性の両立する OSS である Broadway と，画像処理 OSS である Evision を利用した画像処理パイプラインの構築を検討した.

1.2　目的

この研究の目標は，画像処理を行う Broadway パイプラインを構築し，その中で分散コンピューティング機構である Node を用いて 通信を行うことでネイティブコード実行の耐障

害性を実現する OSS である SpawnCoElixir を用いることで，耐障害性を向上させたパイプラインの開発と評価を行うことである．

1.3 アプローチ

1.3.1 研究手段

本研究では，処理性能と耐障害性の両立する OSS である Broadway と，画像処理 OSS である Evision を利用した画像 処理パイプラインの構築に取り組む．

研究手段として, Broadway と Evision というオープンソースソフトウェア（OSS）を選定し, これらを組み合わせて画像処理パイプラインを構築する．そん中には, Broadway を処理性能と耐障害性を兼ね備えた基盤として利用し, Supervisor を介してプロセスを監視して再起動することで一定の耐障害性を確保する．

また, Evision 内の NIF 関数が異常終了した際の耐障害性向上のために, SpawnCoElixir という OSS を導入した．この手法では, Broadway と Supervisor 監視下の Linux プロセスを生成し, Node を通じて通信することで, NIF が異常終了しても，起動元の Erlang VM に波及することはないことを目指する．

1.3.2 評価手法

研究手段の一環として, 提案手法の有効性を検証するためには, 評価手法を設定する．評価手法は以下通りである．

(1) Broadway パイプラインが画像処理での有効性を検証する為に, 複数の画像処理段階を統合して実行できるかどうか確認する．

(2) 画像処理プロセス途中でメモリ不足などによる異常終了することに対応できるかどうかを確認する為に, スタックを取り出す pop(), スタックに要素を追加する push() とプ ロセスが終了される abort_soft() を定義し, Supervisor の監視の下で push()，abort_soft()，pop() を順番に実行して，スタック値がクリアされているかどうかを観察することで, 提案された Broadway パイプラインが画像処理プロセス途中でメモリ不足などによる異常終了することに対応できるかどうかを確認する．

(3) 提案した Broadway パイプラインで画像処理の時 NIF 関数が異常終了すると， Erlang VM 全体が異常終了するのは回避できるかどうかを確認する為に, SpawnCoElixir を 実装なしの Broadway パイプラインと実装された Broadway パイプライン中に最初の段階 で第 3 章で記述した abort をシミュレートする abort_NIF 関数を呼び出して，Broadway パイプラインの状態を観察することで, 提案された Broadway パイプラインが NIF 関数の異常状態を処理する際, Erlang VM 全体が異常終了を回避できるかどうかを確認する．

1.4 論文の構成

本論文の構成は，以下のとおりである．2 章に本手法を実装するうえで基となる Broadway, Node, SpawnCoElixir と実行時間を測定する Benchee について述べる．3 章において，画像処理を行う Broadway パイプラインの提案と実装方法について説明し，第 4 章で実装成果の評価結果を示す．第 5 章で本稿をまとめる．

第2章　関連研究

2.1　Broadway

Broadway は，Elixir チームによってデータパイプラインを作成および管理できるツールである．イベントと，メトリクス，自動応答，障害処理などの運用機能に重点を置いている [4]．本章では，2.1.1 節で，Broadway の耐故障性能の基盤技術となる Broadway の Supervisor 動作について簡単に述べたうえで，2.1.2 節で Broadway におけるデータ処理パイプラインを作成するための構成要素 GenServer について述べる．

2.1.1　Broadway の Supervisor 動作

Broadway の Supervisor は職場でスーパーバイザーが従業員のグループに対して責任を負うのと同じように, 割り当てられたプロセスに対して責任を負う．下位プロセスも上司に報告する必要があり, 上司はすべてがスムーズに実行されていることを確認する必要がある．これを実現するために, スーパーバイザには, 他のプロセスを効果的に管理できる一連の機能が付属している． プロセスを開始および停止したり, システム内で予期しないエラーが発生した場合にはプロセスを再起動したりできる．これらは終了をトラップするように構成されているため, 監視対象プロセスがエラーで終了すると, そのエラーは隔離され, それ以上伝播することはないである．

