Vol.2019-ARC-240 No.48
Vol.2019-SLDM-191 No.48
Vol.2019-EMB-53 No.48
2019/2/28

関数型言語Elixirによる ROS 2のスケーラビリティを向上させるクライアントライブラリ

今西 洋偉[1,a)]　高瀬 英希[1,2,b)]

概要：ROS 2（Robot Operating System 2）はアプリケーションの処理単位であるノードを複数組み合わせることでロボットシステムを構築する開発支援フレームワークである．2012 年に登場した関数型言語 Elixir は，軽量なプロセスモデルと耐障害性を兼ね備えている．我々は ROS 2 による分散ロボットシステムと Elixir のプロセスモデルとの親和性が高いと考えている．本研究では Elixir による ROS 2 のクライアントライブラリを提案する．ROS 2 における出版購読通信機能を提供する Elixir の API を設計し，これによって分散ロボットシステムのスケーラビリティの向上を図る．さらに Elixir のスーパーバイザを導入することで耐障害性を持たせた．提案手法を実装してその性能を評価し，ROS 2 アプリケーションにおける Elixir の適用可能性および技術課題を明らかにする．

1. はじめに

近年，社会生活を支援するためのロボットの活用分野が広がっており，ロボットアプリケーションの開発を支援する様々なソフトウェアフレームワークが注目されている．中でも特に注目を集めているものとして ROS（Robot Operating System）[1] が挙げられる．ROS では，アプリケーションの機能単位をノードとして表現し，複数のノードを目的に応じて組み合わせることで所望のロボットシステムを構築することができる．ROS はノード間における通信層を提供するミドルウェアでもある．主なノード間通信として，トピック名でデータの種類を識別し，出版されるメッセージを購読する出版購読通信方式がある．

近年，ROS を産業用途に用いる中で生まれた課題を解決するための次世代フレームワークとして ROS 2[2] の開発が進められている．

ROS からの大きな変更として，DDS（Data Distribution Service）と呼ばれる通信プロトコルを採用した点が挙げられる．また，ROS 2 のコアとなる共通機能は C 言語で実装されており，これを基にして様々なプログラミング言語でクライアントライブラリを実現することが可能となっている．アプリケーション全体で 1 つの言語ではなく，ノードごとにユーザーの使いたい言語で実装することも可能である．

[1] 京都大学
[2] JST さきがけ
[a)] emb@lab3.kuis.kyoto-u.ac.jp
[b)] takase@i.kyoto-u.ac.jp

プログラミング言語に関しては，関数型言語である Elixir が注目されている．Elixir は 2012 年に登場し，処理の振る舞いではなくデータの扱いを直接的に操作するためのライブラリや記法が豊富に整備されている言語である．また Elixir には，少ない記述量でのコーディングが可能になっており読み書きしやすく生産性が高いという利点がある．Elixir はプロセスモデルに基づく軽量プロセスを備えており，並列分散システムの構築に適している．また，プロセスリンクやモニタ機能によってプロセス障害時の高速な復旧が実現でき，耐障害性にも優れている．

このような Elixir の並列プロセスモデルは，多数のノードが様々なトピックを介して出版および購読を行う ROS 2 のシステムモデルと親和性が高いと考えられる．これらのノードの振る舞いとデータのやり取りを Elixir のプロセスモデルで実現できれば，より軽量かつスケーラブルな ROS 2 分散ロボットシステムの実現が期待できる．

本研究では，よりスケーラブルな ROS 2 アプリケーションを開発できる枠組みを作ることを目指す．目的達成の鍵として，関数型言語 Elixir で ROS 2 クライアントライブラリを実装し，Elixir の持つ特徴を ROS 2 アプリケーションに活用できるようにする．Elixir の ROS 2 クライアントライブラリである RclEx を設計し，これを実装する．ノード数に応じた CPU 使用率やメモリ使用率，通信時間を性能評価し，ROS 2 アプリケーション開発における Elixir の適用可能性を示す．

本論文の構成は次の通りである．2 章では準備として ROS 2 と Elixir について解説する．3 章では，我々が提

Vol.2019-ARC-240 No.48
Vol.2019-SLDM-191 No.48
Vol.2019-EMB-53 No.48
2019/2/28

案する Elixir クライアントライブラリ RclEx の設計と，RclEx を用いたアプリケーションのプログラミングスタイルを示す．4 章において本研究における実装成果の評価を示し，最後に 5 章で本論文のまとめと今後の課題を述べる．

