Vol.2022-ARC-248 No.50
Vol.2022-SLDM-198 No.50
Vol.2022-EMB-59 No.50
2022/3/11

Elixir による ROS 2 クライアントライブラリの任意型メッセージの通信手法

佐藤 弘之[1]　武田 大輝[2]　菊池 俊介[3]　中村 宏[1]　高瀬 英希[1]

概要：近年のロボットアプリケーション需要の高まりの中で，ロボット開発を支援するプラットフォームとして ROS 2 が注目されている．ROS 2 は，ユーザ定義の構造体を含む任意型メッセージの出版購読通信に基づく汎用的な通信機能を提供する．いっぽう，我々が研究開発を進めている Elixir 向け ROS2 クライアントライブラリである Rclex は，対応するメッセージの型が文字列型のみであるという課題があった．そこで本研究では，Rclex での任意型メッセージ通信を実現することで Rclex の適用範囲の拡大することを目指す．Rclex 上で任意型メッセージを扱う API を設計し，ROS ノードとの出版購読通信を実現する手法を提案する．さらに，設計した API を任意のメッセージ型に対して自動生成し，一貫したプログラミングスタイルで扱うことができる手法を示す．提案手法を実装して性能評価を行い，単一のノード間での通信における有効性を確認した．

1. はじめに

近年，ロボットアプリケーションは多くの分野で活用されている．ロボットシステムには多数の機能の開発を要し，ソフトウェアからハードウェアに至るまで広範な知識を求められる．その中で，ロボット開発を支援するプラットフォームである ROS（Robot Operating System）[1] が注目されている．ROS は様々な機能を実現するツールやライブラリを備え，多種多様なロボットを低い開発コストで開発することが可能となっている．ROS ではノードと呼ばれる機能単位を組み合わせてロボットアプリケーションが構成される．組み合わされたノード同士は，トピックを介した出版購読通信などのメッセージ通信により相互接続される．出版購読通信とは，あるノードがトピックを指定してデータの出版を行い，そのトピックの購読を行う別のノードへデータが渡されるという通信方式である．ROS メッセージは構造体として定義され，トピックごとに扱うメッセージ型が指定されている．これにより，各トピックにおいて必要な情報を構造体にまとめて扱うことができ，汎用性の高いシステムモデルの構築が可能となっている．

ROS の次世代版である ROS 2 では，産業用途を想定したいくつかの変更に加え，そのアーキテクチャ構造にも変更が加えられた．言語間に共通した機能を提供する C 言語で実装された API の層を持つため，クライアントライブラリインターフェースの実現が容易になり，様々なプログラミング言語でアプリケーションを開発できる．

ROS 2 を用いたロボットアプリケーションのユースケースとして，大規模監視システムやクラウドロボティクスなどの，多数のデバイスが参加するシステムが想定される．このようなケースに対応するアプローチとして，関数型言語 Elixir による ROS 2 クライアントライブラリである Rclex が提案されている [2]．Elixir は並列性や耐障害性に優れた Erlang の仮想マシン上で動作する関数型言語である．Elixir では独立したプロセスを起動してコードを実行する．Rclex では Elixir のプロセスが ROS 2 ノードの動作を担う．さらに Elixir のプロセスは Erlang 仮想マシン上で動作する軽量なプロセスであり，Rclex は扱う ROS 2 ノード数が増加したときの処理性能への影響を抑え，高いスケーラビリティを実現している．いっぽう，現状の Rclex のノード間通信では文字列型のメッセージ通信のみが可能であり，その適用範囲は限られたものとなっていた．

本研究では，Rclex の適用範囲の拡大を目的として，Rclex における任意型のメッセージ通信を行う手法を提案する．これにより，任意型メッセージを扱う ROS 2 ノードとの通信システムを構築できるようにする．また，各トピックで扱うメッセージの見通しを良くすることでシステムモデルの簡素化およびアプリケーション開発コストの低減が期待できる．提案手法では，メッセージのインターフェース

[1] 東京大学
The University of Tokyo
[2] 京都大学
Kyoto University
[3] さくらインターネット
SAKURA internet

Vol.2022-ARC-248 No.50
Vol.2022-SLDM-198 No.50
Vol.2022-EMB-59 No.50
2022/3/11

として Elixir 構造体を定義し，それを C 構造体に対応させて ROS 2 の共通機能を呼び出すという形式をとる．

本論文の構成は次の通りである．まず第 2 章では，提案手法のもととなる既存技術について解説し，関連研究について述べる．第 3 章では，提案する Rclex における任意型メッセージの通信手法について，その設計および実装方法を示し，任意型メッセージ通信を行うプログラミング例を示す．第 4 章では，提案手法を実装し，メッセージ通信を実行して性能評価を行い，有効性を検証する．最後に第 5 章では，本論文の結論および今後の課題を述べる．

2. 準備

2.1 ROS 2

2.1.1 概要

ROS 2（Robot operating system 2）はロボット開発を支援するプラットフォームである．主に研究用途として注目されていた ROS の次世代版として，産業用途での利用を想定して開発された．ROS と比較した ROS 2 の特徴として次の事項が挙げられる [3].