Broadway の Supervisor 動作は，主に以下の部品により 構成されており，それぞれの関係を図 2.1 に表す．

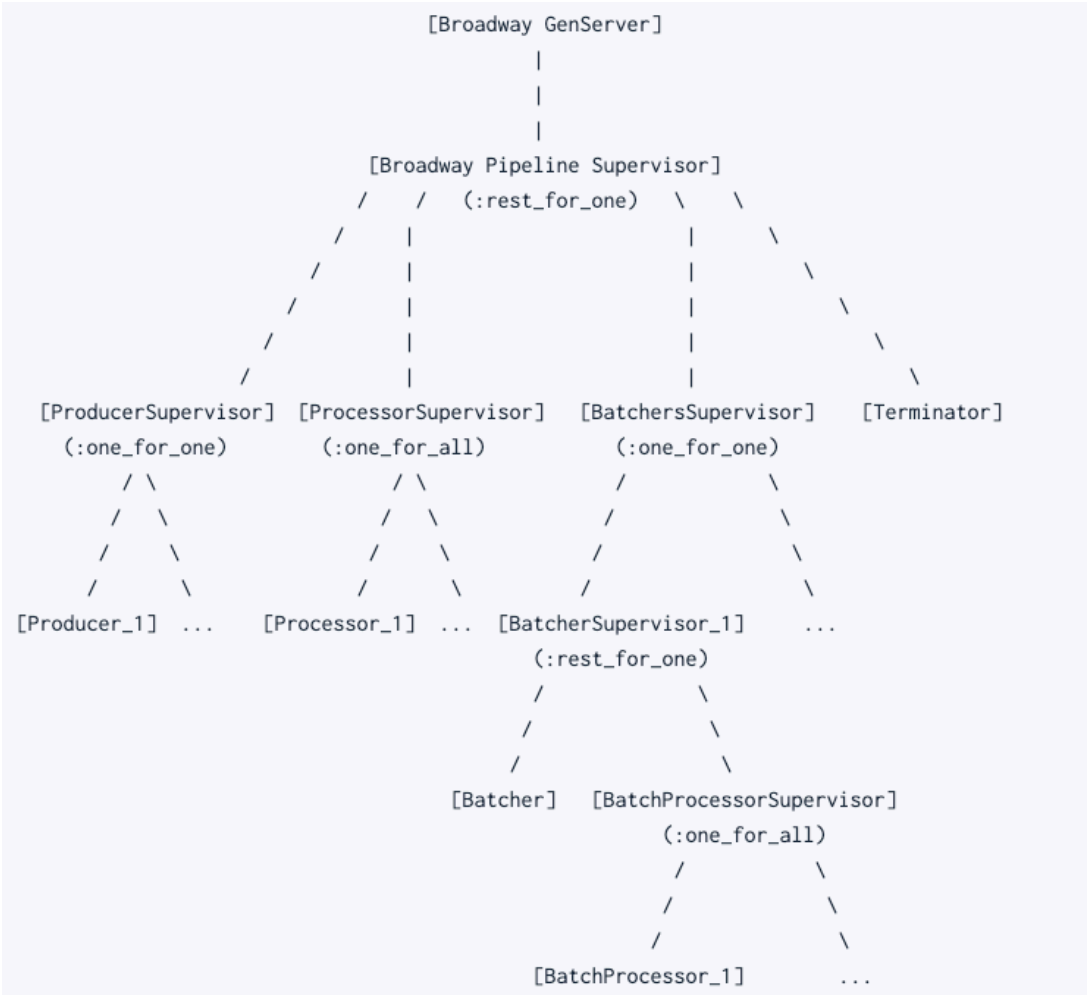

図 2.1: Broadway の内部構造図

ツリーの最上部には, プロジェクト スーパーバイザーである Broadway Pipeline Supervisor があり,**application.ex** ファイルで定義されている.

```
def start(_type, _args) do
children = [
# Starts a worker by calling: Broadway.worker.start_link(arg)
]
# See https://hexdocs.pm/elixir/Supervisor.html
# for other strategies and supported options
opts = [strategy: :one_for_one, name: Broadway.Supervisor]
Supervisor.start_link(children, opts)
end
```

Broadway Pipeline Supervisor は,Broadway のさまざまなコンポーネントをすべて監視し, エラーが発生した場合は再起動する機能を持っている. Broadway Pipeline Supervisor が監視するプロセスとその目的を以下に示す.

- **ProducerSupervisor**: データ プロデューサー プロセスの監視を担当する. この監視ツリーには, 問題が発生している特定のデータ プロデューサーのみを再起動する必要があるため, :one_for_one の戦略が採用されている. すべてが問題なければ, 他のすべてのプロ

デューサーは実行を続けることができる．

- **ProcessSupervisor**: プロデューサーからのデータを消費するワーカー プロセスを監視する責任がある．この監視ツリーには,:one_for_all の戦略がある．これが :one_for_one ではない理由は作成する処理コールバック関数がステートレスである必要があり, 発生したエラーはプロセスをクラッシュさせることなく処理できるためである．エラーが発生してプロセスがクラッシュした場合は,Broadway に関連する内部簿記の一部に問題が発生している可能性があ, すべてのコンシューマーを再起動する必要がある．
- **BatchersSupervisor**: バッチを動的に監視するために使用できる．この監視ツリーには,:one_for_all の戦略がある．これが :one_for_one ではない理由は ProcessSupervisor と同じである．
- **Terminator: Broadway** パイプラインを適切に停止する責任がある．これは, すべてのコンシューマ プロセスに, 終了後にプロデューサに再サブスクライブしてはならないことを通知する．また，すべてのプロデューサに対して, 現在のイベントをすべてフラッシュし, 後続のデータ要求を無視するように通知する．

Broadway Pipeline Supervisor には :rest_for_one という監視ポリシーがあることも重要である． その理由は, プロデューサー監視ツリーがクラッシュした場合, 親 Supervisor は後続のすべての監視ツリーを再起動し, パイプラインを動作中の新しい状態に復元できるためである．

これらすべてにより,Supervisor は耐故障性能の基盤技術となる．

2.1.2 Broadway の GenServer 動作

GenServer(generic server の略) は, クライアントとサーバーの関係のサーバーを実装するための動作モジュールである．状態を保持したり, コードを非同期に実行したりするために使用されるだけではない．また, 標準的なインターフェイス関数セットがあり, トレースおよびエラー報告の機能も含まれる [11].

GenServer モジュールは,GenServer の動作に必要ないくつかの関数のデフォルト実装を提供する．これらの関数はコールバックとして機能する．コールバックを実装するときは, 次の 2 つのことを知っておく必要がある．

- コールバック関数が受け取る引数
- どのような戻り値がサポートされている

GenServer の動作に関する次のコールバック関数について説明する．

- **handle_call/3**

handle_call/3 は, クライアントから GenServer プロセスへの同期呼び出しを処理するために使用される．

- **handle_cast/2**

handle_cast/2 は, 通常は他のプロセスまたはタイマーによってトリガーされる非同期イベントを処理するために使用される．

- **init/1**

init/1 は,GenServer プロセスの状態を初期化するために使用される必須のコールバック関数である． これはプロセスの開始時に呼び出され, 通常はいくつかの初期化操作を実行するために使用され, プロセスの初期状態を含むタプルを返す.