2. 準備

2.1 ROS 2

2.1.1 特徴と利点

ROS 2 の前身である ROS[1] は，WillowGarage とスタンフォード大学が共同で開発したロボットソフトウェアの開発フレームワークである．プログラム部品をノードと呼び，目的に合わせてノードを組み合わせることで，ロボットシステム全体を構築するコンポーネント指向開発が可能になっている。また ROS は，複数の異なるハードウェア間のデータ送受信やスケジューリング，エラー処理などのロボットアプリケーションプログラムの開発と実行に必要な機能をライブラリとして提供している。

ROS 2 は ROS の持つ機能に新たなニーズに応えた機能が追加された次世代版 ROS フレームワークである．もともと ROS は主に研究用途のプラットフォームとして注目されていたが，近年，ROS-Industrial という団体ができるなど産業用途に ROS システムを採用したいという傾向が強くなってきている [3]．そのため ROS に対して以下のような要望が寄せられていた．

(1) 様々な複数台のロボットの同時動作．
(2) リソース制約のある組込みシステム環境のサポート
(3) ネットワークが貧弱な環境への対応

ROS 2 はこれらの要求に対応するため，以下のような設計が施されている．

(1) 通信ミドルウェアに DDS を採用する．これによって単一障害点であった ROS マスタを取り除く．
(2) 核の部分は C 言語で実装して他の機能もプラグインとして提供する．
(3) 通信品質を考慮するための QoS を導入する．

2.1.2 アーキテクチャ

ROS 2 の内部アーキテクチャを図 1 に示す．

DDS とは，OMG（Object Management Group）によって策定された出版購読通信を提供する国際標準規格である．DDS にはノード間の自動検出機構が備わっておりネームサーバが無くても複数の DDS プログラム間で通信することができる．加えて DDS ではノード間の通信品質を調整できる Quality of Service（QoS）パラメータが設定でき，通信データの欠損を防ぐことで通信の信頼性を高めたり，通信速度に応じてメッセージキューのサイズを調整することができる [4]．現在複数のベンダから様々な DDS プロトコルが ROS 2 用に公開されているが，そのうち FastRTPS[5] が現在 ROS 2 の標準 DDS になっている。

RMW は，DDS の通信機能と ROS 2 の基盤機能を吸収する層であり，それぞれの DDS に応じた実装が提供されている．RMW は実行時に変更できるため，使用するデバイスの計算資源や，させたい処理に応じて適した DDS 実装を必要な機能だけ任意で設定することもできる．

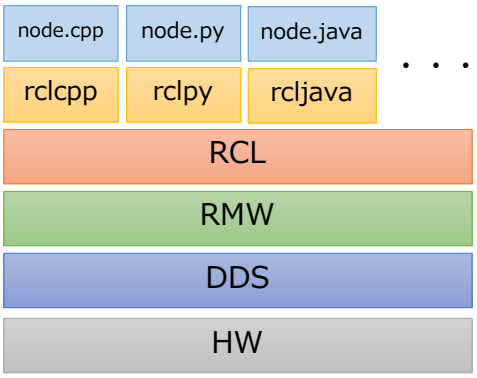

図 1 ROS 2 の内部アーキテクチャ
Fig. 1 Internal architecture of ROS 2

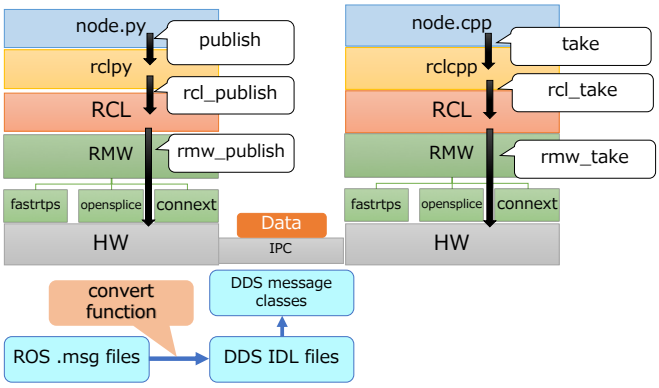

図 2 ROS 2 におけるノード間通信
Fig. 2 Inter-node communication in ROS 2

RCL は，各プログラミング言語に共通する機能を提供する，C 言語で実装された層である．共通する機能として名前空間やロギング，同期および非同期サービスなどが実装されている．RCL の C 言語関数をラップすることで各言語固有のクライアントライブラリを開発できる．公式に提供されている ROS 2 クライアントライブラリとして，C++による rclcpp，および Python による rclpy がある．またノードごとにプログラミング言語が異なっていたとしても上記のように共通機能をラップしているため図 2 のようにメッセージのやり取りをすることができる．