- 複数台ロボットの同時利用に対応
- 組み込みプラットフォームにも対応
- UDP プロトコルをベースとしたリアルタイム性の確保された通信
- データの欠損や通信の遅延を許容
- プログラミング形式をある程度固定

ROS 2 のアプリケーションはノードと呼ばれる機能単位により構成され，ノード間でトピックを介した出版購読通信が行われる．ROS 2 の出版購読通信は，パブリッシャがトピックを指定してメッセージを送信し，そのトピックからの情報を受け取るよう指定されているサブスクライバにそのメッセージが渡されるという流れになっている．このような出版購読通信モデルは，各ノードが通信相手について関知せず非同期で行われるためノード同士の結合が弱く，分散疎結合システムの構築に有効である．

ROS 2 のアーキテクチャは機能ごとにいくつかのレイヤーに分かれ，それらが重なった階層構造となっている．図 1 に ROS 2 アーキテクチャの階層構造を示す．まず通信基盤の構造について，ROS 2 は通信仕様として DDS（Data Distribution Service）を採用している．DDS とは，OMG（Object Management Group）の制定した出版購読通信の標準規格である．DDS では出版購読通信に必要な API の仕様が決められており，その実装は複数のベンダにより提供されている．ROS 2 では，RMW（ROS MiddleWare）の層が異なる DDS 実装ごとの違いを吸収して通信の共通機能を提供している．次にユーザインターフェイスについて，ROS 2 ではアプリケーションの開発に用いるプログラミング言語の選択肢が複数用意されている．これは RCL（ROS Client Library）の層が ROS 2 の共通機能を API として提供し，各プログラミング言語向けのクライアントライブラリがその API を呼び出すことで実現している．RCL の API は C 言語で実装されており，多くの言語内から呼び出すことができる．

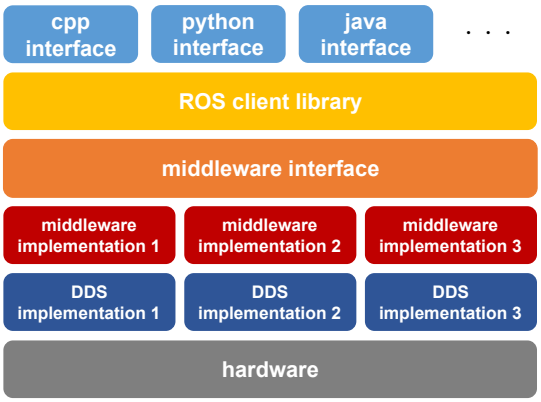

図 1 ROS 2 アーキテクチャの階層構造

2.1.2 メッセージ型

ROS 2 のメッセージは構造体として定義される．ここでは ROS 2 メッセージのために定義された構造体をメッセージ型と呼ぶ．メッセージ型はそのメンバとして基本型およびすでに定義されているメッセージ型，さらにそれらの配列を持つことができ，扱うメッセージ型は各 topic について定めることとなっている．これによりその topic が扱うべき情報をまとめて 1 つのメッセージ単位でやりとりすることができ，簡潔な通信システムモデルの構築が可能である．

ROS 2 メッセージの通信を行うプログラムを開発する際には，ROSIDL により指定される書式にて.msg ファイルを作成しメッセージ型を定義する必要がある．.msg ファイルは rosidl パッケージにより読み込まれ，メッセージ通信のためのファイルが生成される．生成されるファイルは主に次の 2 つである．

- 構造体定義ファイル
- type support と呼ばれるシリアライズ関数・デシリアライズ関数などを含む関数群

これを用いてメッセージ通信を行うプログラムの手順は次のようになる．

(1) パブリッシャノードおよびサブスクライバノードの作成時に，type support を登録
(2) 構造体ファイルで定義されるメッセージオブジェクトを publish 関数に渡し，take 関数から受け取る

各ノードに登録された type support は RMW から DDS に渡され，実際のメッセージ構造体オブジェクトのシリアライズ・デシリアライズは DDS 実装において行われる．rosidl パッケージは C，C++，Python などのクライアントライブラリインターフェイスの言語ごとに用意されている．例として Python 向けの rosidl パッケージで生成されるファイルは次の 2 つとなる．

Vol.2022-ARC-248 No.50
Vol.2022-SLDM-198 No.50
Vol.2022-EMB-59 No.50
2022/3/11

- 構造体定義クラスファイル（.py）
- Python のメッセージ構造体オブジェクト向け type support

Python 向け ROS 2 クライアントライブラリである Rclpy でのメッセージ通信では，DDS 実装が type support 関数を利用して Python のメッセージ構造体オブジェクトをシリアライズ・デシリアライズするという流れになっている．

また，メッセージ型の定義および使用にあたっては，生成したファイルなどをまとめてメッセージ型定義パッケージを作成し，そのパッケージをアプリケーション側が依存関係により読み込んで利用する方式が推奨されている．1 つのパッケージで複数のメッセージ型を定義することもでき，メッセージ型を利用する際には [パッケージ名]/[メッセージ型名] や [パッケージ名]/msg/[メッセージ型名] といった名前で識別される．