これら 3 つのコールバック関数が一緒になって GenServer のコア動作を形成し, 同期および非同期メッセージを処理できるようになり,Broadway パイプラインが同時実行および

並列実行を処理できるようになる．また,Broadway パイプラインをカスタマイズできる．更にバックグラウンドでも実行される存続期間の長いプロセスを作成し, より優れた制御と柔軟性を提供する．

2.2 Node

Node は，Elixir が Erlang 言語から継承するライブラリ関数である [8]．Node により Elixir は Erlang の強力な分散機能へアクセス可能である [10]．

Node には，作成された各ノードは実行中の Erlang ランタイム システムとなる．ランタイム システムには, 同時実行性, 分散, フォールト トレランスのサポートが組み込まれている．

これによって,Elixir コンテキストでの配布は, 言語の多くの機能がコードを変更せずにネットワーク上で動作することを意味する． 同時に, プロセスやメッセージ パッシングなどの基本的なプリミティブとプロセス リンクやモニターなどのより高度な概念も含まれる．

相互に接続する Erlang VM インスタンスは,Erlang ノードのクラスターと呼ばれる． Erlang ノードがクラスターに接続すると, その ID が他のすべてのノードに伝達され,Erlang ノードと他のすべてのノードの間に別のネットワーク接続が設定される．これは、いわゆる全結合メッシュになる．すべてのノードは他のすべてのノードに接続されているため, クラスター内の合計接続数は二次関数的に増加する．接続されたノードのセットのすべての接続の合計 c は

$$\sum_{c=1}^{n-1} c = \frac{n(n-1)}{2}$$

で定義される．通常のポイントツーポイント トラフィックに加えて,Erlang ノードは各ノード接続上でティックと呼ばれるハートビート メッセージを送信し, リモートノードがまだ生きているかどうかを確認する．

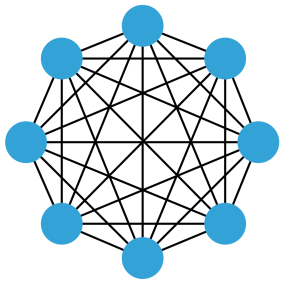

図 2.2: Node 通信網

この一般的なアプローチにより, 高い拡張性と耐障害性を備えたネットワーク ノードを構築できる． 分散 Erlang の元の設計は, プライベートの安全なネットワーク内で実行されることを目的としていた． したがって, デフォルトでは Erlang のノード間通信は暗号化されず [9], 共有秘密 (Cookie) によってのみ保護される．異なるネットワーク内のノード間でメッセージを送信する時, 同じ Cookie で起動されたノードのみが相互に正常に通信できる．

2.3 SpawnCoElixir

SpawnCoElixir は，山崎進によって書かれた Elixir のライブラリ関数である．Node と GenServer を使用して監視される協力的な Elixir ノードを起動，停止し，管理するためのコードを提供している [12].

本稿は，次の特徴を基づいてマルチノード フォールト トレランスに SpawnCoElixir を利用する．

動的なノードの生成と管理:

SpawnCoElixir は，動的なノードの生成と管理をできる．CoElixir ワーカーは，異なるノード上で動作できる．

プロセスの監視と再起動:

CoElixir ワーカープロセスは DynamicSupervisor で監視されており，プロセスがクラッシュした場合に再起動が行われる．これにより，ワーカーの耐障害性が向上できる．

Elixir プロセスモデルの活用:

SpawnCoElixir モジュールは Elixir プロセスモデルを活用しており，軽量かつ並列処理が容易である．これにより，大規模かつ複雑な分散システムの構築がしやすくなる．

動的なコードの実行:

SpawnCoElixir モジュールでは，動的にコードを指定して CoElixir ワーカーを生成できる．これにより，柔軟でカスタマイズ可能な実行環境を提供している。

2.4 Benchee

Benchee は Elixir での簡単で優れた (マイクロ) ベンチマークのためのライブラリである. Benchee を使用すると, さまざまなコード部分のパフォーマンスを一目で比較できる. また, 関数だけに依存する多用途性と拡張性も備えている [13].

実際のコード実行環境に近づけるために，Benchee は, 最初のウォームアップ後に各関数を一定時間実行し, 実行時間と必要に応じてメモリ消費量を測定する. 次に, 平均, 標準偏差などのさまざまな統計値が表示される.

次の統計データ指標を説明する.

average : 平均実行時間/メモリ使用量である.

ips : 1 秒あたりの反復数, 別名, 指定された関数を 1 秒以内に実行できる頻度である.

deviation : 標準偏差である.

median : すべての測定値を並べ替えたときの中央の値である.

99 % : 最悪の場合のパフォーマンスである.

Benchee を使用すると，信頼性が高く，再現性があり，分析が簡単なパフォーマンス テストを作成できる. そして, Benchee を通じて取得されたデータは, Elixir コードのパフォーマンスを客観的に測定および比較するために使用できる.

第3章　提案手法

3.1　提案手法の概要

今回提案する一つ手法は，Elixir 言語で画像処理に Broadway のフォールトトレランスと並列処理を適用することを目指すというものである．Broadway は，スケーラブルなデータストリーム処理システムを構築するための Elixir のライブラリである．この提案は，特に画像の読み取り，画像の分割，保存操作などの画像バッチ処理にブロードウェイを使用することを目的としている．

既存手法からの変更点は，画像処理に Broadway を使用すると，並列処理プログラムがより簡潔に記述され，画像の流れがより明確になり得る．また，Broadway パイプラインで画像タスクを処理するときに監視および回復メカニズムにより，システムの耐障害性が向上し，プロセッサのクラッシュによる画像処理の中断が軽減される．

も一つ手法は，Elixir のライブラリ関数である SpawnCoElixir の動作における親と子のノード間のメッセージ通信を介して，クラッシュの原因となる NIF 関数は子ノードで呼び出されて，このようにして，子のノードがクラッシュした後でも，親ノードが影響を受けない．これによって，NIF が異常終了しても，起動元の Erlang VM に波及することはないである．