2.2 関数型言語 Elixir

Elixir は Erlang VM 上で動作する関数型言語である．Elixir の特徴として，開発生産性の高さ，並行性，耐障害性が挙げられる [6]．加えてこれまでの関数型言語に比べて言語仕様や概念がシンプルであるが応用が効くように設計されている．

並行性について，Elixir のコードはアクターまたはプロ

セスと呼ばれる分散された軽量の実行スレッドの中で動作し，あるプロセスが新しく別のプロセスを生成，終了することも可能である．これをプロセスモデルまたはアクターモデルと呼ぶ．1 プロセスは起動時間が数マイクロ秒，メモリはおよそ 300 ワードという軽さで [7]，同一のマシン内で数千のプロセスを同時に起動することも容易である．

Elixir の耐障害性について，各プロセスがそれぞれでガベージコレクションを含む堅牢なメモリ管理をしている．さらにレスポンス性が高く，例外が発生しプロセスに障害が起こった場合は try/catch のよう例外処理を行うのではなく，そのプロセスごと監視プロセスから高速に再起動し，監視プロセスで例外処理を行う．そのため複雑な例外処理を記述する必要がなく，障害復旧が簡潔に行われる．

これらの特徴を生かした Web フレームワーク Phoenix も注目されている．Phoenix[12] は，Ruby における Ruby on Rails と同等以上の生産性を誇りながら，レスポンス性が極めて高い．[13] によれば，Phoenix によって構築されたサーバは Ruby on Rails のそれよりも 10.63 倍のスループットを達成できたという評価結果が報告されている．

3. ROS 2 クライアントライブラリ RclEx

本研究では，Elixir による ROS 2 クライアントライブラリである RclEx を提案する．Elixir の利点を ROS 2 に取り入れることで，よりスケーラブルで堅牢なロボットアプリケーションが開発できることが期待される．我々は RclEx を，よりスケーラブルな ROS 2 アプリケーションを開発できる枠組みにすることを目標としている．本章では我々が実装を進めている RclEx の設計と実現方法について説明したのち，RclEx のプログラミングスタイルを例を用いて示す．

3.1 本実装の目標と提供する API

本研究の実装では Elixir のコード内で，1 つのトピックに対する任意個数のノードの生成，ノードからのパブリッシャもしくはサブスクライバの生成，それらの間での出版購読通信を実現できるようにする．出版購読にはプロセスモデルを採用することで，通信の軽量化およびスケーラビリティの向上を図る．目標の実現のために実装する必要のあるユーザー API の一覧を表 1 に示す．

3.2 実装方法

3.1 で示した実装を実現するための要素技術を簡潔にまとめる．まず ROS 2 の各言語に共通機能がまとめられている RCL から今回の目標とする実装に必要な構造体や関数を選定する．次に Elixir コードからそれら C 言語関数を呼び出すために，Elixir の動作する Erlang VM で提供されている NIF（Native Implemented Function）という機能を用いて RCL の関数をラップする．これによって ROS 2 における基本的な出版購読通信の実現を目指す．

さらに本研究では，Elixir のプロセスモデルの特徴を活かした ROS 2 ノードの生成手法を提案する．具体的には，プロセスモデルのひとつである Task を採用し，出版購読通信する大量のノードを同時に生成する機能を提供できるようにする．

以下で各要素技術について細かく説明していく．

3.2.1 Elixir から呼び出す関数の選定

現在 RCL には膨大な数の API が実装されている．その中から出版および購読に必要となる基本的な構造体および関数を選定した．その中から代表的なものを表 2，3 に示す．

3.2.2 NIF による RCL のラップ

NIF を用いることで C/C++で実装された外部コードが Erlang VM と同じメモリアドレスにロードされ [11]，Elixir のデータ構造が C 言語で対応するものに変換および統合される．

C 言語で構造体として表現されるデータ構造は Erlang においてメモリブロックであるリソースオブジェクトで扱われる [10]．構造体を Elixir から扱うサンプルコードを図 3，4 に示す． このように NIF を経由することで Elixir から C 言語の関数や構造体を操作することができる．これを用いて表 2，3 で示した関数や構造体を Elixir から呼び出すことができる．

実装フローは以下の通りであり，図 6 に示す．

(1) RCL に定義されている関数や構造体をラップした NIF を実装する．
(2) (1) で実装した NIF をコンパイル，ROS 2 共有ライブラリをリンクして共有ライブラリ rclex.so を作成する．
(3) RclEx モジュールで rclex.so をロード，ラップした API を呼び出すための関数を定義することで，RCLAPI を利用可能にする．