2.2 Elixir

Erlang は高い障害耐性および同時並行性を持つプログラミング言語である．Erlang では軽量プロセスを同時並列で動かし，エラーが生じた際には問題のあるプロセスのみを切り離して対処するという方針をとり，優れた障害耐性を獲得している．この方針を実現するために，Erlang のプロセスは OS のプロセスではなく Erlang VM と呼ばれる仮想マシンの上で生成・実行される．

Elixir は，この Erlang VM 上で動作する関数型言語である．Erlang の持つ並行性および耐障害性に加え，わかりやすい文法による習得容易性を備えている．Elixir では，関数は入力を出力に変換するものと捉える [4]．どのような処理を施すかではなくどのようにデータを変換するかに重点を置き，個々のシンプルな関数を組み合わせて機能を実現する柔軟で応用性の高い設計となっている．Elixir ではこうしたスタイルを支えるパターンマッチングやパイプ演算子などの構文を備えている．

Elixir はアクターと呼ばれる独立したプロセスの中でコードを実行する．プロセスからプロセスを生成することなども可能であり，プロセス同士のつながりはメッセージの送受信のみである．Elixir のプロセスは軽量な Erlang VM のプロセスを利用しているため，複数のプロセスを起動し並列に動作させることが容易になっている．

NIFs（Native Implemented Functions）は，C 言語で実装された関数を Erlang の関数として読み込む Erlang VM の機能である．NIFs は共有ライブラリとしてコンパイルされ，実行時に Erlang のコードにより読み込まれる．NIF コード内では Erlang のデータは ERL_NIF_TERM という構造体として定義される．NIF の引数および返り値の型は ERL_NIF_TERM であり，Erlang 関数として NIF を呼び出す際には引数および返り値はその中身に対応する Erlang のデータとなる．さらに，NIF のライブラリでは ERL_NIF_TERM および C 言語のデータ間の変換を行う関数が用意されている．また，NIF において構造体オブジェクトを扱うためには，NIF のロード時に型の登録を行い，NIF ライブラリのメモリ割り当て用関数を用いる必要がある．これにより，構造体オブジェクトへのポインタを Erlang のリソースオブジェクトとして保持することが出来，そのポインタを渡すことで同じモジュールに定義された NIF によって構造体オブジェクトへアクセスできる．

図 2 (a) 同一トピックへ出版購読を行う複数ノード通信モデル
(b) 異なるトピックへ出版購読を行う複数ノード通信モデル

2.3 Rclex

Rclex は現在開発されている Elixir 向けの ROS 2 クライアントライブラリである [2]．Elixir のプロセスモデルを活用しており，特長として高いスケーラビリティおよび耐障害性を持つ．またユースケースとして，多数のセンサを配置する大規模な監視システムや多数のデバイスが参加するクラウドロボティクスのサーバーサイドでの利用が想定されている．そのため多数のノードを作成して通信することを想定し，複数ノードの同時生成機能が提供されている．この機能では図 2 のように同一のトピックへ出版購読を行うノードの生成および各々が異なるトピックへ出版購読を行うノードの生成を容易に行うことができる．

ROS 2 はノードを機能単位としたシステムを構成するが，Rclex では一つの Erlang プロセスが ROS 2 ノードの処理を担う．これにより，Elixir の軽量プロセス大量生成機能を活用し，ROS 2 ノード数を増やした際の性能への影響を低減して高いスケーラビリティを実現している．また，Elixir のスーパーバイザによるプロセス監視機能を利用することで，問題の発生したノードの自動検知および再起動が実現され耐障害性にも優れている．

ROS 2 のクライアントライブラリインターフェースは C で実装された RCL の API を実行する形式がとられているが，Rclex では NIFs が RCL の API 実行を担っている．

しかし，現在 Rclex では通信可能なメッセージ型が文字列型のみであり，その適用範囲は限られたものとなっている．本論文ではこの課題の解決を目指し，Rclex において任意型メッセージの通信を実現する手法を提案する．

2.4 関連研究

文献 [5] では，開発初期段階の ROS 2 の通信性能を，ROS と比較しながら，いくつかの DDS 実装を用いて評価

Vol.2022-ARC-248 No.50
Vol.2022-SLDM-198 No.50
Vol.2022-EMB-59 No.50
2022/3/11

し，DDS 実装ごとの特性を検証している．文献 [6] では，ROS ノード間の通信における拡張性および信頼性の課題を，Erlang の通信フレームワークを用いて改善できると報告されている．文献 [7] では，メッセージ構造体が複雑な場合に，DDS 実装でのシリアライズ・デシリアライズのオーバーヘッドが増大するが，クライアントライブラリインターフェースの層でシリアライズ・デシリアライズを行うことで性能が改善することを指摘している．さらに，メッセージ構造体の複雑さに応じてオーバーヘッドが小さくなるようにシリアライズ・デシリアライズを行う層を選択する手法が提案されている．

3. 任意型メッセージの通信手法

Rclex において ROS 2 が対応するメッセージ型全てを扱えるようにすることで Rclex の適用範囲を広げ，さらにシステムモデルの簡素化およびそれによる開発コストの低減を目指す．この目的を達成するため，提案手法では Rclex 上のノードが他 ROS 2 ノードとの間で任意の ROS 2 メッセージの出版購読通信を行えるようにする．