3.2　実装例

以下で画像処理やマルチノード耐障害における Broadway パイプラインについて述べる．

3.2.1　パイプラインの基本構成

パイプラインの基本構成:
def start_link(_opts) do
Broadway.start_link(_ __MODULE__ _,
name: _ _MODULE___,
producer: [
module: {Counter, 1},
transformer: {_ __MODULE__ _, :transform, [ ]},
concurrency: 1
],
processors: [
default: [concurrency: 4]
],
batchers: [
sqs: [concurrency: 2, batch_size: 10],

s3: [concurrency: 1, batch_size: 10]
]
)
end

このコードは，Elixir の Broadway フレームワークを使用してパイプラインを開始するための関数である．以下に、このコードの主な機能と構造を説明する．

Broadway.start_link/2 関数: これは Broadway パイプラインを起動するための関数である．

name: ___MODULE___: Broadway パイプラインに名前を付けている．

producer: パイプラインの最初のステージであるプロデューサモジュールに関する設定である．

transformer: {___MODULE___, :transform, [ ]}: トランスフォーマモジュールに関する設定である．

concurrency: 1, concurrency: 4: パイプライン内の各ステージの並列処理の設定である．

processors: パイプライン内のプロセッサモジュールに関する設定である．

batchers: パイプライン内のバッチャモジュールに関する設定である．

この関数を呼び出すことで，Broadway パイプラインが設定され，各ステージが連携してデータ処理が行われる．

3.2.2 画像処理におけるパイプラインの実装

画像の読み取り:

def init(_a) do
img1 = "https://upload.wikimedia.org/wikipedia/en/7/7d/Lenna_%28test_image%29.png"
|> HTTPoison.get!()
|> then(&(&1.body))
|> Evision.imdecode(Evision.Constant.cv_IMREAD_COLOR())
{:producer, img1} end

この init/1 関数は Broadway パイプラインの初期化ステップを定義している．以下はこの関数の主な機能と構造を説明する．

img1 の定義: この部分ではウェブ上の画像を URL からダウンロードして，それを Elixir で扱える形式に変換している．

HTTPoison.get!/1: HTTPoison ライブラリを使用して指定された URL から画像を取得する．

then/2: then/2 は前のステップの結果を取り，関数を適用する．

Evision.imdecode/2: 画像のバイナリデータを Evision ライブラリを使用してデコードする．

:producer, img1: パイプラインの初期化の最後に，:producer, img1 というタプルを返す．

この **init/1** 関数は，Broadway パイプラインが始まる際に一度だけ呼び出され，初期化のためのデータが設定される．

閾値を指定して画像の二値化:

{threshold, mono_img} =
src_file

|> Evision.imread(flags:
Evision.Constant.cv_IMREAD_GRAYSCALE())
|> Evision.threshold(127, 255,
Evision.Constant.cv_THRESH_BINARY())

このコードは Evision ライブラリを使用して画像のしきい値処理を行っている．以下はこのコードの主な機能と各部分の説明である．

Evision.imread/1: src_file で指定されたファイルを Evision ライブラリを使用して読み込む．

Evision.Constant.cv_IMREAD_GRAYSCALE/1：画像をグレースケールで読み込む．

Evision.threshold/3: 読み込まれたグレースケール画像に対して，しきい値処理を行う．127 が閾値の下限で，255 が閾値の上限である．

このコードは，指定された画像ファイルを読み込んでグレースケールに変換し，さらにしきい値処理を行っている．

画像の分割:

```
def handle_demand(demand, img1) when demand > 0 do
src_file = "Lenna.png"
Evision.imwrite(src_file, img1)
div_size = 256
dst_file_ext = Path.extname(src_file)
dst_file_basename = Path.basename(src_file, dst_file_ext)
dst_files =
Stream.unfold(0, fn counter -> counter, demand - 9 end)
|> Stream.map(&"#dst_file_basename_#{&1}#{dst_file_ext}")
div_img =
mono_img
|> Evision.Mat.to_nx()
|> Nx.to_batched(div_size)
|> Enum.map(&Evision.Mat.from_nx_2d(&1))
z_img =
Enum.zip(div_img, dst_files)
|> Enum.to_list()
{:noreply, z_img, dst_files}
end
```

この **handle_demand/2** 関数は，Elixir の GenStage のハンドラ関数である．以下はこの関数の主な機能と各部分の説明である．

Evision.imwrite/2: この関数を使用して img1（しきい値処理された画像）を "Lenna.png" に保存する．

div_size: 画像を分割する際のサイズです．256 として設定されている．

dst_files: 各数字を "Lenna_数字.png" のような形式のファイル名に変換する．これにより、分割された画像を保存するためのファイル名リストが作成される．

div_img: mono_img を Nx 形式に変換し，指定されたサイズでバッチ処理した後，それを元に画像データを再構築する．

z__img: div_img と dst_files を組み合わせて，それぞれの分割画像と対応するファイル名をタプルで持つリストを生成する．

最終的には、:noreply とともに生成されたデータ（z_img と dst_files）が返される．このデータは GenStage の次のステップに渡される．

コマンドラインに出力:

```
def handle_message(:default, message, _context) do
message.data
|> Enum.map(fn img, dst_file -> Evision.imwrite(dst_file, img) end)
|> IO.inspect
end
```

この **handle_message/3** 関数は Elixir の GenStage のハンドラ関数である．この関数は**:default** メッセージを処理する．以下はその機能の説明である．

message.data: メッセージのデータ部分を取得する．

Enum.map/2: メッセージのデータ部分（img と dst_file のタプル）を取り出し，それぞれの組み合わせに対して **Evision.imwrite(dst_file, img)** を実行する．これにより，各分割画像がそれに対応するファイル名で保存される．