3.2.3 パブリッシュとサブスクライブ

ROS 2 の出版購読には Elixir のプロセスモデルを対応させる．プロセスモデルで主に用いられるモジュールには Task，Agent および GenServer がある．Task は状態を持たずバックグラウンドで関数を走らせるためのプロセスであり，コマンド操作などアプリケーションの動作を妨げることなく，実行コストの高い処理を行うことが可能である．Agent は状態を維持するためのプロセスを生成し，GenServer は Agent と Task 両方の特徴を備えている．本研究では，パブリッシャやサブスクライバの状態は RCL で定義されている各種構造体にて維持されるため，状態を持たず軽量である Task を採用することにした．

加えて Task に備えられているプロセス監視モジュール Task.Supervisor モジュールも利用する．これにより，パブリッシュまたはサブスクライブ関数を走らせるタスク（それぞれパブリッシュタスク，サブスクライブタスクとする）

Vol.2019-ARC-240 No.48
Vol.2019-SLDM-191 No.48
Vol.2019-EMB-53 No.48
2019/2/28

表 1　実装する RclEx の API
Table 1　Implementation target of APIs in RclEx

API 名	引数	返り値	説明
rclexinit/0	-	コンテキスト	ROS 2 初期設定
create_nodes/3	コンテキスト，ノード名，ノード数	ノードのリスト	ノード生成
create_publishers/2	生成されたノードのリスト，トピック名	パブリッシャのリスト	指定したトピックのパブリッシャを生成
create_subscribers/2	生成されたノードのリスト，トピック名	サブスクライバのリスト	指定したトピックのサブスクライバを生成
publish/2	パブリッシャのリスト, 出版するメッセージのリスト	パブリッシャプロセスの PID リスト	データ出版
subscribe/2	サブスクライバのリスト, コールバック関数	サブスクライバプロセスの PID リスト	データ購読
initialize_msg/2	生成するメッセージの数，メッセージの型	メッセージリスト	通信に用いるメッセージを複数初期化
setdata/2	メッセージ，データ	-	メッセージオブジェクトにデータを入れる
read_msg/1	メッセージ	データ	メッセージオブジェクトに入れられたデータを取り出す
create_timer/3	パブリッシャもしくはサブスクライバのリスト，タイマー周期（ミリ秒），コールバック関数	-	指定した関数を周期的に実行する

表 2　実装対象とする RCL の構造体
Table 2　Structures implemented in RCL which we are targetting to call from elixir

構造体名	説明
rcl_context_t	ROS 2 の初期化からシャットダウンまでのローカル状態をカプセル化したもの．ノード生成に必要となる．
rcl_node_t	ノード情報．rcl_context_t に加えてノード名などが加えられている．
rcl_publisher_t	パブリッシャ情報
rcl_subscription_t	サブスクライバ情報

表 3　実装対象とする RCL の関数
Table 3　Functions implemented in RCL which we are targetting to call from elixir

関数名	説明
rcl_init	RCL の初期化．rcl_context_t を有効な状態にする
rcl_node_init	ノード初期化．rcl_node_t を有効にする．これ以降パブリッシャやサブスクライバ生成に利用できる．
rcl_publisher_init	パブリッシャ初期化．rcl_node_t，トピック名およびメッセージの型から，ゼロ初期化した rcl_publisher_t を有効化．
rcl_subscription_init	サブスクライバ初期化．rcl_node_t，トピック名およびメッセージの型から，ゼロ初期化した rcl_subscription_t を有効化．
rcl_publish	メッセージ出版．
rcl_take	メッセージ購読

が仮に何らかの原因でダウンしたとしてもスーパーバイザプロセスから高速に再起動させることができる．

Task によるパブリッシャおよびサブスクライバ生成から出版購読までの流れを図 5 に示す．

メインプロセスから Task.Supervisor モジュールの start_link 関数により，タスクを監視するスーパーバイザを生成する．生成されたスーパーバイザプロセスが Task.Supervisor モジュールの start_child 関数により，パブリッシュタスクおよびサブスクライブタスクを監視対象プロセスとして起動する．タスク内でそれぞれが出版関数，購読関数を呼び出す．サブスクライブタスクは購読関数を再帰的に走らせることにより，トピックに出版されたメッセージを非同期に購読することができる．