3.1 設計方針

任意型メッセージの通信を実現するために Rclex に必要となる機能要件を整理し，設計の方針を示す．従来の Rclex では，標準の文字列型メッセージの出版購読通信が実現されており，すべてのメッセージ型に共通な通信処理機能の部分は本手法においても利用可能である．そのため，本手法では任意のメッセージ型に対し，型固有の処理機能を追加することで通信を可能にする方針をとる．ROS 2 の通信におけるメッセージ型に固有の処理は次の通りである．

- 出版通信

(1) パブリッシャ設定時，メッセージ型に対応する type support をパブリッシャに登録する
(2) 出版時，値の設定されたメッセージ構造体オブジェクトを publish 関数に渡す

- 購読通信

(1) サブスクライバ設定時，メッセージ型に対応する type support をサブスクライバに登録する
(2) 購読通知を受けた後，take 関数でメッセージ構造体オブジェクトを受け取り，値を読み取る

以上の処理を Rclex 上で実現するにあたり，まず本研究では RCL の publish 関数や take 関数に渡すメッセージオブジェクトとして C 言語の構造体オブジェクトを用いる方針をとる．これは，Rclpy などの従来のクライアントライブラリにおける，メッセージオブジェクトとしてインターフェース言語の構造体オブジェクトを用いる手法とは異なるものである．

提案手法の方針の利点として，次の事項が挙げられる．

- RCL より下の層に Elixir の要素を含まないことにより ROS 2 アーキテクチャの階層構造との整合性がとれる
- NIFs の機能を活用して C の構造体オブジェクトを扱う処理が容易に実現できる
- rosidl により生成される C の構造体定義ファイルや type support をそのまま利用できる

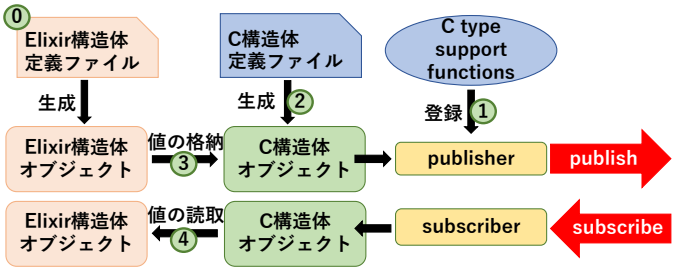

図 3 任意型メッセージの通信処理フロー

いっぽう，Rclex のアプリケーションのコード内で C 構造体オブジェクトをそのまま扱おうとすると，メンバの値の参照や代入を行う度に NIF を呼び出す必要がある．これはアプリケーションの開発者にとって煩雑である上，Elixir のコーディングスタイルとそぐわないという問題がある．そのため，本手法では扱うメッセージ型について Elixir の構造体定義を用意し，メッセージの内容の操作は Elixir 構造体オブジェクトに対して行う方針をとる．そして Elixir 構造体オブジェクトおよび C 構造体オブジェクトを対応付ける機能を提供することで，RCL 以下の層では C 構造体オブジェクトをメッセージとして扱えるようにする．

この方針に基づき設計したファイルおよび API を次に示す．

0. Elixir でのメッセージ構造体の定義ファイル
1. メッセージ型に対応する C の type support への参照を取得する関数
2. 空の C メッセージ構造体オブジェクトを生成し，Erlang リソースオブジェクトとして返す関数
3. Elixir メッセージ構造体のメンバの値を C メッセージ構造体オブジェクトのメンバの値にコピーする関数
4. C メッセージ構造体のメンバの値を Elixir メッセージ構造体オブジェクトのメンバの値にコピーする関数

設計されたファイルおよび API を用いて実現したい出版購読通信の流れを図 3 に示す．図中の各番号は上記のファイル・API の番号と対応している．

3.2 実装方法

3.2.1 メッセージ構造体定義

メッセージ型の情報から Elixir メッセージ構造体を定義する方法を説明する．.msg ファイルの Built-in-type および Elixir メッセージ構造体のメンバの型の対応を表 1 に示す．

この対応に基づいて Elixir の構造体定義モデュールを与

Vol.2022-ARC-248 No.50
Vol.2022-SLDM-198 No.50
Vol.2022-EMB-59 No.50
2022/3/11

える．表 1 の Elixir 値型について，integer は整数型，float は実数型を表す．また，Elixir では文字列表現としてバイナリによる表現および整数のリストによる表現が存在するが，今回メッセージ型の文字列型として整数のリストによる表現を用いる．さらに，Elixir はアトムとよばれる値型を持つ．アトムとは名前を表現する定数であり，コロン (:) から始まる文字列で表現される．Elixir の bool 値はアトムとして定義されているため，.msg ファイルの bool 型にはアトムが対応する．