IO.inspect/1: 最終的に生成されたリストを出力する．

この関数は **:default** メッセージを処理し，画像と対応するファイル名のリストを使って画像を保存する役割を果たしている．

3.2.3 耐障害性を用いるパイプラインの実装

SpawnCoElixir の構成オプションを定義する．

```
options = [
{:code, "abort_nif()"},
{:deps, []},
{:host_name, "host"},
{:co_elixir_name, "co_elixir"}
]
{:ok, _} = SpawnCoElixir.run(options)
```

options には 4 つの構成項目がある．Elixir コード，依存関係，ホスト ノード名プレフィックス，および CoElixir ノード名プレフィックスが含まれている．これらの構成項目は **SpawnCoElixir.run/1** 関数に渡され，Supervisor の監督で子ノードを起動し，渡された関数を実行する．

3.2.4 abort をシミュレートする NIF 関数の実装

```
#include <stdlib.h>
#include <erl_nif.h>
static ERL_NIF_TERM abort_nif(ErlNifEnv *env, int argc, const ERL_NIF_TERM
argv[])
{
abort();
}
```

```
static ErlNifFunc nif_funcs [] =
{
{"abort_nif", 0, abort_nif}
};
ERL_NIF_INIT(Elixir.AbortNif, nif_funcs, NULL, NULL, NULL, NULL)
```

このコードは，**Erlang NIF (Native Implemented Function)** と呼ばれる，C 言語で書かれた Erlang の拡張機能を作成する．abort_nif 関数は，C 言語の abort() 関数を呼び出している．これにより，Erlang プロセスがクラッシュし，Erlang VM が終了する．以下はコードの説明である．

#include <stdlib.h>: 標準ライブラリの stdlib.h をインクルードしている．これは **abort()** 関数が定義されているヘッダーファイルである．

#include <erl_nif.h>: Erlang NIF のためのヘッダーファイルをインクルードしている．

static ERL_NIF_TERM abort_nif(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[]): abort_nif 関数が宣言されている．この関数は，Erlang から呼び出されると abort() 関数を実行し，プロセスをクラッシュさせる．引数として env（Erlang の実行環境），argc（引数の数），argv（引数のリスト）を取りる．

static ErlNifFunc nif_funcs []：使用可能な NIF 関数のリストが宣言されている．この例では，abort_nif 関数のエントリが含まれている．

ERL_NIF_INIT(Elixir.AbortNif, nif_funcs, NULL, NULL, NULL, NULL): NIF の初期化が行われる．これにより，NIF ライブラリが Erlang VM に登録され，Erlang コードから呼び出すことができる．

この NIF は，Erlang からの要求で abort() 関数を呼び出して Erlang VM を強制的にクラッシュさせるための関数である．

第 4 章　評価実験

本章では，提案する Broadway パイプラインについて安全性・回復速度の 2 つの観点から性能を評価する．まず 4.1 節で，実験を行う際に用いた環境について述べる．そして 4.2 節で提案手法の評価について述べ，4.3 節で障害の発生による耐故障処理の実験と Broadway の性能について述べる．

4.1　実験環境

実験を行っ た環境は以下の通りである．

- OS : macOS Sonoma ver 14.1.1
- CPU : Apple M1
- メモリ 16GB
- Elixir 言語コンパイラ : Elixir 1.15.4
- Erlang 言語コンパイラ : Erlang/OTP 26 [erts-14.0.2]
- Broadway バージョン : 1.0.7
- Evision バージョン : 0.1.2
- Nx バージョン : 0.3.0
- SpawnCoElixir バージョン : 0.3.0

次の構成で実行されるベンチマーク スイート:

- warmup: 2 s
- time: 5 s
- memory time: 0 ns
- reduction time: 0 ns
- parallel: 1
- inputs: none specified
- Estimated total run time: 7 s

4.2　評価手法

提案する Broadway パイプラインの評価実験は, 安全性・回復速度の 2 つの観点から実施する．以下でそれぞれの項目について説明する．

安全性:

安全性の検証手法として, 実際にメモリ不足などによる異常終了が発生した場合, 再起動ができないかどうか実験を行う．

回復速度:

Benchee を使用して, 実際にメモリ不足などによる異常終了が発生した場合, 回復速度をテストを行う．

実験は 3 つの部分に分かれている．

- 4.3.1 節ではパイプを使用して通常のタスクを処理する． パイプラインの可用性を確認する．
- 4.3.2 節では，FI を使用して，子プロセスをシャットダウンし，パイプラインで呼び出す関数 1 を作成する．タスクが異常終了したときの状況を実際にシミュレートする．プロセスの異常なシャットダウンに対する回復機能を確認してから, 回復速度をテストを行う．
- 4.3.3 節では，FI を使用して，abort を伴う NIF 関数を作成し，パイプラインで呼び出す．NIF 関数の使用時に発生する abort をシミュレートする．NIF 関数が abort された後，起動元の Erlang VM に波及することを確認する．

4.3 Broadway パイプラインの実験

4.3.1 画像処理における Broadway パイプラインの可用性

まず，**HTTPoison.get!()** コマンドを使用して，必要なイメージをダウンロードする．コマンドラインでは画像ファイルを表示できないため，実験結果をより直感的に感じられるよう，Livebook を使用してパイプライン実験の過程をセクションごとに表示する．