3.3　RclEx を用いた ROS 2 アプリケーション

RclEx は，Elixir のプロジェクト管理ツールである mix で利用できる．

図 7，図 8 は，表 1 で示した RclEx のユーザ API を利用して，同一トピックに関するパブリッシャ，サブスクライバノードを 10 個生成して実行する例を示している．ノードは，create_nodes 関数に渡したノード名にインデックスをつけるようにしてノード名に重複が無いように複数生成される．図 7 では 1000ms 周期で callback 関数が呼び出され，関数内でデータを取得して出版している．図 8 では購読時

Vol.2019-ARC-240 No.48
Vol.2019-SLDM-191 No.48
Vol.2019-EMB-53 No.48
2019/2/28

```
#include <erl_nif.h>

typedef struct {
  int data;
} Example;
/* Elixir から呼び出したい関数 */
void set(Example* arg){
  arg->data = 1;
}
/* リソースオブジェクトのクラスである
   リソースタイプを宣言 */
ErlNifResourceType* resource_type;
ERL_NIF_TERM set_nif(ErlNifEnv* env,
int argc,const ERL_NIF_TERM argv[]){
  Example* res;
  ERL_NIF_TERM ret;
  /* 構造体のサイズ分メモリを割り当てる */
  res = enif_alloc_resource(resource_type,
    sizeof(Example));
  /* erlang で扱うためのリソースオブジェクトへの
    ハンドラを取得する */
  ret = enif_make_resource(env,res);
  /* 関数を呼び出す */
  set(res);
  /* ハンドラを返す */
  return ret;
}
int load(ErlNifEnv* env){
  /* 構造体のリソースタイプを作成 */
  resource_type =
    enif_open_resource_type(env,"Elixir.Sample",
    "Example",NULL,ERL_NIF_RT_CREATE,NULL);
  return 0;
}
/* NIF と Elixir で関数をマッピング */
ErlNifFunc nif_funcs[]={
  {"set",0,set_nif},
};
/* NIF ライブラリを初期化 */
ERL_NIF_INIT(Elixir.Sample,nif_funcs,
  &load,NULL,NULL,NULL);
```

図 3　Example 構造体を Elixir から操作するための NIF
Fig. 3　NIF to operate Example structure from Elixir code

のコールバック関数をユーザーが定義する．RclEx 内の購読関数にて RCLAPI である rcl_take() の返り値をパターンマッチングさせており，正しくメッセージを受け取った際にコールバック関数が呼ばれるようになっている．

4. 評価

本章では，提案する RclEx について様々な計測を行い性能を評価する．比較対象として rclcpp による実装を用いた．

```
defmodule Sample do
  @on_load :load_nifs
  def load_nifs do
    :erlang.load_nif('./sharedlib',  0)
    #NIF から生成された共有ライブラリをロードする.
  end
  def set do
    #NIF が実装されていない時にエラーを発生させる.
    raise "NIF set/1 is not implemented"
  end
end
```

図 4　NIF を経由して Example を操作する Elixir コード
Fig. 4　Elixir code to operate Example structure via NIF

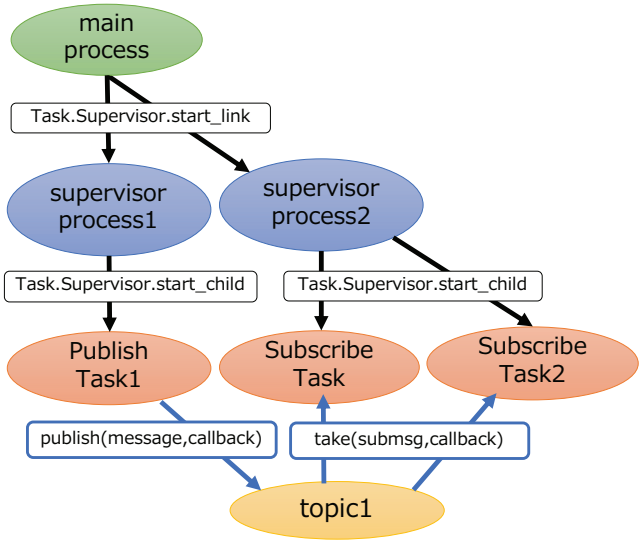

図 5　Task による出版購読
Fig. 5　publish and subscribre by Elixir Task

4.1 評価環境

今回性能評価に使用した PC のスペックとツールのバージョンは表 4 の通りである．今回はそれぞれ別の OS プロセスでパブリッシャとサブスクライバを実行し出版購読を行った．出版購読は rclcpp 同士，RclEx 同士で行い，データは String 型を用いた．以下で計測する通信時間はデータが出版された直後の時間と購読された直後の時間を 100 回計測しその平均の差から算出している．計測には rclcpp では gettimeofday 関数を，RclEx では Erlang/OTP の:os.system_time 関数をそれぞれ用いた．また CPU 使用率およびメモリ使用率は，パブリッシャとサブスクライバ以外のアプリケーションを閉じた状態で，sar コマンドを用いて PC 全体の負荷状況を 1 秒周期で合計 100 回取得し，取得した値から算出した平均値を用いた．