3.2.2 関数実装

上述の API を Elixir 関数として実装する方法を示す．本手法では，各 API を C オブジェクトを扱う処理を担う NIF およびその NIF を呼び出す Elixir プロトコル実装を組み合わせることで実装する．Elixir のプロトコルとは，引数の型に応じて呼び出す関数の実装を変更する機能である．まずプロトコルの定義において関数名および引数のみを宣言し，その実装は別で与える．プロトコルの実装は引数の型を指定して与えられ，呼び出し時に第一引数の型に応じた関数実装が実行される．つまり，プロトコルの実装は引数の型ごとに複数用意することができる．本手法では Elixir プロトコルを用い，NIF を呼び出す関数をメッセージ型ごとに実装する方法をとる．これにより，すべてのメッセージ型についてアプリケーション側で呼び出す関数名を同一のものにすることができ，コードを明快にし，開発コストを低減することが期待できる．各 API における NIF および Elixir プロトコル実装の名前および機能を表 2 に示す．

3.2.3 ソースファイルの自動生成

以上で説明した構造体定義ファイルおよび関数を，メッセージ型を指定して自動で生成する．各メッセージ型について，生成すべきファイルは Elixir 構造体定義ファイル・NIF の C ソースファイルおよび C ヘッダファイル・Elixir プロトコル実装のソースファイルである．また，NIF を Elixir の関数として読み込むためには，NIFs を初期化するマクロへ関数の情報を渡し，さらに NIF と同関数名の Elixir での実装を与える．そのため，上記のファイル生成に加え NIFs の初期化マクロへ渡す関数情報の追記および NIF と同関数名の Elixir 関数定義の追記も行う必要がある．

本手法では，Rclex プロジェクトのコンパイル時に Make-

表 1 .msg ファイルの Built-in-type および Elixir の値型の対応．

Type name	Elixir
bool	atom
byte	integer
char	integer
float32,64	float
int8,16,32,64	integer
uint8,16,32,64	integer
string	[integer]
wstring	[integer]

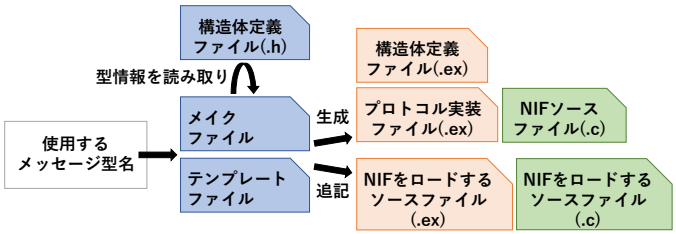

図 4 ソースファイル自動生成フロー

file を用いてこれらのファイル生成・追記を行う．まず，ROS 2 メッセージパッケージとして利用可能な状態であるメッセージ型の中から，扱いたいメッセージ型名を Makefile が読み込む．そして，扱うメッセージ型の C 構造体定義ファイルから，メンバの型および名前を読み込む．この情報を基に，上述の生成すべきファイルをテンプレートファイルから生成および追記する．このとき，扱うメッセージ型がそのメンバに他のメッセージ型を持つ場合，その型についても Elixir 構造体定義ファイルが必要となるため，同様にテンプレートファイルから Elixir 構造体定義ファイルを生成する．以上の流れを図 4 に示す．

3.3 提案手法による通信プログラム例

前節での実装方法を実際に適用し，他 ROS 2 ノードとの通信を行うプログラムの例を示す[*1]．図 5 は，パブリッシャおよびサブスクライバを 1 つずつ持つノードを起動し，パブリッシャが 100ms 周期で出版を行うプログラムである．まず，5 行目の `create_nodes` 関数で 10 個のノードを生成し，それらの名前を受け取る．次に 6,7 行目にて各ノードに対しパブリッシャおよびサブスクライバを生成し，それらの識別子を受け取る．このとき，パブリッシャやサブスクライバが利用するトピックおよび扱うメッセージ型の名前を指定する必要がある．メッセージ型名は [パッケージ名].Msg.[メッセージ型名] の形式で指定する．図 5 で使用されているメッセージ型について，Twist 型は並進速度および回転速度を表す型であり，linear および angular という名前の 2 つの Vector3 型メンバを持つ．Vecotr3 型は x, y, z の 3 つの実数値を持つ型である．String 型は data という名前の string 型のメンバのみを持つメッセージ型である．`create_publishers` 関数および `create_subscribers` 関数は入力のメッセージ型名から内部で type support 関数により type support を取得し，作成されるパブリッシャおよびサブスクライバに登録する．

そして 9 行目では 100ms ごとに `pub_callback` 関数を呼び出すタイマを起動する．`pub_callback` 関数では，次の処理を行う．

(1) 値を定めて Elixir メッセージ構造体オブジェクトをパブリッシャの数だけ作成

[*1] 現段階では配列を持つメッセージ型のうち，静的配列のみに対応していて，動的配列には対応していない

Vol.2022-ARC-248 No.50
Vol.2022-SLDM-198 No.50
Vol.2022-EMB-59 No.50
2022/3/11

表 2　実装する API の関数名および機能.