図 4.1 は未処理の元の画像である．

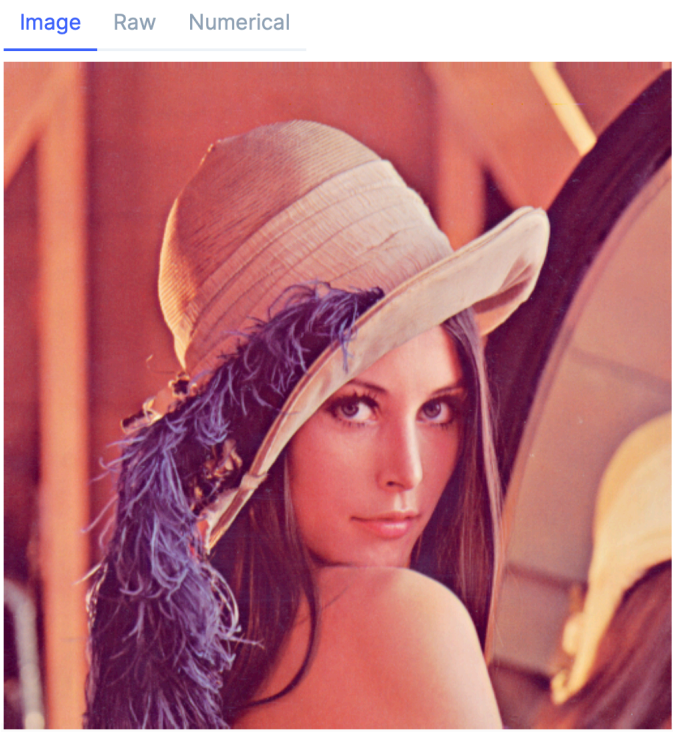

図 4.1: 未処理の元の画像

次に、コマンド ラインで **iex -S mix** コマンドを使用して，Broadway パイプラインを開始する．パイプラインは連続しているため，図 4.2 に示すように最終出力のみが表示される．

```
root@84df33344504:/home/broadway_img# iex -S mix
Compiling 1 file (.ex)
Erlang/OTP 26 [erts-14.0.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1]
[jit]

Interactive Elixir (1.15.4) - press Ctrl+C to exit (type h() ENTER for help)
{%Evision.Mat{
   channels: 3,
   dims: 2,
   type: {:u, 8},
   raw_type: 16,
   shape: {256, 512, 3},
   ref: #Reference<0.3038945523.3844210696.115578>
 }, "Lenna_0.png"}
{%Evision.Mat{
   channels: 3,
   dims: 2,
   type: {:u, 8},
   raw_type: 16,
   shape: {256, 512, 3},
   ref: #Reference<0.3038945523.3844210696.115579>
 }, "Lenna_1.png"}
```

図 4.2: Broadway パイプライン最終出力

図 4.2 は，処理した画像の内容と画像名を示している．コマンドラインでは，リストを使用して処理された画像を表示する．パイプラインを閉じると，図 4.3 に示すように，処理された画像ファイルがフォルダーの下に表示される．

```
root@84df33344504:/home/broadway_img# ls
Lenna.png  Lenna_0.png  Lenna_1.png  README.md  _build  deps  lib
 mix.exs  mix.lock  test
```

図 4.3: 出力画像ファイル

パイプラインでのバッチ処理後，図 4.3 から，ファイルに Lenna_0 と Lenna_1 の 2 つの画像ファイルが含まれている．同じ方法を使用して両方の画像を別々に表示する．

```
1  img = Evision.imread("Lenna_0.png")
```

Image Raw Numerical

図 4.4: Lenna_0

```
1   img = Evision.imread("Lenna_1.png")
```

Image Raw Numerical

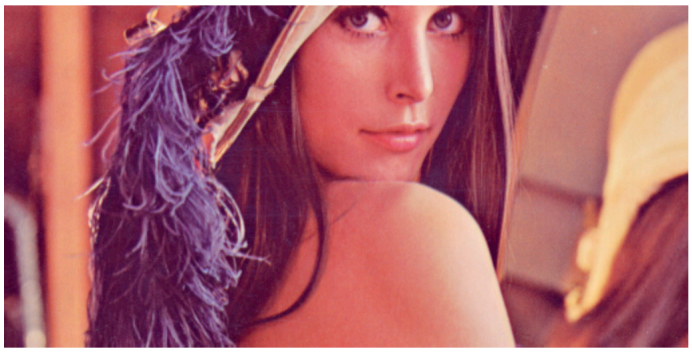

図 4.5: Lenna_1

パイプライン分割後の画像は図 4.4 と 4.5 に示す.
図 4.6 と 4.7 に示す．分割後画像はさらに二値化する.

```
1   img = Evision.imread("Lenna_0.png")
```

Image Raw Numerical

図 4.6: 二値化した Lenna_0

図 4.7: 二値化した Lenna_1

上記の実験により, 実際に Broadway のデータ取り込みとデータ処理機能を画像処理パイプラインとして活用することとが確認できた.

4.3.2 Broadway パイプラインにおける対処害機能 1 の評価

Broadway パイプラインは連続的で中断できないため，GenServer を使用して supervisor のフォールト トレランス メカニズムを実験する.

:pop と :push を使用して，データの書き込みと読み取りをシミュレートする．:pop と :push の定義は以下の通り.

def handle_call(:pop, _from, []) do
{:reply, :empty, []}
end
def handle_cast({:push, element}, state) do
{:noreply, [element | state]}
end

プロセスの異常シャットダウンのロジックをシミュレートするために abort_soft() を定義する．abort_soft() の定義は以下の通り.

def handle_cast(:abort_soft, _state) do
exit(:shutdown, :abort$_{s}oft$)
end

:abort_soft メッセージが AbortNif プロセスに送信した後，メッセージは handle_cast/2 関数によってキャプチャされ，プロセス終了処理ロジックがトリガーされる．:abort_soft メッセージを受信すると，exit/1 関数が呼び出され，ソフト終了信号 {:shutdown, :abort_soft} が現在のプロセスに送信される．これにより，GenServer は終了信号を受信した後にプロセスを終了する.

次に，Supervisor を使用して，異常終了したプロセスを再起動できるかどうかをテストする．Supervisor の構成は以下のとおり.

defmodule AbortNif.Application do

```
# See https://hexdocs.pm/elixir/Application.html
# for more information on OTP Applications
@moduledoc false
use Application
@impl true
def start(_type, _args) do
children = [
AbortNif
# Starts a worker by calling:
AbortNif.Worker.start_link(arg)
# AbortNif.Worker, arg
]
# See https://hexdocs.pm/elixir/Supervisor.html
# for other strategies and supported options
opts = [
strategy: :one_for_one, name:
AbortNif.Supervisor
]
Supervisor.start_link(children, opts)
end
end
```

実験結果を図 4.8 に示す.