4.2 通信時間

まずパブリッシャとサブスクライバを 1 つずつ用意し，0.1 秒周期でデータを出版，データサイズを 256Byte から

Vol.2019-ARC-240 No.48
Vol.2019-SLDM-191 No.48
Vol.2019-EMB-53 No.48
2019/2/28

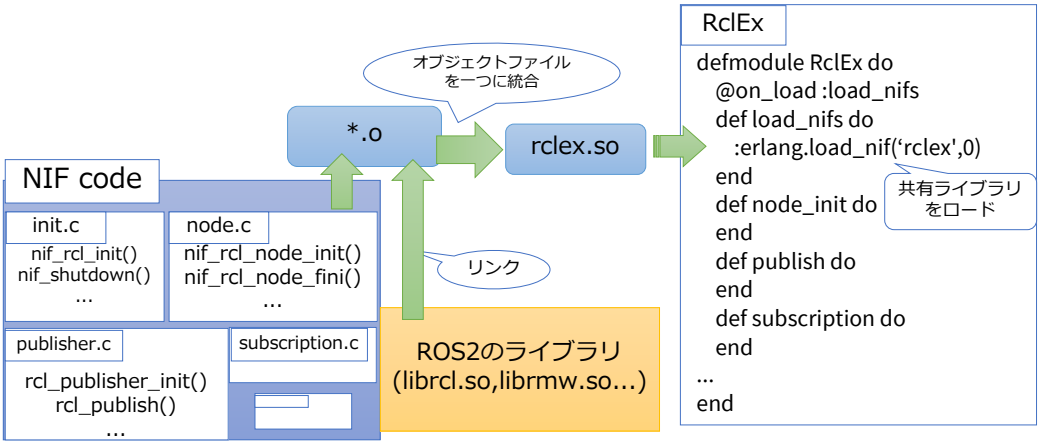

図 6　RclEx 実装の流れ
Fig. 6　RclEx implementation flow

```
defmodule PubSample do
  def pubmain do
    # Publsher となるノード数を指定
    num_node = 10
    RclEx.rclexinit
    |> RclEx.create_nodes('test_pub_node',num_node)
    |> RclEx.create_publishers('testtopic')
    |> RclEx.Timer.create_timer(1000,&callback/1)
  end
  @doc """
    タイマーによる周期的な処理を記述
  """
  def callback(publisher_list) do
    #publisher の数に応じてメッセージのリストを作成する
    msg_list =
    length(publisher_list)
    |> RclEx.initialize_msgs(:string)
    {:ok,data} = File.read("test.txt")
    #メッセージリストにデータをセット
    Enum.map(0..length(msg_list)-1,fn(index)->
      RclEx.setdata(Enum.at(msg_list,index),
      data)
    end)
    #パブリッシャタスクに出版させる
    RclEx.Publisher.publish(publisher_list,msg_list)
  end
end
```

図 7　パブリッシャを 10 個実行するサンプルコード
Fig. 7　Sample code to run 10 publisher

256kByte まで 2 のべき乗ずつ変え，通信時間を計測した．その結果を図 9 に示す．結果からデータサイズが 64kbyte までは平均して約 250 マイクロ秒，RclEx による出版購読通信の遅延が小さかったが 128kB 付近で逆転した．

次にパブリッシャを 1 つ，サブスクライバを任意の数作成し，通信するデータサイズを 256Byte に固定，1 秒周期

```
defmodule MinimalSubscriber do
  def submain do
    #ノード数を指定
    num_node = 10
    RclEx.rclexinit
    |> RclEx.create_nodes('test_sub_node', num_node)
    |> RclEx.create_subscribers('testtopic')
    |> RclEx.Subscriber.subscribe(&callback/1)
  end
  @doc """
    購読時のコールバック関数を記述
  """
  defp callback(msg) do
    {:ok, received_msg} = RclEx.read_data(msg)
    IO.puts "received msg:#{received_msg}"
  end
end
```