Elixir 関数名 対応する NIF	引数 返り値	機能
typesupport() get_typesupport_<msgtype>()	なし type support handle への参照	メッセージ型の type support を取得する
initialize() create_empty_msg_<msgtype>() init_msg_<msgtype>()	なし 初期化された C 構造体オブジェクトへの参照	メッセージ構造体を初期化
set() setdata_<msgtype>()	C 構造体オブジェクト，Elixir 構造体オブジェクト なし	Elixir 構造体メンバの値を C 構造体メンバへ格納
read() readdata_<msgtype>()	C 構造体オブジェクト Elixir 構造体オブジェクト	C 構造体メンバの値を Elixir 構造体メンバへ読み出す

(2) 初期化された C メッセージ構造体オブジェクトをパブリッシャの数だけ作成
(3) C メッセージ構造体オブジェクトに Elixir メッセージ構造体オブジェクトの値を格納
(4) メッセージを出版

Elixir メッセージ構造体は Rclex.[パッケージ名].Msg.[メッセージ型名] の名前のモジュールに定義される．また，Rclex.Msg モジュールの関数は，入力のメッセージ型名から空のメッセージ構造体を生成し，節 3.2.2 で述べたプロトコルに入力として与えることで，メッセージ型に対応するプロトコル実装を呼び出している．

さらに 10 行目では，サブスクライバの購読を開始する．サブスクライバが購読通知を受け取ると，出版されたメッセージを取得しそれを入力としてコールバック関数が起動される．図 5 の `sub_callback` 関数では，read 関数を用いてメンバの値を読み込んだ Elixir 構造体を取得し，String 型のメンバである data の値を表示している．

以上のメッセージ通信の流れは任意のメッセージ型について API 名などを変えず同様に適用可能であり，低い開発コストおよび高い汎用性が期待できる．

4. 評価

本章では，前章の提案手法を実装してメッセージ通信を実行した結果を示し，その通信性能から提案手法の有効性を評価する．また，Rclex の想定する多数のノードにより構成されるシステムにおける有効性を評価するため，パブリッシャやサブスクライバの数を増やして通信を行い，その通信時間を計測する．

4.1 評価環境

今回の性能評価でのメッセージ通信は同一の PC 内にて実行した．用いた PC のスペックは表 3 の通りである．メッセージ通信は従来の Rclex による通信および Rclex に提案手法を追加で実装したソフトウェアによる通信を行った．

```
1  defmodule Samplenodes do
2    def pubsub_main do
3      num_node = 10
4      context = Rclex.rclexinit()
5      {:ok, nodes} = Rclex.ResourceServer.create_nodes(
           context, 'sample_node', num_node)
6      {:ok, publishers} = Rclex.Node.create_publishers(
           nodes, 'GeometryMsgs.Msg.Twist', 'pub_topic', :
           single)
7      {:ok, subscribers} = Rclex.Node.create_subscribers(
           nodes, 'StdMsgs.Msg.String', 'sub_topic', :
           single)
8
9      {:ok, timer} = Rclex.ResourceServer.create_timer(&
           pub_callback/1, publishers, 100, "sample_timer")
10     Rclex.Subscriber.start_subscribing(subscribers,
           context, &sub_callback/1)
11   end
12   def pub_callback(publishers) do
13     n = length(publishers)
14     msgdata = %Rclex.Geometry.Msg.Twist{linear: %Rclex.
           Geometry.Msg.Vector3{x: 1.0, y: 1.0, z: 1.0},
           angular: %Rclex.Geometry.Msg.Vector3{x: 1.0, y:
           1.0, z: 1.0}}
15     msg_list = Rclex.Msg.initialize_msgs(n, 'GeometryMsgs
           .Msg.Twist')
16     Enum.map(msg_list, fn msg ->
17       Rclex.Msg.set(msg, msgdata,'Geometry.Msg.Twist')
18     end)
19
20     Rclex.Publisher.Publish(publishers, msg_list)
21   end
22   def sub_callback(msg) do
23     received_msg = Rclex.Msg.read(msg, 'StdMsgs.Msg.
           String')
24     IO.puts("received msg : #{received_msg.data}")
25   end
26 end
```

図 5　パブリッシャおよびサブスクライバを起動するプログラム例

4.2 単一のパブリッシャおよびサブスクライバ同士の通信

Rclex のパブリッシャおよびサブスクライバを 1 つずつ別 OS プロセスにて起動し，1 秒周期で 500 回の出版を 3 つの条件で行った．出版するメッセージは Twist 型メッセー

Vol.2022-ARC-248 No.50
Vol.2022-SLDM-198 No.50
Vol.2022-EMB-59 No.50
2022/3/11

ジまたは Twist 型の情報を文字列に変換した文字列型のメッセージである．Twist 型は 2 つの Vector3 型のメンバを持ち，Vecotr3 型は 3 つの実数値のメンバを持つ．行った通信の 3 つの条件は次の通りである．

(1) 従来の Rclex において，Twist 型の情報を文字列に変換して文字列型のメッセージ通信を行う
(2) 提案手法を実装した Rclex において，Twist 型の情報を文字列に変換して文字列型のメッセージ通信を行う
(3) 提案手法を実装した Rclex において，Twist 型のメッセージ通信を行う

なお，Twist 型メッセージの 6 つの実数値は Erlang 標準ライブラリの関数により生成された乱数を用いた．

本研究では，2 種類の計測区間で評価を実施した．まず，生成した実数値を出版するメッセージに格納する処理の直前から，購読したメッセージから実数値を読み出す処理の直後までの時間を計測した．次に，メッセージを出版する直前から，メッセージ購読のコールバック関数が起動された直後までの時間を計測した．これらの計測区間によって，メッセージ型の情報の変換または型ごとの通信時間を区別して評価できる．計測結果を図 6 に示す．