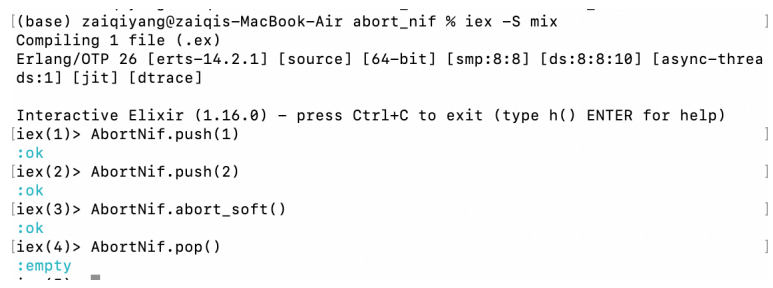

図 4.8: プロセスの異常シャットダウン

図 4.8 から，プロセスが終了した後 Supervisor の監視の下でプロセスが再開され，同時に書き込まれたデータがクリアされることがわかる.

それから, Benchee を使用して, 実際にメモリ不足などによる異常終了が発生した場合, 回復速度をテストを行う.

結果を表 4.1 に示す.

Name	IPS	Average	Deviation	median	99th%
Supervisor	3.51 M	284.79 ns	± 6400.68%	250ns	375 ns

表 4.1: supervisor の回復速度

4.3.3 Broadway パイプラインにおける対処害機能 2 の評価

まず，Broadway で書かれた中止関数を呼び出す．マルチノード フォールト トレランス メカニズムを使用しない場合の結果を図 4.9 に示す．Erlang VM 全体がシャットダウンした．

```
root@84df33344504:/home/broadway_nif# iex -S mix
Compiling 2 files (.ex)
Generated broadway_nif app
Erlang/OTP 26 [erts-14.0.2] [source] [64-bit] [smp:7:7] [ds:7:7:10] [async-threads:1] [jit]

Interactive Elixir (1.15.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> Aborted
root@84df33344504:/home/broadway_nif#
```

図 4.9: マルチノード フォールト トレランス 未使用

次に，Broadway パイプラインでマルチノード フォールト トレランス メカニズムを実行し，第 3 章で記述した abort_NIf 関数を子ノードに送信する．

それから，Erlang VM が異常終了するかどうかを監視する．実験結果を図 4.10 と 4.11 に示す．

```
root@84df33344504:/home/broadway_nif# iex -S mix
warning: unused import Calc
  lib/a.ex:3

Erlang/OTP 26 [erts-14.0.2] [source] [64-bit] [smp:7:7] [ds:7:7:10] [async-threads:1]
 [jit]


15:17:33.431 [info] Port epmd is active.
Interactive Elixir (1.15.4) - press Ctrl+C to exit (type h() ENTER for help)

15:17:33.434 [info] wait launching epmd...

15:17:33.434 [info] Port epmd is active.

15:17:33.434 [info] done.

15:17:33.485 [debug] os.type: {:unix, :linux}

15:17:33.489 [debug] short hostname: 172.17.0.2

15:17:33.489 [debug] fully qualified hostname: 172.17.0.2

15:17:33.489 [info] starting node host_mia2dxcu@172.17.0.2

15:17:33.497 [info] Node host_mia2dxcu@172.17.0.2 has been launched.

15:17:33.497 [debug] os.type: {:unix, :linux}

15:17:33.499 [debug] short hostname: 172.17.0.2

15:17:33.499 [debug] fully qualified hostname: 172.17.0.2

15:17:33.499 [info] Register worker :"co_elixir_3m3xjomz@172.17.0.2"
```

図 4.10: マルチノード フォールト トレランス 使用後 1

図 4.10 から、Erlang VM がクラッシュすることなく host_mia2dxcu@172.17.0.2 ノードと co_elixir_3m3xjomz@172.17.0.2 を生成した．

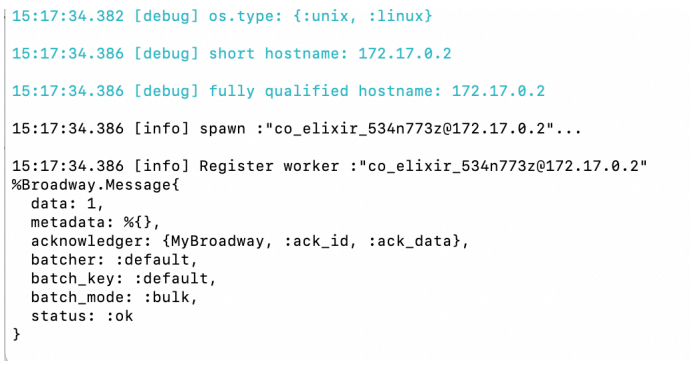

図 4.11: マルチノード フォールト トレランス 使用後 2

図 4.11 からわかるように，Erlang VM はクラッシュされておらず，Broadway パイプライン内のデータは完全にコマンド ラインに出力されている．

第5章　まとめと将来課題

5.1　まとめ

本研究では，Broadway と，Supervisor 監督下の Linux プロセスを生成してネイティブコードを実行し，分散コンピューティング機構である Node を用いて通信を行うことでネイティブコード実行の耐障害性を実現する OSS である SpawnCoElixir を組みわせることを提案した. 実験方法は以下通りである.

実験 (1) は Broadway のデータ取り込みとデータ処理機能を画像処理パイプラインとして活用できることを確認する為に, 実装した Broadway パイプラインで HTTPoison.get!() 関数を使用して画像を取得して, 画像がパイプラインにロードされた後, 画像に Elixir の関数ライブラリ Evision によって分割, 二値化が行われる.

実験 (2) は画像処理プロセス途中でメモリ不足などによる異常終了することに対応できるかどうかを確認する為に, スタックを取り出す pop(), スタックに要素を追加する push() とプロセスが終了される abort_soft() を定義し, Supervisor の監視の下で push()，abort_soft()，pop() を順番に実行して，スタック値がクリアされているかどうかを観察する. 同時に Benchee で abort_soft() が測って, 回復速度をテストを行う.