図 8　サブスクライバを 10 個実行するサンプルコード
Fig. 8　Sample code to run 10 subscriber

でデータを出版し通信時間を計測した．その結果を図 10 に示す．rclcpp による実装に比べて RclEx による実装では，ノード数増加に伴う通信時間の増加は抑えられていた．

4.3　CPU およびメモリ使用率

先ほどのサブスクライバを任意の数作成する構成で CPU およびメモリ使用率を計測した．結果を図 11，12 にそれぞれ示す．RclEx の CPU 使用率は平均して 100%弱を記録した．メモリ使用率は PC を起動しただけの状態では 40.3%であった．RclEx はサブスクライバノードを増やしてもメモリ使用率の上昇は 2%未満であった．

4.4　考察

図 9 において，1 パブリッシャ 1 サブスクライバの場合

Vol.2019-ARC-240 No.48
Vol.2019-SLDM-191 No.48
Vol.2019-EMB-53 No.48
2019/2/28

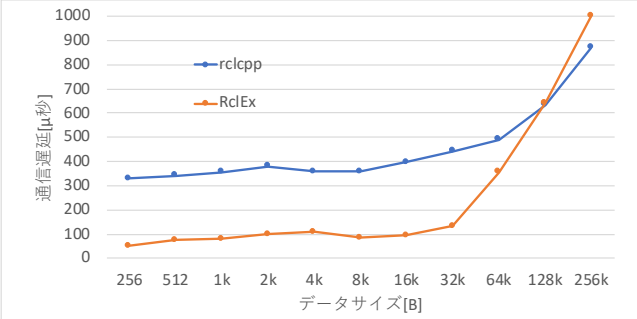

図 9 データサイズによる通信時間の変化
Fig. 9 Latency according to data size

における通信時間が逆転したことについて，これはデータ購読時における処理が要因であると考えられるが根本的な原因を究明することは現時点ではできていない．

図 10 から，サブスクライバを多数作成したケースでは，通信時間を大きく短縮できていることがわかった．これは Erlang VM のプロセススケジューラによる並列処理により実現されていると考えられ，出版購読の頻度が高いほど遅延軽減の効果が大きいと言える．

次に図 11 から，RclEx による実装の CPU 使用率が非常に高いことについて，現在はタスク内で 1 回の購読関数を再帰的に実行している仕様になっているためこのような結果になっていると考えられる．Elixir の特徴であるスケーリングにより，評価環境 PC に搭載された 2 コア 4 スレッドをノード数に関わらず有効活用しているとも言えるが，CPU 使用率を抑えるために今後の課題としてコールバックにイベントハンドラの仕組みを採用しなければならないと考える．

図 12 からは，RclEx のメモリフットプリントが非常に小さいことがわかった．Elixir の特徴である，各プロセスで個別に行われているガベージコレクションによって実現されていると考えられる．

以上の測定結果から，多数のノードで構成され出版購読通信が頻繁に行われる ROS 2 アプリケーションにおいて，RclEx を適用することは十分可能であると考えられる．

表 4 評価環境とツールのバージョン
Table 4 Evaluation environment and version of tools

OS	Ubuntu18.04
コア数	2 コア
コア	Intel Core i7
論理プロセッサ	4
クロック	2.6GHz
メモリ	16GB
ROS 2 version	dashing
DDS	FastRTPS
Elixir	1.9.4
Erlang/OTP	22