図 6 より，条件 1 および条件 2 での通信時間にはほとんど差が無く，条件 3 での通信時間は他の 2 条件と比べその平均は小さく，ばらつきは大きくなっていることがわかる．条件 1 および 2 において通信時間の差がほとんどないことから，本研究で追加したプロトコル実装による NIF 呼び出しなどの機能のオーバーヘッドは小さいと考えられる．また，条件 3 において通信時間の平均が他条件よりわずかに小さくなった要因としては，Twist 型の情報を文字列に変換したデータよりも Twist 型のままの方がデータサイズが小さいことが挙げられる．さらに，2 つの計測区間についてはほぼ同様の結果が得られ，今回の通信では Twist 型構造体や文字列へのデータの格納および読み出しのオーバーヘッドは小さいものだった．以上より，提案手法は 1 対 1 のノード間通信では性能面においても有効であるといえる．

4.3 単一パブリッシャおよび複数サブスクライバの通信

Rclex のパブリッシャ 1 つと複数のサブスクライバを起動し，前節と同じ 3 つの条件で，それぞれサブスクライバの数を 20,40,60,80,100 と変えて通信を実行した．出版は 1 秒周期で 200 回行い，前節と同様にデータの格納から読み出しまでの区間 1 と，出版から購読までの区間 2 で通信時間の計測を行った．各試行での計測結果の平均を図 7 のグラフに示す．なお，各試行の 200 回の出版のうち，はじめの数回の出版では通信時間が極端に大きくなる場合が見られたため，通信時間の平均値ははじめの 10 回の出版を除いた 190 回の出版についてのものを用いた．

表 3 評価に用いた PC の環境．

OS	Ubuntu20.04.3
CPU コア	Intel Core i7
CPU コア数	4
論理プロセッサ数	8
クロック周波数	1.80GHz
メモリ	16GB
ROS 2 version	Foxy
DDS	FastRTPS
Elixir	1.9
Rclex	0.5.2

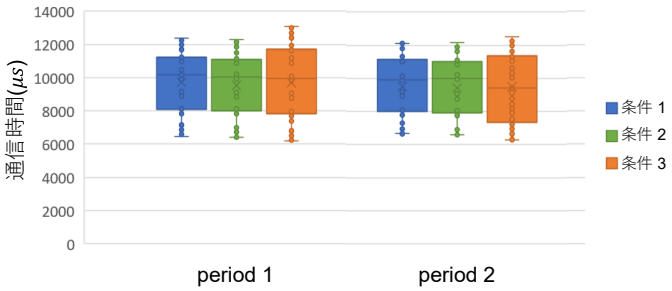

図 6 単一パブリッシャおよびサブスクライバのメッセージ通信時間

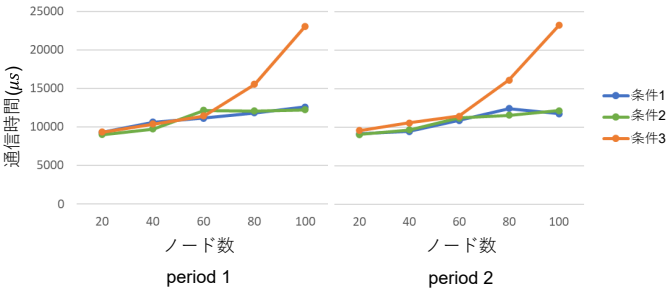

図 7 サブスクライバ数を変化させたときのメッセージ通信時間

図 7 より，条件 1 と条件 2 ではノード数によらずその平均値は同程度であるが，条件 3 ではノード数が 80 を超えた範囲において他の条件より通信時間の平均が増大していることがわかる．この傾向は 2 つの計測区間で同様に見られたことから，条件 3 での通信時間増大の要因は，出版から購読までの処理の中に存在すると考えられる．その中のメッセージ型固有の処理としては，メッセージのシリアライズ・デシリアライズが挙げられる．Twist 型は入れ子構造であるために，シリアライズ・デシリアライズのオーバーヘッドが文字列型と比べ大きく，ノード数が増えることでその影響が増大していると予想されるが，詳しい原因究明はできていない．

4.4 複数パブリッシャおよび単一サブスクライバの通信

Rclex の複数のパブリッシャおよび 1 つのサブスクライバを起動し，節 4.2 と同じ 3 つの条件で，それぞれパブリッシャの数を 20,40,60,80,100 と変えて 5 セットの通信を行った．出版は 1 秒周期で 200 回行い，生成した実数値を出版するメッセージに格納する処理の直前から，購読したメッセージから実数値を読み出す処理の直後までの時間を計測