実験 (3) は提案した Broadway パイプラインで画像処理の時 NIF 関数が異常終了すると，Erlang VM 全体が異常終了するのは回避できるかどうかを確認する為に, SpawnCoElixir を実装なしの Broadway パイプラインと実装された Broadway パイプライン中に最初の段階で第 3 章で記述した abort をシミュレートする abort_NIF 関数を呼び出して，Broadway パイプラインの状態を観察する.

結果は以下通りである.

実験 (1) の結果, はフォルダー内に 2 つの分割された画像 Lenna_0 と Lenna_1 が入った. つまり, Broadway パイプラインを利用することで, 画像の読み込み, 画像分割, 画像二値化, 画像保存といったバッチ処理ができた.

実験 (2) の結果は, push() スタックに書き込まれている値が空になった. Benchee で得られた abort_soft() の平均回復時間は 284.79ns である.Supervisor がプロセスを再起動すると, プロセスの状態が再初期化されるから, push() スタックに書き込まれている値が空になったにより，Supervisor の監視の下, 異常終了をシミュレートするプロセスが再起動できることを確認した.

実験 (3) の結果は, SpawnCoElixir を実装されない Broadway パイプラインと Erlang VM が同時にクラッシュした. SpawnCoElixir を実装された Broadway パイプラインはクラッシュしなっかた. 上記の実験結果により SpawnCoElixir を実装した Broadway パイプライは Erlang VM 全体が異常終了を回避した. つまり, 提案した Broadway パイプラインは画像処理の時, NIF 関数が異常終了すると，Erlang VM 全体が異常終了を回避できることを確認した.

提案手法が設計意図通りネイティブコード障害時の耐障害性を備えていること を実証するために，ネイティブコードに Fault-Injection を埋め込むことで異常終了をシミュレート

した. また障害から回復するまでに必要な処理時間も計測した. Fault-Injection を用いた異常終了シミュレーションの評価結果では，設計通りに障害復旧できることを確認した. また，障害復旧に必要な処理時間の平均は 284.79ns と極めて短かった.

以上の実験結果を総合すると, CoElixir と Broadway フレームワークを導入したパイプラインは画像処理で優れた耐障害性と信頼性を発揮し, 画像バッチ処理の信頼性が向上した. 改良されたプロセスにより，大規模な画像処理タスクへの適応性が高まり，実際の応用シナリオに有効な解決策を提供した.

5.2 将来課題

将来課題としては，ネイティブコードで途中まで データ処理を実行できた場合に，ネイティブコード中での障害発生時にデータ消失する問題を解決することを求められる.

謝辞

この論文の執筆にあたり, 多くの方々にご支援いただきました. まず, 指導教員である山崎進先生に深く感謝申し上げます. 先生の専門的な知識と的確なアドバイスは, 私の研究を大きく前進させる助けとなりました. また, 多忙な中でも丁寧なご指導を賜り, 心より感謝申し上げます.

私の家族や友人にも深い感謝を捧げます. 彼らの理解と励ましは, 私が研究に専念できる大きな支えとなりました.

最後に, これまで私を支えてくださったすべての方々に心からの感謝を申し上げます. 皆様のおかげで, この論文を完成させることができました.

本研究の一部は，北九州産業学術推進機構 (FAIS) 旭興産グループ研究支援プログラムの支援を受けた.

参考文献

[1] Tom B. Brown, Benjamin Mann et al. 2020. Language models are few-shot learners. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20). Curran Associates Inc. Red Hook, NY, USA, Article 159, 1877–1901.

[2] Jean Bacon 1996. Concurrent Systems: An Integrated Approach to Operating Systems, Distributed Systems and Database. Addison-Wesley. pp.265–276.

[3] 山崎進，森正和，上野嘉大，高瀬英希: Hastega: Elixir プログラミングにおける超並列化を実現するための GPGPU 活用手法，情報処理学会第 120 回プログラミング研究会， Vol. 2018，No. 2，pp. 8，2018.

[4] Broadway: Build concurrent and multi-stage data ingestion and data processing pipelines with Elixir `https://github.com/dashbitco/broadway` (2023.10.2).

[5] Susumu Yamazaki: Robust, Distributed, and Parallel Processing for Enormous Images `https://www.youtube.com/watch?v=RkMzCQm-Ws4t=1087s` (2023.11.12).

[6] Hsueh, M.-C., Tsai, T.K. and Iyer, R.K.: Fault Injection Techniques and Tools, IEEE Computer, Vol.30, No.4, pp.75–82 (1997).

[7] The Elixir Team: Elixir is a dynamic, functional language designed for building scalable and maintainable applications `https://elixir-lang.org`. (online), `https://github.com/elixir-lang/elixir` (2023.07.20).

[8] The Elixir Team: Functions related to VM nodes. Retrieved June 20, 2023 from `https://hexdocs.pm/elixir/1.16.0-rc.0/Node.html`

[9] Alexandre Jorge Barbosa Rodrigues and Viktória Fördős. 2018. To- wards Secure Erlang Systems. In Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang (Erlang 2018). ACM, New York, NY, USA, 67–70. `https://doi.org/10.1145/3239332.3242768`

[10] Ericsson AB: Erlang Kernel Reference Manual Version 6.4, net kernel:monitor nodes. Retrieved June 20, 2023 from `http://erlang.org/doc/man/net_kernel.html`.

[11] The Elixir Team: GenServer: A behaviour module for implementing the server of a client-server. `https://hexdocs.pm/elixir/GenServer.html`

[12] SpawnCoElixir: spawns cooperative Elixir nodes that are supervised. `https://hexdocs.pm/spawn_co_elixir/SpawnCoElixir.html` (2024.1.13).

[13] Benchee: Library for easy and nice (micro) benchmarking in Elixir. `https://hexdocs.pm/benchee/Benchee.html` (2024.1.13).

付録