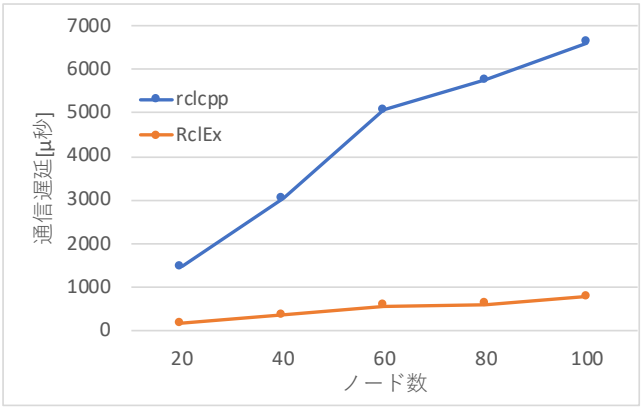

図 10 ノード数による通信時間の変化
Fig. 10 Latency according to number of nodes

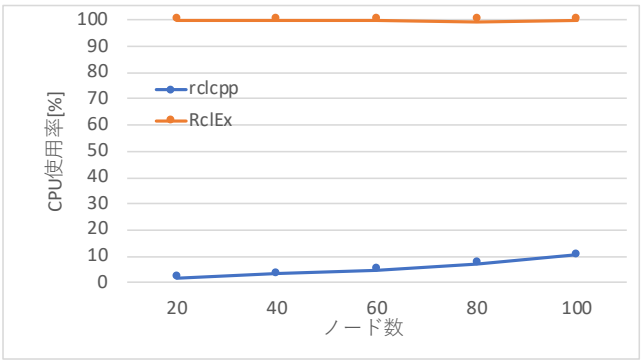

図 11 ノード数による CPU 使用率の変化
Fig. 11 CPU usage according to number of nodes

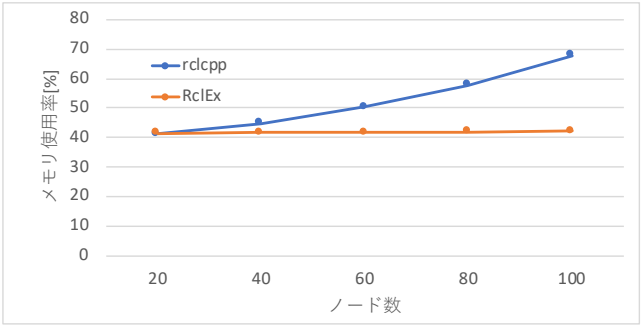

図 12 ノード数によるメモリ使用率の変化
Fig. 12 Memory usage according to number of nodes

5. まとめ

本研究では，Elixir の特徴であるプロセスモデルによる並列処理とスーパーバイザによる耐障害性を活用した ROS 2 クライアントライブラリ RclEx を提案した．Elixir の API を設計，実装し，あるトピックに対してパブリッシャ，サブスクライバを任意の個数作成，Elixir のプロセスモデルを採用したデータの出版購読を実現した．また Task.Supervisor モジュールを用いることでパブリッシュタスクやサブスクライブタスクが何らかの理由により障害が発生した場合におけるタスクの高速な復帰を実現した．

Vol.2019-ARC-240 No.48
Vol.2019-SLDM-191 No.48
Vol.2019-EMB-53 No.48
2019/2/28

実装した RclEx を用いて性能評価を行い，ノード数に応じた CPU 使用率とメモリ使用率，通信時間を示した．今回の評価結果から，データの出版購読が多数のノードで行われる場合に RclEx が適用可能であることが示された．RclEx を活用することで，ノード数増加による影響が小さいスケーラビリティの優れた堅牢な ROS 2 分散システムを容易に設計できることが期待できる．

今後の課題として，コールバックにイベントハンドラの仕組みを採用した実装を施すこと，サービスやアクションなど ROS 2 の出版購読以外の通信方式をサポートすること，また現在は String 型のみの対応になっているため，ユーザー定義のメッセージ型をサポートできるようライブラリを充実させることなどが挙げられる．

謝辞 本研究は，JST さきがけ JPMJPR18M8 の支援を受けたものである．

参考文献

[1] Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." ICRA workshop on open source software. Vol. 3. No. 3.2. 2009.
[2] https://index.ros.org/doc/ros2/
[3] ROS 2 ではじめよう次世代ロボットプログラミング 近藤豊
[4] 表 允晳，倉爪 亮，鄭 黎ウン『ROS ロボットプログラミングバイブル』オーム社，2018 年，pp.3-7
[5] e PROSIMA Fast RTPS https://www.eprosima.com/index.php/products-all/eprosima-fast-rtps
[6] Elixir official page https://elixir-lang.jp/
[7] Erlang Efficiency Guide http://erlang.org/doc/efficiency_guide/processes.html
[8] ROSCON 2016 Open Source Robotics Foundation ROS 2 Update https://roscon.ros.org/2016/presentations/
[9] Fedrecheski, Geovane, Laisa CP Costa, and Marcelo K. Zuffo. "Elixir programming language evaluation for iot." 2016 IEEE International Symposium on Consumer Electronics (ISCE). IEEE, 2016.
[10] Erlang document erl_nif https://erlang.org/doc/man/erl_nif.html
[11] Marx, Ben, Jose Valim, and Bruce Tate. Adopting Elixir: From Concept to Production. Pragmatic Bookshelf, 2018.
[12] Phoenix: Productive, Reliable. Fast. A productive web framework that does not compromise speed and maintainability (online), http://phoenixframework.org
[13] Chris Mccord: Elixir vs Ruby Showdown - Phoenix vs Rails (online), https://littlelines.com/blog/2014/07/08/elixir-vs-ruby-showdown-phoenixvs-rails.