Vol.2022-ARC-248 No.50
Vol.2022-SLDM-198 No.50
Vol.2022-EMB-59 No.50
2022/3/11

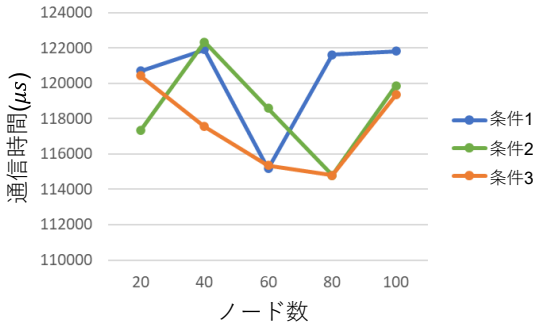

図 8　パブリッシャ数を変化させたときのメッセージ通信時間

した．計測結果の平均値を図 8 に示す．ただし，ノード数あたり 200 回の出版のうち，購読されたことが確認できたものは少なく，多くのメッセージが受け取られずに欠損してしまっていた。表 4 に示すとおり，ノード数によらずすべての試行で購読回数が 1100 程度となっている．

購読回数が限られていた原因として，出版のタイマコールバック関数内で作成した複数のノードによる同一トピックへの出版を一斉に行っているために，出版されたメッセージが購読前に上書きされてしまうことなどが考えられる．しかし，200 回の出版コールバックのうち 1 ノードからの出版のみが受け取られる回および 10 ノード程度の出版が受け取られる回が交互に生じる現象が観察され，具体的な原因は現時点では究明できていない．

5. 結論

本研究では，Elixir 向け ROS 2 クライアントライブラリである Rclex の適用範囲拡大を目的とし，ROS 2 の提供する任意のメッセージ型による出版購読通信の手法を提案した．これにより，Rclex により構築可能なシステムの幅を広げるとともに，任意型メッセージを扱う他 ROS 2 ノードとの通信システムも実現可能となる．また，提案手法ではメッセージのインターフェースとして Elixir 構造体や Elixir のプロトコル機能を用いることで，任意型のメッセージについて画一的な流れでのプログラミングが可能であり，開発コストの低減への効果を期待できる．さらに，Elixir メッセージ構造体をインターフェースとし，RCL との接点では C メッセージ構造体を用いることで，ROS 2 の階層構造や Erlang の機能と整合したプログラミング構造を実現している．

提案手法を実装し，実際に通信を行うことでその性能を評価した．単一のパブリッシャとサブスクライバによる通信では提案手法の通信性能は従来手法の性能と大差なく，提案手法による Rclex の適用範囲拡大の有効性が示されたといえる．サブスクライバのノード数を増加させた通信では，ノード数が増加するにつれ提案手法のオーバーヘッドが増大することが確認され，Rclex が想定する多数のノードを含むシステムへの適用にあたっては，許容される通信性能などを検証した上で用いるメッセージ型を選択する必要が生じる．しかし，すでに存在するパッケージの ROS 2 ノードとの通信は提案手法により容易に実現できる点において，提案手法の有効性は確かなものと考えられる．

提案手法の技術的課題として，動的配列を持つメッセージ型への対応，および API を追加するメッセージ型の指定を現状の Rclex プロジェクト内のファイルを書き換える形ではなく，アプリケーションプロジェクト内で行うようにすることが挙げられる．

また今後の展望として，任意型メッセージの通信を実現させながらノード数増加の影響を小さくする通信方法の開発が挙げられる．具体的には，文献 [7] で提案されるように，メッセージ型の構造が複雑な場合にシリアライズ・デシリアライズを Rclex の層で行うことで，DDS の層でのシリアライズ・デシリアライズのオーバーヘッドを削減できることが期待される．

表 4　パブリッシャ数を変化させたときのメッセージの購読回数

ノード数	条件 1	条件 2	条件 3
20	1098	1094	1100
40	1100	1092	1098
60	1098	1092	1106
80	1104	1094	1110
100	1102	1096	1120

謝辞　本研究の一部は，JST さきがけ JPMJPR18M8 の支援ならびに国立研究開発法人情報通信研究機構の委託研究（04001）により得られたものである．

参考文献

[1] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A. Y. et al.: ROS: an open-source Robot Operating System, *ICRA workshop on open source software*, Vol. 3, No. 3.2, Kobe, Japan, p. 5 (2009).
[2] 今西洋偉，高瀬英希：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，情報処理学会研究報告，Vol. 2020, No. 48, pp. 1–8 (2020).
[3] 近藤豊：ROS2 で始めよう 次世代ロボットプログラミング，技術評論社 (2019).
[4] Thomas, D.: プログラミング Elixir，オーム社 (2020). 笹田耕一，鳥井雪 訳.
[5] Maruyama, Y., Kato, S. and Azumi, T.: Exploring the Performance of ROS2, *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/2968478.2968502 (2016).
[6] Lutac, A., Chechina, N., Aragon-Camarasa, G. and Trinder, P.: Towards Reliable and Scalable Robot Communication, *Proceedings of the 15th International Workshop on Erlang*, Erlang 2016, New York, NY, USA, Association for Computing Machinery, p. 12–23 (online), DOI: 10.1145/2975969.2975971 (2016).
[7] Jiang, Z., Gong, Y., Zhai, J., Wang, Y.-P., Liu, W., Wu, H. and Jin, J.: Message Passing Optimization in Robot Operating System, *International Journal of Parallel Programming*, Vol. 48, No. 1, pp. 119–136 (2020).